2005

Combining generic programming with vector processing for machine vision

Bing-Chang Lai
University of Wollongong

Combining Generic Programming with Vector Processing for Machine Vision

A thesis submitted in fulfilment of the
requirements for the award of the degree of

Doctor of Philosophy
(Computer Science)

from

UNIVERSITY OF WOLLONGONG

by

Bing-Chang Lai, BCompSc (Hons)

School of Information Technology and Computer Science
2005

Declaration

I, Bing-Chang Lai, hereby declare that I am the sole author of this thesis. I also declare that the material presented within is my own work, except where duly acknowledged, and that I am not aware of any similar work either prior to this thesis or currently being pursued.

_____

Bing-Chang Lai

_____

Date

Abstract

This thesis addresses the integration of generic programming with vector processing for machine vision. While generic libraries have been shown to provide near optimal performance without sacrificing flexibility and adaptability, current generic libraries do not utilise the vector processing unit (VPU), nor can they be vectorised directly. Generic vectorised libraries require a mechanism for expressing vectorised algorithms independently of the VPU. This is a problem since different VPUs can have different instructions and different limitations; programs written to use one vector technology are not portable to other vector technologies. Lastly, most existing machine-vision libraries do not provide image capture from sequence grabbers; the programmer has to use another library to capture images, and to supply additional code to enable the two libraries to work together.

To allow vectorised, machine-vision algorithms to be portable across different vector technologies, this thesis proposes the use of an abstract VPU. The abstract VPU represents a set of real VPUs with a virtual VPU that has an idealised instruction set and constraints common to the real VPUs being represented. An abstract VPU, named Virtual Vector Machine (VVM), was developed to support generic programming. Different methods of implementing VVM were evaluated against hand-coded AltiVec (a vector technology found in PowerPC G4 and G5 processors) and scalar programs. The implementation chosen has no significant overheads when processing VVM vectors with a single AltiVec vector or a single scalar when compiled using Apple GCC 3.1 20021003. VVM vectors with a single AltiVec vector or scalar cover all byte AltiVec vectors in AltiVec mode and all types in scalar mode. When processing VVM vectors that use more than one AltiVec vector, the VVM implementation chosen is within 24% slower than a hand-coded program.

Vectorised algorithms are difficult to implement, because they handle VPU-specific issues such as memory alignments, edges and prefetching. Thus, to reduce the number of algorithms required, a categorisation of image processing operations based on input-to-output correlation is proposed. This categorisation maps easily to generic programming and provides implementation hints. The categorisation scheme separates image processing operations into three categories, which this thesis refers to as quantitative, transformative and convolutive operations. Quantitative operations require one input element to produce zero or more output elements. Transformative operations require one input el-

ement to produce one output element. Convolutive operations produce a single output element from a rectangle of input elements. Each category requires only one general algorithm. Variations of the algorithm to handle differing input and output set combinations are also required.

The generic, vectorised, machine-vision library, Vectorised Vision (VVIS), developed in this thesis uses the abstract VPU (VVM) and the three categories to provide cross-platform, vectorised algorithms. Because the division of duties used by existing generic libraries is unsuitable for vectorisation, two new divisions of duties are proposed and their performance is evaluated. Generic, vectorised algorithms for each category were evaluated against hand-coded programs and the speedup gained by using VVIS in AltiVec mode instead of scalar mode was collated. The VVIS implementation is comparable to hand-coded AltiVec and scalar programs when operating on single-channel byte images when processing quantitative and transformative operations. For convolutive operations, the final VVIS implementation is twice as slow in AltiVec mode when processing single-channel byte images, because the hand-coded AltiVec program did not need to support variable kernel sizes. In scalar mode, the final VVIS convolutive algorithm is comparable to hand-coded scalar programs. VVIS was slower than hand-coded programs when processing non-byte images, because of overheads introduced by VVM. For the transformative operations tested, VVIS, when executed in AltiVec mode instead of scalar mode, provides at most a four fold speedup depending on the operation when processing single-channel byte images. For convolutive operations tested, VVIS provides a speedup of approximately 1.5 times when processing single-channel byte images, despite using `signed short` internally; the VVM implementation used had noticeable overheads when operating on `signed shorts` in AltiVec mode.

Results show that a generic, vectorised, machine-vision library generally have comparable performance to hand-coded programs when processing single-channel byte images. Because of overheads introduced by the VVM implementation, VVIS does not have comparable performance for all image types. Most image processing operations however use either single-channel byte or multi-channel byte images.

Acknowledgements

I would like to thank my primary supervisor, Associate Professor Phillip John McKerrow, for providing guidance and support, and allowing me the freedom to pursue my interests; and my secondary supervisor, Dr Jo Abrantes, for sharing her valuable experiences and providing support while Phillip was away. I would also like to thank Associate Professor Neil Gray for taking time to review and to provide comments on this thesis.

I would like to thank my parents for giving me the opportunity to undertake a Ph.D. at the University of Wollongong and for raising me to be a responsible, up-standing citizen. I would also like to express my gratitude to my elder brother, Hsuan-Cheng Lai, for checking part of this thesis.

Finally, I would like to thank Chia-Yun Chen, my girlfriend, who kept me company, and motivated me throughout my thesis. I would also like to mention my cat, Jaime, who re-wrote portions of my thesis by walking on my keyboard.

This research was supported by a grant from the Apple University Development Fund in Australia.

Publications from this Thesis

Lai, B., McKerrow, P.J. & Woolley, D. (2001), Developing a Java API for digital video control using the FireWire SDK, *in* N. Smythe, ed, 'e-Xplore 2001: a face to face odyssey', Apple University Consortium.

Lai, B. & McKerrow, P.J. (2001), Programming the Velocity Engine, *in* N. Smythe, ed, 'e-Xplore 2001: a face to face odyssey', Apple University Consortium.

Lai, B. & McKerrow, P.J. (2001), Image processing libraries, Australasian Conference on Robotics & Automation.

Lai, B., McKerrow, P.J. & Abrantes, J. (2002), Vectorized machine vision algorithms using AltiVec, Australasian Conference on Robotics & Automation.

Lai, B. & McKerrow, P.J. & Abrantes, J. (2005), The abstract vector processor, Microprocessors and Microsystems, Accepted June 10th 2005.

Lai, B. & McKerrow, P.J. (2005), Design of a generic, vectorised, machine-vision library, *in* Cutting Edge Robotics, ISBN 3-86611-038-3, International Journal of Advanced Robotic Systems, DAAM International Publishing, Mammendorf, Germany, pp 153-174.

Contents

List of Figures

List of Tables

List of Algorithms

Chapter 1

Introduction

Machine vision refers to the process by which useful information is extracted autonomously by a machine through the use of vision. Machine vision covers image acquisition, and the extraction of useful information from the acquired images. Machine vision is used extensively in many scientific disciplines, such as astronomy, biology and computer science. It is used in tasks such as counting the number of stars in the sky, extracting the DNA sequence from digitised DNA sequencing gel for storage in databases and in colouring images of planets taken by monochrome cameras aboard the Voyager (Parker 1994). Apart from scientific disciplines, machine vision is also used heavily in industry, in quality control and manufacturing (Davies 1990).

Images have typically been acquired through the use of specialised hardware image capture cards. After acquisition, these images are then processed and analysed using specialised hardware cards. The computer is only responsible for obtaining user input, and to display the output.

The increasing popularity of key technologies on the desktop computer makes it possible to undertake more and more sophisticated machine-vision applications on the desktop. These key technologies are the vector processing unit (VPU) and FireWire cameras. VPUs allow more images to be analysed in the same time. FireWire allows a high-quality real-time feed from cameras into the computer. These two technologies together reduce the need for specialised hardware cards.

Generic programming was introduced by Musser and Stepanov (Musser & Stepanov 1994, 1989) as a paradigm for structuring flexible libraries. The key idea of generic programming is the separation of the algorithm from the data that it operates on. With this separation in place, it is possible for the same algorithm to be used on user-defined data. Vision with Generic Algorithms (Köthe 2001, 1999, 2000*c*, 1998, 2000*a*,*b*) is a generic, image processing library that lacks VPU and support for image capture via sequence grabbers. While it would be possible to add image capture support to VIGRA, adding VPU support requires structural changes.

Providing the services of the VPU and FireWire cameras in a generic package is the

focus of this thesis. This thesis investigates the implementation of a generic, vectorised, machine-vision library. To address the issue of creating generic VPU code, the author proposes the abstract VPU. To reduce the amount of algorithms that the library will need, a categorisation scheme based on input-to-output correlation is proposed.

1.1 Problems and proposed solutions

This thesis seeks to address one main problem: the integration of generic programming with vector processing.

Integration of generic programming with vector processing: Existing generic image-processing libraries, like VIGRA, do not use the VPU. On the other hand, existing vectorised image-processing libraries, like VSIPL, are not generic. A generic library is desirable because generic libraries are able to provide flexibility and performance comparable to hand-coded programs. Using the VPU is desirable for image processing because the VPU can increase the throughput of many image-processing operations.

Existing generic libraries cannot use the VPU efficiently because iterators do not provide information about the relative positions of the pixels in memory. Without this information, there is no mechanism for deciding whether a vector can be loaded or stored efficiently. While the iterator could return vectors, because vectors in desktop computers typically have fixed sizes, only image sizes that are exactly divisible by the vector size can be processed correctly. Other problems associated with vector programs, such as alignment, edges and prefetching also have to be considered. Another important consideration is how the user would be expected to write functors for such a library, since VPU instructions are not portable between different architectures.

The proposed solution uses an abstract VPU to allow users to write functors that are potentially portable to different architectures, a categorisation scheme based on input-to-output correlation to reduce the number of algorithms that are required, and a new division of duty for generic, vectorised libraries. Using the abstract VPU allows the library to use an idealised instruction set. Reducing the number of algorithms required is desirable because vectorised algorithms are generally more difficult to implement, especially since the proposed solution uses separate scalar and vectorised implementations for each algorithm. The new division of duty is the cornerstone to integrating generic programming with vector processing.

Other problems that were partially addressed as a result of the investigation into the main problem are the non-portability of vectorised machine-vision algorithms, and the poor integration of image capture with machine-vision libraries.

Non-portability of vectorised machine-vision algorithms: Generic programming implies that algorithms and functors should be generic and thus not tied to any particular hardware. However, vector programs that target different vector technologies are typically incompatible, due to different instructions and different limitations. This is illustrated in VSIPL AltiVec implementations from DNA (DNA 2001, Ded 2001), TransTech (Tra 2001), MPI (MPI 1999-2000) and SKY (SKY 1999). While code for these libraries are not available to the author, their practice of hand-coding for each processor can be inferred from the vector support of their libraries. All the libraries, except the one from MPI, support only the AltiVec processor, with other versions (if any) being slated for a later release.

What is needed to enable generic, vectorised, machine-vision libraries, is a mechanism for expressing vector programs in a generic fashion. Thus the abstract VPU is proposed. The abstract VPU represents a family of real VPUs with a virtual VPU that has an idealised instruction set and constraints common to the real VPUs being represented.

Poor image capture integration with machine-vision libraries: Image capture from sequence grabbers is typically not straightforward and not integrated with machine-vision libraries. A vectorised, machine-vision library performing image capture should also consider data conversions and processing live video feeds. Time would be wasted if the image capture component stores data in a vastly different manner from the library. In addition, the library should allow the programmer to process frames as they become available. While most machine-vision libraries do not support image capture, adding image capture to these libraries should not be overly difficult. Adding VPU support would be more difficult though. LabVIEW (Nat 2005), a visual programming language developed by National Instruments, provides image capture routines.

To address this problem, the generic, vectorised, machine-vision library developed as part of this thesis includes image capture routines. Conversion costs might still be associated with changing the image from the format provided by the image capture routines to a form suitable for vector processing.

1.2 Thesis structure

The chapters start with background information, followed by discussions on the abstract VPU, the categorisation of operations, and the generic, vectorised, machine-vision library. The chapters are summarised below:

Chapter 2, Machine-Vision and Image-Processing Libraries: In this chapter, some currently available image-processing libraries are discussed and evaluated.

Chapter 3, Vector Processor Programs: This chapter describes how to write programs that use the VPU, what issues a VPU program needs to handle, and currently available methods of using the VPU. It also includes an empirical investigation into how different VPU issues affect the performance of a VPU program.

Chapter 4, Implementation Techniques: This chapter presents several uncommon C++ implementation techniques that are used in this thesis.

Chapter 5, Abstract Vector Processor: The abstract VPU is detailed in this chapter. The intrinsic and desired characteristics of an abstract VPU are discussed. The differences between an abstract VPU and other methods of programming the VPU are also presented.

Chapter 6, Vector Virtual Machine's Design: An abstract VPU, called Virtual Vector Machine (VVM), is described in this chapter. The specification and the reasons behind included features are discussed. VVM is designed to enable the creation of a generic, vectorised, machine-vision library.

Chapter 7, Vector Virtual Machine's Implementation: This chapter investigates whether the VVM specification can be implemented using only Standard C++, and what the overheads of such implementations are.

Chapter 8, Categorisation of Operations based on Input-to-Output Correlation: A categorisation based on input-to-output correlation is presented in this chapter. Some of the criteria used to categorise, and the implications of such criteria on a library based on generic programming are presented. This chapter concludes with a discussion of the categories to be used in a generic, vectorised, machine-vision library. Applying this categorisation reduces the number of algorithms required. Since vector programs are not necessarily easy to write, this eases the implementation of the generic, vectorised, machine-vision library.

Chapter 9, A Generic, Vectorised, Machine-Vision Library: The design and implementation of a generic, vectorised, machine-vision library, called Vectorised Vision (VVIS), is presented in this chapter. Reasons are given for why current generic programming libraries are unsuitable for vectorisation and viable solutions are presented.

Chapter 10, Vectorisability of Machine-Vision Algorithms: For each category identified, this chapter discusses the implementation of a generic, vectorised algorithm for VVIS, and compares it with hand-coded programs. The functor requirements and the speedup attained for some functors running in AltiVec mode over scalar mode are described.

Chapter 11, Examples: Three examples of how to use VVIS are presented.

Chapter 12, Conclusions: This chapter summarises the main contributions and limitations of the research described in this thesis. Future directions are also suggested.

This thesis contains three appendices, which are described below:

Appendix A, Glossary: A short description of terms used in this thesis is presented.

Appendix B, VVM's AltiVec Vector-Processor Implementation: This appendix describes how the AltiVec vector-processor implementation of the VVM implementation used by the generic, vectorised, machine-vision library maps VVM types to AltiVec types, and VVM functions to AltiVec functions .

Appendix C, Constant Scalar Count: The reasons why constant scalar count is required for generic programming are discussed in this appendix. Since VVM's purpose is to enable the creation of generic, vectorised libraries, any requirements of generic programming also apply to VVM.

Chapter 2

Machine-Vision and Image-Processing Libraries

Image-processing routines are currently available in many different libraries. These libraries differ in programming language support, approach to performing image processing, execution performance, functionality, amongst others.

To help highlight the differences between these libraries, this chapter starts with a description of the evaluation criteria that were used to analyse each library. After discussing the criteria, four different libraries are discussed. These libraries are Vector, Signal and Image Processing Library, Vision with Generic Algorithms, Intel Performance Primitives and National Instrument's IMAQ Vision library. These particular libraries are discussed because they are referred to later in this thesis as a source of inspiration. Of all these libraries, only Vision with Generic Algorithms is based on generic programming paradigms.

2.1 Evaluation Criteria

Six criteria are used to analyse the libraries to gain a better understanding the strengths and weaknesses of each library. Understanding the strengths and weaknesses of each library provides insight into the repercussions of design decisions.

1. Performance

   Performance is important in machine vision because there are many "ideal" operations that just take too long to complete. Coupled with real-time needs of some applications, this makes performance essential. This is where the VPU comes in. The AltiVec VPU should theoretically allow for 4 to 16 fold performance increases. The VPU does not change the scalability of an algorithm, though it will allow the same algorithm to handle larger data sets practically.

There are other techniques available, apart from using the VPU, that can increase performance. These other techniques include using more than one processor, multi-threading, and clustering. Most of these other techniques though are only suitable if the operation runs for extremely long periods.

Typically, libraries based on generic programming provide their benefits with little performance degradation (Köthe 1999).

2. Ease of use

   Difficult to use libraries tend to be used less. In today's world where hundreds of computer languages and libraries exist, there is simply too much to learn and too little time to learn it in. Therefore the ideal library should be as easy to use as possible.

   Libraries that use low-level C interfaces are given the lowest mark for ease of use. This is followed by libraries based on generic programming and object-oriented libraries. Languages based on visual programming are given the highest marks because they should be the easiest to use.

3. Adaptability

   The library needs to be easy to adapt or extend to work in the user's environment. Adaptability makes the user's code more legible and maintainable, since it helps reduce the number of different types that need to be dealt with. It allows the user to introduce the use of the library to existing applications gradually.

   Generic programming libraries are the most adaptable because of the separation of data from algorithms. This separation allows algorithms to operate on any data, as long as it satisfies the criteria set by the algorithm. Once the criteria required have been fulfilled by the user's data, the algorithm can operate directly on the user's data. One advantage generic libraries have over object-oriented libraries is that the user's data do not have to inherit from any library-defined classes. While object-oriented libraries are adaptable through the use of inheritance, the user will sometimes have to re-implement the algorithms because the algorithms are tied to the data that they operate on. Low-level C interfaces are not adaptable, because the data must be provided in the format required by the library. The library cannot use any user-defined formats. Such libraries typically offset this by accepting a large number of different formats.

4. Functionality

   A machine-vision library should have routines to aid image processing, image analysis and image acquisition. Image-processing libraries typically have routines for image processing and analysis, with image capture being the responsibility of other

libraries. Some of the libraries discussed here are actually image-processing libraries.

Functionality refers the the number of different things that the library can do. For the purposes of this thesis, for full marks, the library should provide image acquisition functions in addition to image processing and analysis. Images can be acquired from a camera or from files.

5. Language Support

   This refers to the number of programming languages that can use the library, either directly or indirectly.

   Direct usage means that the library is usable without any need for additional compilation steps. For example, a C library can be called directly in C, C++, Objective-C (App 2002*h*), Objective-C++ (App 2002*h*) and D (Dig 2004). However a C++ library can only be called directly in two of them: C++ and Objective-C++. Because of this, a C library will have higher marks than one based on C++.

   Indirect usage refers to cases where the library can be used through the use of additional libraries or compilation steps. For example, Java can call C through JNI. C is typically callable indirectly by many programming languages. C++ is also callable indirectly from many programming languages if it exports its functions as C-style functions. Calling the routines indirectly means that the library cannot be used as easily.

6. Platform Neutrality

   There are two kinds of platform neutrality — hardware and operating system. Libraries are usually only concerned with being operating system neutral. Machine-vision systems are also concerned about hardware neutrality, because different hardware have different features that can impact the algorithms used.

   The ideal library should be both hardware and operating system neutral in order to allow more developers to use it. This however contradicts with the use of the VPU to increase performance. Currently, VPUs in general are very different from each other. The library should probably at least consider the issue; perhaps it should be built in a manner conducive to porting.

2.2 Vector, Signal and Image Processing Library

The aim of the Vector, Signal and Image Processing Library (VSIPL) (*VSIPL website* 2001, Geo 2001) is to provide an API for vector, signal and image processing that does not depend on any particular hardware platform. The project was started by the Defence

Advanced Research Projects Agency (DARPA) for the development of the Tactical Advanced Signal Processor Common Environment (TASP COE), which required standards for some of its key Application Programming Interfaces (APIs). The problem was that the military wished to use commercial products in developing military weapons systems. However commercial product configurations were not stable over the life cycle of a typical military weapons system. A standard API, which commercial companies could follow, was created to reduce the amount of re-coding needed by the military when a new commercial product is available.

VSIPL is currently controlled by a forum of commercial companies and universities, including Intel, Silicon Graphics, MIT Lincoln Laboratory and Georgia Tech/GTRI among others. These member organisations meet a few times a year to discuss directions for the VSIPL API. The VSIPL API is currently in version 1.01, and there is a proposal pushing VSIPL towards a real object-oriented design.

Specific VSIPL libraries are referred to as "Core" or "Core Lite" profiles. The "Core" profile includes most of the signal processing and matrix algebra functions, while "Core Lite" includes a smaller subset, suitable for vector-based signal processing applications. Apart from these profiles, VSIPL also defines two versions of libraries referred to as development and performance. Development libraries run slower, but contain extra code for error reporting. VSIPL-compliant library suppliers may provide either one, or both versions.

2.2.1 Programming VSIPL

VSIPL consists of a large number of C functions with different versions for different types and different precision. Even though the API itself does not use an object-oriented language like C++ or Java, it is designed with an "object-oriented" view. Structs are used to represent objects. Implementations of VSIPL are allowed to add their own data to these objects, though they should not be exposed[1].

VSIPL functions work with blocks and views. Blocks consist of data arrays, which is the actual memory used to store the data, and a block object, which stores information that allows VSIPL to access the data array. Views of data can be created, and these views consists of a block and a view object, which stores information that allows VSIPL to access the data of interest stored in the block.

Blocks and views are VSIPL data structures that are opaque to the user. The user is not allowed to directly access any information inside blocks and views. In addition, creation and deletion of blocks and views are handled by VSIPL. Data arrays on the other hand, exist in one of two logical data spaces - the user data space, and the VSIPL data

[1]This is done in C using incomplete type definitions (see "NOTE TO IMPLEMENTORS" in (Geo 2001)).

space. Basically, only the user is allowed to change the data array when it is in the user data space, and vice versa. Moving data arrays between these two logical data spaces may occur penalties; these penalties, if any, depend on the implementation.

VSIPL provides functions for operations such as block allocation, basic scalar operations, basic vector operations, random number generation, signal processing and linear algebra. Since VSIPL tries to provide a hardware-neutral API, functions are defined for virtually every scalar mathematics operation, excluding basic operations like addition and subtraction. Vector operations are even more complete than scalar functions, with operations from basic operation like addition and subtraction to cosine and tangent to selection operations. The available image-processing functions make no indication of whether they use vector or scalar implementations. Therefore, it is up to the library supplier to decide how to process them.

2.2.2 Evaluation

1. Performance (●●●●●)

   VSIPL itself is simply an API and therefore has no standard implementation. However, because VSIPL is endorsed by many organisations, there are implementations of VSIPL available that interface with hardware and can therefore provide good performance. The hardware supported by the implementations cover not only VPUs, but also include specialised image-processing cards. Tra (2001), DNA (2001), MPI (1999-2000), and SKY (1999) provide VSIPL libraries that use the AltiVec VPU.

2. Ease of use (●●○○○)

   VSIPL uses the C language because it desires to be as platform neutral as possible. In using the C language, it sacrifices some of the newer programming paradigms, such as object-oriented programming, which make programs easier to use. VSIPL however does try to be easy by using "object-oriented" concepts (Geo 2001). At the end of the day however, VSIPL is not the easiest library to use. For example, all VSIPL object creation and deletion is handled by the user exclusively, and there are many different types.

3. Adaptability (●○○○○)

   VSIPL is not highly adaptable. It does not support any user-defined types.

4. Functionality (●●●○○)

   VSIPL has only image-processing functions. Analytical and acquisition features are missing. It provides no functions for capturing images from a camera or even loading from files. It however provides other functions such as vector functions.

5. Language Support (●●●●●)

   VSIPL is provided for the C language. The C language is available directly in other languages like C++ and indirectly in virtually all languages.

6. Platform Neutrality (●●●●●)

   This is VSIPL's strongest point, because it is one of its primary objectives. VSIPL was designed to be portable not only to desktop computers, but to embedded computers as well. For example, VSIPL has functions for many "standard" functions like cosine and sine of scalars.

2.2.3 Image inversion example

The following code example shows how to perform image inversion using VSIPL. The program also demonstrates how to move data between the VSIPL data space and the user data space, and how to access elements in a VSIPL vector view.

```
#include <iostream>
#include <stdio.h>
#include <vsip.h>

using namespace std;

void user_print_i(vsip_scalar_i data[], int count)
{
  for(int i = 0; i < count; ++i)
    printf("%2x ", data[i]);
}

void vu_vprint_i(vsip_vview_i *p_view)
{
  for(vsip_length i = 0; i < vsip_vgetlength_i(p_view); ++i)
    printf("%2x ", vsip_vget_i(p_view, i));
}

int main(int argc, char *argv[])
{
  // Number of elements
  const int COUNT = 16;

  // Initialise VSIPL
```

```
vsip_init(static_cast<void *>(0));

// User memory
vsip_scalar_i image[COUNT];
vsip_scalar_i dest[COUNT] = {};

// Bind to VSIPL
vsip_block_i *p_image_blk =
  vsip_blockbind_i(image, COUNT, VSIP_MEM_NONE);
vsip_block_i *p_dest_blk =
  vsip_blockbind_i(dest, COUNT, VSIP_MEM_NONE);

// Create views
// p_image_view: A view built from p_image_blk
//   p_255_view: A view created to store 255s
//  p_dest_view: A view built from p_dest_blk
vsip_vview_i *p_image_view =
  vsip_vbind_i(p_image_blk, 0, 1, COUNT);
vsip_vview_i *p_255_view =
  vsip_vcreate_i(COUNT, VSIP_MEM_NONE);
vsip_vview_i *p_dest_view =
  vsip_vbind_i(p_dest_blk, 0, 1, COUNT);

// Put 1 to COUNT into image
// Note that the user_memory variable image is directly edited.
// This is allowed because the block associated with memory is
// not yet under VSIPL control
for(int i = 0; i < COUNT; ++i)
  image[i] = i + 1;

// Fill p_255_view with all 255s
vsip_vfill_i(255, p_255_view);

// Output Before
cout << "< Original Image" << endl;
cout << "<   255: "; vu_vprint_i(p_255_view);  cout << endl;
cout << "< Image: "; user_print_i(image, COUNT); cout << endl;
cout << "<  Dest: "; user_print_i(dest, COUNT);  cout << endl;
```

```
  // Move p_image_blk and p_dest_blk to VSIPL control before
  // using them with VSIPL routines
  vsip_blockadmit_i(p_image_blk, VSIP_TRUE);
  vsip_blockadmit_i(p_dest_blk, VSIP_TRUE);

  // Calculate p_dest_view as 255 - image[i]
  vsip_vsub_i(p_255_view, p_image_view, p_dest_view);

  // Return p_image_blk and p_dest_blk to user control before
  // using normal functions to output their values
  vsip_blockrelease_i(p_image_blk, VSIP_TRUE);
  vsip_blockrelease_i(p_dest_blk, VSIP_TRUE);

  // Output After
  cout << "> Inverted Image" << endl;
  cout << ">   255: "; vu_vprint_i(p_255_view);  cout << endl;
  cout << "> Image: "; user_print_i(image, COUNT); cout << endl;
  cout << ">  Dest: "; user_print_i(dest, COUNT);  cout << endl;

  // Destroy p_255_view
  // Rest of views do not need to be destroyed with VSIPL
  // because they were not created by VSIPL
  vsip_valldestroy_i(p_255_view);

  // Close VSIPL
  vsip_finalize(static_cast<void *>(0));

  return 0;
}
```

The program outputs:

```
< Original Image
<   255: ff ff ff ff ff ff ff ff ff ff ff ff ff ff ff ff
< Image:  1  2  3  4  5  6  7  8  9  a  b  c  d  e  f 10
<  Dest:  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0
> Inverted Image
>   255: ff ff ff ff ff ff ff ff ff ff ff ff ff ff ff ff
> Image:  1  2  3  4  5  6  7  8  9  a  b  c  d  e  f 10
>  Dest: fe fd fc fb fa f9 f8 f7 f6 f5 f4 f3 f2 f1 f0 ef
```

The first four lines refer to values of `255`, `image` and `dest` arrays before the inversion, while the last four correspond to their respective values after the performing the inversion `dest` $=$ `255` $-$ `image`.

2.3 Vision with Generic Algorithms

Vision with Generic Algorithms (VIGRA) (Köthe 2001, 1999, 2000*c*, 1998, 2000*a*,*b*) is a computer-vision library created by Ullrich Köthe, as part of his Ph.D. thesis "Generische Programmierung für die Bildverarbeitung", focused primarily on flexible algorithms and generic programming. Built using template techniques similar to those used in the Standard Template Library (STL), VIGRA allows users to easily adapt VIGRA components to their needs. This flexibility comes almost for free, since the design uses compile-time polymorphism (templates).

2.3.1 Programming VIGRA

VIGRA brings the world of generic programming to image processing. As dictated by generic programming, data are separated from algorithms. Iterators allow access to the data. A variety of iterators are available, some of which provide read-only access. In general every iterator in VIGRA is two-dimensional, capable of walking of two separate directions — X and Y. Accessors (Köthe 1998, Kühl & Weihe 1997) provide another level of indirection. Data are set and read using accessor objects, which can change data representations between the iterator and user as needed. The Standard Template Library (STL) on the other hand does not use accessors. Accessors allow VIGRA to support certain data structures, like multi-band RGB images, easily (Köthe 1998).

Several general algorithms are provided by VIGRA, such as `vigra::transformImage` and `vigra::inspectImage`. These algorithms require functors to be of any use. The algorithms accept a functor, which they use to analyse, or change the data provided through iterators and accessors.

These algorithms however require a large number of arguments. To reduce the number of argument required to make each call, VIGRA uses functions like `vigra::srcImageRange` and `vigra::destImage` to construct tuples of the required arguments. These tuples usually take an image as a parameter, and provide default iterators and accessors to the algorithm functions.

2.3.2 Evaluation

1. Performance (●●●●○)

VIGRA was built with performance in mind, though one of its other aims is extensibility through generic programming. According to (Köthe 2000*c*), VIGRA's performance depends on the compiler used. VIGRA aims to be as fast as a hand-tuned program, and from the results presented in (Köthe 2000*c*), it comes within 5% of that aim. While VIGRA has good scalar performance, it does not use the VPU that is available on many desktop computers.

2. Ease of use (●●●●○)

   VIGRA is quite easy to use by anyone who understands the principles of generic programming and has experience with generic programming libraries like STL. Those without any experience of generic libraries might have a bit of culture shock as they try to understand how it works.

   In an effort to be easier to use, VIGRA make extensive use of pairs and tuples to reduce the number of arguments to function calls.

3. Adaptability (●●●●●)

   Since VIGRA is built on the principles of generic programming, it is highly adaptable. It is easy to use VIGRA routines on data structures created by the user.

4. Functionality (●●●●○)

   VIGRA supports many image-processing and image-analysis routines. While it lacks routines for capturing images from cameras, it supports loading and writing of popular image formats, such as TIFF and JPEG, via image libraries like ImageMagick.

5. Language Support (●●○○○)

   Being written in C++, and requiring templates, VIGRA is only usable fully in C++ (and Objective-C++, assuming the additional keywords of Objective-C++ have not been used as variable names). Other languages can however call C functions which would call VIGRA functions indirectly. This means all work must be done in C++.

6. Platform Neutrality (●●●●●)

   VIGRA compiles on many desktop computers, including Oberon, MacOS X, Windows and Linux. While it is not supported by any hardware manufacturer cards, it supports all the major desktop computer operating systems, and should be easily portable to other less popular systems like BeOS and Atheos. Therefore in the context of desktop machine vision, VIGRA can be regarded as being just as platform neutral as VSIPL.

2.3.3 Image inversion example

The following example shows the same inversion operation as the example in VSIPL.

```
#include <iostream>
#include <sys/time.h>
#include "vigra/stdimage.hxx"
#include "vigra/stdimagefunctions.hxx"
#include "vigra/impex.hxx"

using namespace std;
using namespace vigra;

void outputFunctor(BImage::PixelType pixel)
{
  printf("%2x ", pixel);
}

class linearInit
{
public:
  linearInit(int c) { _count = c; }
  BImage::PixelType operator()(BImage::PixelType pixel)
  { return _count++; }
private:
  int _count;
};

int main(int argc, char *argv[])
{
  const int COUNT = 16;

  try
  {
    // Create images of size width COUNT, and height 1
    BImage image(COUNT, 1);
    BImage dest(COUNT, 1);

    // Initialise image with 1 to 16
    transformImage(srcImageRange(image), destImage(image),
```

```
            linearInit(1));

    // Output Before Inversion
    cout << "< Original Image" << endl;
    cout << "< Image: ";
    inspectImage(srcImageRange(image), outputFunctor);
    cout << endl;

    cout << "<  Dest: ";
    inspectImage(srcImageRange(dest), outputFunctor);
    cout << endl;

    // Invert image using transformImage
    transformImage(srcImageRange(image), destImage(dest),
            linearIntensityTransform(-1, -255));

    // Output After Inversion
    cout << "> Inverted Image" << endl;
    cout << "> Image: ";
    inspectImage(srcImageRange(image), outputFunctor);
    cout << endl;

    cout << ">  Dest: ";
    inspectImage(srcImageRange(dest), outputFunctor);
    cout << endl;

  }
  catch(StdException &e)
  {
    cout << e.what() << endl;
    return 1;
  }

  return 0;
}
```

And the output produced by this program is:

```
< Original Image
< Image:  1  2  3  4  5  6  7  8  9  a  b  c  d  e  f 10
```

```
< Dest:  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0
> Inverted Image
> Image:  1  2  3  4  5  6  7  8  9  a  b  c  d  e  f 10
>  Dest: fe fd fc fb fa f9 f8 f7 f6 f5 f4 f3 f2 f1 f0 ef
```

2.4 Intel Performance Primitives for the Intel Architecture

The Intel Performance Primitives (IPP) for Intel Architecture (Int 2000-2001) library consists of routines for signal processing, image processing, and small matrix operations. IPP's functions take advantage of the vector-processor technologies MMX and Streaming SIMD extensions to provide performance benefits. Like VSIPL, it is a C library, with requirements on how data are provided. In addition, like VSIPL, it adds suffixes to the function name to indicate the types of the arguments. This feature is undesirable for template programming, because it makes it difficult to write one templated version. Qureshi (2004) discusses how they implemented templated programs using the IPP library.

IPP functions operate on arrays of primitive types (`short`, `float` and so on). They are one-dimensional for signal processing and two-dimensional for image processing. IPP provides many functions, including image arithmetic, colour conversions, thresholding, morphological operations, filtering and statistical information.

2.4.1 Evaluation

1. Performance (●●●●●)

   IPP uses the vector-processor technologies available on the Intel platforms. According to (*Performance Benchmarks for Intel® Integrated Performance Primitives* 2003), IPP's use of the VPU provides about one and a half to four-fold speedup for most domains, except audio where it manages to attain about an eight-fold speedup.

2. Ease of use (●●○○○)

   IPP is not easy to use because of its low-level C interface. In addition, many of its functions expect the user to provide information such as the size of steps.

3. Adaptability (●○○○○)

   While the user always has to provide images as arrays, since the user specifies the number of bytes per pixel, the user has a little bit of control over how to arrange the data.

4. Functionality (●●●●●)

IPP provides functions for many application domains, including audio, video, image, signal, speech, computer vision, matrix and cryptography (*Intel® Integrated Performance Primitives Product Features* 2004).

5. Language Support (●●●●●)

   As a general rule, it is usually possible to call C routine from other languages. Since IPP is a C library, it should be possible to call its functions from other languages with the right bindings.

6. Platform Neutrality (●●●○○)

   IPP is available on computers based on the Intel architecture, running Microsoft Windows and Linux.

2.5 IMAQ Vision

National Instruments' IMAQ Vision (Nat 1999) is an image-processing library for use with the LabVIEW (Nat 2005). Since it is for use with LabVIEW, programming IMAQ is done visually, and looks like data-flow diagrams. IMAQ provides blocks for processing and analysing grey-scale, colour and binary images, pattern matching, file input/output and capture using the FireWire camera among others. In additional, National Instruments provides Vision Builder that provides an interactive prototyping environment that is able to generate LabVIEW diagrams or builder files, which are detailed step-by-step descriptions of the operations including their parameters, for LabWindows/CVI, an ANSI C library, and Visual Basic (Nat 2004).

According to (Nat 1999), IMAQ provides three types of images — grey-level, colour and complex images. These images can be made up of a number of different types. Gray-level images, for example, can be 8-bit unsigned integer, 16-bit signed integer or 32-bit floating point. Colour images are processed by breaking the image into its components and then processing it like any grey-level image.

IMAQ has a large number of functions. In addition to image loading and storing and sequence grabbing, IMAQ also has lookup transformations, arithmetic and logic operators, spatial filters, frequency filters, morphological analysis and quantitative analysis functions.

2.5.1 Evaluation

1. Performance (●●●●●)

   IMAQ Vision takes advantage of the Intel MMX technology where available. According to (Nat 2002*b*), many IMAQ Vision functions show a four-fold increase in performance, when MMX is available.

2. Ease of use (●●●●●)

   LabVIEW is generally acknowledged to be an easy to use language, because of its visual nature.

3. Adaptability (○○○○○)

   Users cannot use their own types for use with the IMAQ library. Users however can call their own routines in dynamically linked libraries.

4. Functionality (●●●●●)

   IMAQ Vision has a wide range of functions for image processing. It provides blocks for performing image-processing functions like equalise, arithmetic operators, logic operator, spatial filters, frequency filters, morphological analysis and quantitative analysis. In addition, it is able to read and write from and to files, and capture images from FireWire cameras.

   Coupled with LabVIEW's graphical abilities, it is very easy to also show this output in a graphical user interface.

5. Language Support (●○○○○)

   IMAQ Vision is usable only in LabVIEW.

6. Platform Neutrality (●●●●●)

   LabVIEW is available on Windows, the Macintosh, Sun Solaris and Linux. However, IMAQ Vision, which is part of the Vision Development Module, is available only for LabVIEW 7.0 for Windows currently (Nat 2002*a*). Older versions were available for Mac OS 9.

2.6 Summary

Four existing image processing libraries, VSIPL, VIGRA, IPP and IMAQ, were presented in this chapter. While all the libraries presented can use the VPU, except VIGRA, only VIGRA is based on generic programming paradigms. The most influential library on this thesis was VIGRA, because like the main outcome of this thesis, it is a generic library. It is just missing VPU support.

Chapter 3

Vector Processor Programs

High-performance programs are required for machine-vision applications. Faster programs means more analysis and/or decision making can be done. While the desktop computer's processing power increases by leaps and bounds every year, poorly written programs would not see the same speedup. This is because today's processors use many different and complex techniques to achieve high speeds. Programs written with little regard to these techniques typically waste much of the processor's time. While these inefficiencies affect both scalar and vector programs, vector programs are more susceptible because of their higher memory bandwidth requirements; memory access is typically far slower that the processor. This chapter discusses how to write high-performance VPU programs, using AltiVec as an example.

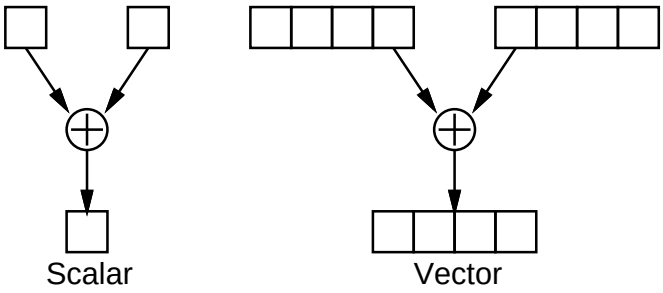
This chapter starts with an introduction to the VPU. After this introduction, this chapter focuses more on AltiVec processors. While the general concepts discussed are applicable to other vector-processor technologies, like MMX and 3DNow!, implementation details cover only AltiVec. After describing what a VPU is, how the VPU might be detected is covered. This is followed by a discussion on focusing on throughput instead of low latency to achieve higher speeds, memory and cache issues. After this, some vector algorithms, a summary of current methods of programming for the VPU, and finally an investigation into the effect of different issues on AltiVec programs are presented.

3.1 The vector processor

Scalar processors compute scalars one at a time. A VPU is simply a processor that computes a number of scalars simultaneously (see Figure 3.1). This is different from multithreading where the processor still computes one scalar at a time, but runs two or more separate instruction streams, switching between them constantly.

A desktop VPU usually handles a fixed number of elements at a time. This fixed number of elements together form a vector. Usually, the number of elements in a vector

Figure 3.1 A scalar processor adds a pair of numbers together in the same time a vector processor adds n pairs of numbers.



differs across types while the overall size of the vector remains constant. A VPU works with these vectors in much the same way as the scalar processor. For the purposes of this thesis, when the term VPU is used, it refers only to a VPU in the desktop computer unless otherwise stated. It is important to keep this in mind because the VPUs that some super-computers have can be quite different.

VPUs normally augment scalar processors, because not all algorithms are suitable for vectorisation. VPUs are suitable for applications where there is a large amount of data that the same series of instructions are to be applied to — Single Instruction Multiple Data (SIMD) problems. Examples of applications which should make good use of the VPU include matrix multiplication, video, image and sound processing. In fact, on the desktop, VPUs were built primarily for multimedia applications. Examples of vector-processor technologies include MMX, SSE, and AltiVec.

Desktop VPUs have a number of common restrictions that affect how vectorised programs function. In general, all VPUs can process only a fixed number of elements per vector. In addition, they usually have restrictions on where they can fetch and store data quickly. Data stored in these locations are referred to as being aligned. Unaligned data typically result in large delays, because of the extra processing required. In fact, one of the main bottlenecks in vector programs is memory. A VPU requires a much wider memory bandwidth than an equivalent scalar processor, because of its larger vector sizes and other popular processor techniques such as pipelining. The final characteristic that may influence the way programs are written is that vectors of different types are all the same size and thus they contain different numbers of elements.

This discussion on vectorisability uses AltiVec as the VPU. This is because AltiVec is the only vector-processing technology currently available on desktops, to have a separate vector-processing unit. It does not follow technologies like MMX and 3DNow! which use the floating-point unit as the vector-processing unit. Having a distinct vector-processing unit gives AltiVec better vector-processing performance, and more vector functions. For more information about AltiVec programming, please consult Mot (1999), AltiVec.org (2002), App (2002*c*), Ollmann (2001), and Lai & McKerrow (2001).

Algorithm 3.1 Detecting AltiVec using Apple's Gestalt Manager from (App 2002*b*)

```
#include <Gestalt.h>
Boolean IsAltiVecAvailable(void)
{
  long cpuAttributes;
  Boolean hasAltiVec = false;
  OSerr = Gestalt(gestaltPowerPCProcessorFeatures,
                  &cpuAttributes);
  if(noErr == err)
    hasAltiVec = (1 << gestaltPowerPCHasVectorInstructions)
      & cpuAttributes;
  return hasAltiVec;
}
```

Algorithm 3.2 Detecting AltiVec without Carbon from (App 2002*b*)

```
#include <sys/sysctl.h>
Boolean IsAltiVecAvailable(void)
{
  int selectors[2] = {CTL_HW, HW_VECTORUNIT};
  int hasVectorUnit = 0;
  size_t length = sizeof(hasVectorUnit);
  int error = sysctl(selectors, 2, &hasVectorUnit, &length,
                     NULL, 0);
  if(0 == error)
    return hasVectorUnit != 0;
  return FALSE;
}
```

3.2 Detecting the vector processor

In C/C++, compilers defines certain preprocessor macros in the presence of different processor features. GCC, for example, defines `__ALTIVEC__` when AltiVec is available. This can be a simple and effective technique for detecting the presence of the AltiVec processor. However, this detection method works only at compile time. Determining the processor type at compile time means that the program does not have to waste time during execution trying to decide whether AltiVec is available. However, sometimes programmers wish to release a single binary to multiple platforms. For AltiVec, Apple provides run-time AltiVec detection for Carbon applications using the Gestalt Manager, or via a `sysctl` call for other programs. Algorithms 3.1 and 3.2 show how such run-time detection can be done using Carbon and without Carbon respectively.

According to App (2002*b*), with AltiVec, it is not enough to wrap AltiVec specific code in `if{}` blocks, because AltiVec code generation usually adds a `mfspr vrsave` instruction in the preamble of any function which uses vector types. Since the `vrsave`

Algorithm 3.3 `DotProduct()` function from (App 2002*i*)

```
float DotProduct(float a[3], float b[3])
{
  return a[0] * b[0] + a[1] * b[1] + a[2] * b[2];
}
```

Algorithm 3.4 `DotProduct()` in Assembly from (App 2002*i*)

```
fmuls  temp, a0, b0           // temp = a0 * b0
fmadds temp, a1, b1, temp     // temp = a1 * b1 + temp
fmadds result, a2, b2, temp   // result = a2 * b2 + temp
```

register does not exist on non-AltiVec processors, this can lead to problems. Avoiding the preamble is not recommended because it is required for proper operation on the MacOS. Therefore the only safe technique for writing a program which determines at run-time whether to use vector instructions or not is to put AltiVec specific code in its own function.

3.3 High-throughput computing

Functions are used by many programmers to divide and conquer problems. After conquering the problem, programmers then identify the most expensive functions, and try to make them faster. Since functions are about dividing problems, each function typically handles only a small amount of data. This approach to optimisation where the programmer endeavours to make individual functions faster is known as the low-latency approach — the aim is to reduce function latency.

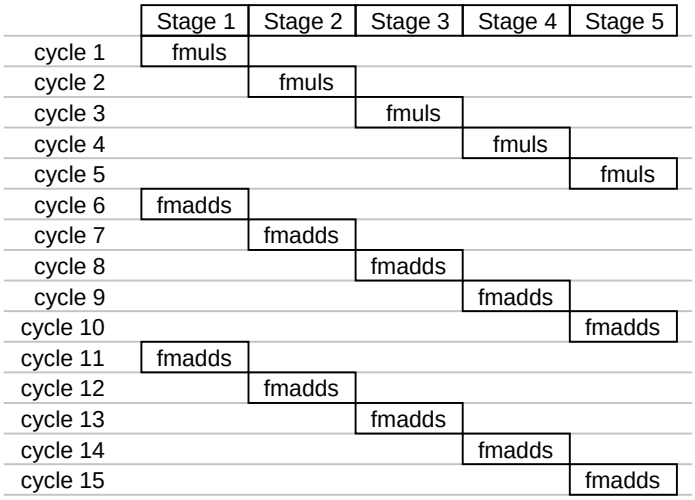
Another technique of optimisation is known as high-throughput. With high-throughput, the emphasis is on crunching as much data as possible in the shortest feasible time. High-throughput functions are focused on making the best use of the available hardware. However, in general, high-throughput optimisation produces code that is more difficult to write and maintain.

High-throughput optimisation usually produces faster programs than low-latency optimisation because they tend to result in programs that fill the processor's pipelines more efficiently. Taking an example from App (2002*i*), which assumes a perfect hypothetical pipeline, the scalar dot-product in Algorithm 3.3, becomes Algorithm 3.4 in assembly.

As illustrated in Figure 3.2, because each instruction depends on the data before it, the instructions will have to wait for each other to complete before executing. On a PowerPC 7450 and 7455, each instruction in Algorithm 3.4 takes 5 cycles each. This results in 15 cycles for a single dot-product.

Changing the dot-product to multiply five at a time allows the pipelines to fill more

Figure 3.2 Hypothetical PowerPC 7450/7455 five-stage pipeline executing `DotProduct()` (Algorithm 3.3 and 3.4) from (App 2002*i*)



Algorithm 3.5 `DotFive()` from (App 2002*i*)

```
void DotFive(float a[5][3], float b[5][3], float result[5][3])
{
  result[0] = a[0][0]*b[0][0]+a[0][1]*b[0][1]+a[0][2]*b[0][2];
  result[1] = a[1][0]*b[1][0]+a[1][1]*b[1][1]+a[1][2]*b[1][2];
  result[2] = a[2][0]*b[2][0]+a[2][1]*b[2][1]+a[2][2]*b[2][2];
  result[3] = a[3][0]*b[3][0]+a[3][1]*b[3][1]+a[3][2]*b[3][2];
  result[4] = a[4][0]*b[4][0]+a[4][1]*b[4][1]+a[4][2]*b[4][2];
}
```

completely, because of the reduction in the number of data-dependency stalls. Algorithm 3.5 should execute in the manner depicted by Figure 3.3.

By filling the pipelines completely, five dot-products were computed in 19 cycles as opposed to 15 cycles for one dot-product. In actuality, the `DotFive()` in Algorithm 3.5 does not achieve anywhere close its potential speedup, because it requires more loads and stores. It does however manage to be faster than five `DotProduct()`s. The point is, processing a large number of elements at a time can reduce data-dependency stalls. This allows the processor to make fuller use of its pipeline, increasing the performance of the program.

3.4 Memory and cache management

VPUs typically have higher data-processing capacities than equivalent scalar processors. Because both vector and scalar processors use the same bus technology, vector programs have to be much more careful with how they access and use their memory. A simple example from Apple (App 2004*b*) illustrates just how disparate the data capabilities of

Figure 3.3 Hypothetical PowerPC 7450/7455 five-stage pipeline executing `DotFive()` (Algorithm 3.5) from (App 2002*i*)

	Stage 1	Stage 2	Stage 3	Stage 4	Stage 5
cycle 1	fmuls				
cycle 2	fmuls	fmuls			
cycle 3	fmuls	fmuls	fmuls		
cycle 4	fmuls	fmuls	fmuls	fmuls	
cycle 5	fmuls	fmuls	fmuls	fmuls	fmuls
cycle 6	fmadds	fmuls	fmuls	fmuls	fmuls
cycle 7	fmadds	fmadds	fmuls	fmuls	fmuls
cycle 8	fmadds	fmadds	fmadds	fmuls	fmuls
cycle 9	fmadds	fmadds	fmadds	fmadds	fmuls
cycle 10	fmadds	fmadds	fmadds	fmadds	fmadds
cycle 11	fmadds	fmadds	fmadds	fmadds	fmadds
cycle 12	fmadds	fmadds	fmadds	fmadds	fmadds
cycle 13	fmadds	fmadds	fmadds	fmadds	fmadds
cycle 14	fmadds	fmadds	fmadds	fmadds	fmadds
cycle 15	fmadds	fmadds	fmadds	fmadds	fmadds
cycle 16		fmadds	fmadds	fmadds	fmadds
cycle 17			fmadds	fmadds	fmadds
cycle 18				fmadds	fmadds
cycle 19					fmadds

the memory and the VPU are. A simple instruction like `vec_add` could spend up to 93% of its time waiting for data to appear, for a 400 MHz PowerPC G4 processor coupled with a 100 MHz bus. More modern systems use higher processor and bus speeds. A 933 MHz PowerPC G4 processor could consume 30 gigabytes of data per second (32$\times$ 933 MHz), while a 133 MHz bus will deliver only 1 gigabyte of data per second (8$\times$133 MHz) . Combining the two together means the `vec_add` instruction should spend around 96% of its time waiting. A 2.0 GHz PowerPC G5 would spend about 95% of its time waiting for data to appear (App 2004*b*). Operations requiring both reading and writing of vectors could increase the data consumption by as much as three times. However, even if all the data are in the L1 data cache, the full throughput is still not possible. A top throughput of 6.4 gigabytes of data per second is more realistic for a 400 MHz PowerPC G4 (App 2004*b*).

This section discusses how to use memory more efficiently. Using memory more efficiently will increase the speed of memory-bound functions, which are functions that are limited by the speed of memory. A memory-bound program that reduces its memory usage would increase its speed. Another technique is simply to do more with the data once you have it. The final technique is prefetching — moving data from memory to the cache before you even use it, so that when you do want to use it, it is already in the cache.

3.4.1 Doing more with data

Apart from reducing the memory usage of a program, the program can elect to do more with the data. Once the data are in the processor, the program should endeavour to perform

as much computation to them as possible. It is therefore better to merge loops that handle the same data together, so that they are not loading the data into the processor twice.

3.4.2 Reducing memory usage

There are many different ways of reducing the memory usage of a program. A simple method is to pick an algorithm that uses less memory. Other suggestions for reducing memory usage from Apple (App 2004*b*) are described below.

Use AltiVec friendly storage formats: Since loading from memory is slow, programs requiring high performance should try to minimise the effect of loading memory. This can be done by avoiding scattered loads, and aligning data correctly. In addition, arranging the data as uniform vectors makes it easier for a vector program to process the data.

Reduce memory consuming "optimisations": Many classic programming techniques were developed at a time when the CPU was slow and memory was fast (by comparison). Such techniques therefore try to use more memory and less CPU to attain their goals, because it would be faster on such systems. Today, the situation is reversed. The CPU is far faster than memory. Therefore avoiding these memory consuming "optimisations" can be beneficial to the program. A popular example of a memory consuming "optimisation" is the lookup table. Lookup tables can cause a scalar code bottleneck in a vector program, unless the lookup size is really small. App (2004*b*) suggests up to 16 elements, while Ollmann (2001) believes 64 elements can be looked up up with ease. According to Scales (2000), 256-elements parallel lookups are also possible.

Code also uses memory: Code also needs to be loaded into memory and the cache before it can be executed. The L1 cache in PowerPC processors has a portion dedicated to instructions only. For example, the G4 has a 32 KB L1 data cache and a 32 KB L1 instruction cache (App 2004*a*). The G5 has a larger 64 KB L1 instruction cache and 32 KB L1 2-way data cache (App 2004*a*). Code that is larger that the L1 instruction cache will start spilling into the L2 cache. This means that there is less cache left for data.

Globals and constants: Globals should be avoided because a compiler cannot usually determine if a global value will change or not (because of the possibility of other threads changing the global). This means that the code generated would fetch globals from memory every time they are required, instead of loading from memory for the first time only, and subsequently from the cache. Constants can be generated directly from instructions rather than being loaded from main memory. (Bettag 2001)

contains information on the instructions required to generate 16-bit integer constants for AltiVec. Constants that can not be generated, either automatically by the compiler or by the programmer, require loading from memory. In some cases, the use of constants can be avoided by using different instructions. Constant generation is discussed later in Section 3.5.1.

3.4.3 Prefetching

According to App (2004*a*), when loading data, the G4 processor checks the L1 data cache for the data first. If the data are not in the L1, it checks the L2 (and then L3, if there is an L3 cache). Failing that as well, the processor will look for the data in memory. After finding the data, the data have to be moved into the L1. The movement of data will cause some data in the L1 cache to be displaced. The displaced data go into the L2 cache by default, before memory.

From this scenario, two important optimisation rules can be deduced. It is evident that for shortest load times, data should always be in the cache by the time the processor wants to use it. However, in order to be in the cache by the time the processor wants it, the data will have to be pre-loaded into the cache. The second rule is that because the default behaviour moves data from L1 to L2 (to L3) before being flushed to memory, it can displace other valid information in the L2 (and L3). In some cases, it is better to displace data directly from the L1 to main memory. Doing so will help the rest of the program run faster. This process of moving data into the caches before it is needed is called prefetching.

AltiVec provides data stream instructions to allow the programmer to handle these two issues. They allow the programmer to schedule data for loading into the cache before the data are actually needed. The G4 processor is able to do this concurrently, without affecting the execution of code. In addition, it allows the programmer to specify how to flush the data after use. For data that is used once only, or once in a while, it is better to displace directly to main memory.

Using the data stream instructions to load data into the caches concurrently might have no effect on the program speed or can cause the program to run slower if not done correctly. Some errors, highlighted by App (2004*a*), which can cause a program to have no noticeable improvement are listed as follows.

1. Data are not being prefetched early enough.

2. Data are prefetched too early. By the time the processor wants to use it, it has already been displaced.

3. Too much data are being prefetched, causing more bus traffic than necessary.

4. The prefetch has stopped prematurely. For example, another program may be trying to use the same data stream as you are.

5. The data are already in the cache.

6. Cache stalls are not a major bottleneck in the program.

Because of the low cost of starting a prefetch and the difficulty in keeping the program and the data stream synchronised, App (2004*a*) recommends the program start prefetching often. The only problem with this is that because less data are prefetched, there is a higher chance that the processor will overtake it. If the processor takes longer to compute the data than to load the data, then the loading of data can occur in parallel and it is easy to stay synchronised with the data stream.

Not all VPUs require prefetching, or handle prefetching in the same manner. According to App (2003*c*), the G5 starts prefetching automatically. In addition, the G5 interprets the AltiVec data stream instructions differently. It cannot execute the Data Stream Touch instructions speculatively, and therefore causes the execution engine to be drained completely before execution. This can result in large bubbles (clock cycles during which the processor is idle). In addition, transient hints are ignored for the G5.

3.5 Vector processor specific algorithms

This section contains discusses VPU specific algorithms. Topics covered are generating constants, performing unaligned loads and stores, edge handling and type conversion.

3.5.1 Constants

Considerations raised in Section 3.4 suggest it is usually faster to generate constants than to load them from memory. Bettag (2001) provides a list showing how to generate constants having scalars 0 to 65355 on the AltiVec. Such a list is useful for generating constant vectors whose scalars the programmer knows at program-time. Table 3.1 shows how to generate `__vector unsigned short` constants with scalars 0 to 19 without loading data from memory (Bettag 2001).

Sometimes the constant required is not known at program-time. In such cases, Algorithm 3.6, which was adapted from (App 2002*g*), is useful. This algorithm works by loading the scalar `s` into the vector `result`. The position of `s` in `result` actually depends on the address of `s`. Since `s` is passed in as a argument, we can assume that it is naturally aligned, which means that it is aligned as appropriate for its size. For example, for `char` types, the address would be a multiple of 1, for `shorts` the address would be a multiple of 2, for `ints` the address would be a multiple of 4 and so on. After loading `s` into `result`,

Table 3.1 Generating 0-19 AltiVec `unsigned short` constants using instructions (Bettag 2001)

	Instruction Count	Instructions
0	1	`vec_splat_u8(0)`
1	1	`vec_splat_u16(1)`
2	1	`vec_splat_u16(2)`
3	1	`vec_splat_u16(3)`
4	1	`vec_splat_u16(4)`
5	1	`vec_splat_u16(5)`
6	1	`vec_splat_u16(6)`
7	1	`vec_splat_u16(7)`
8	1	`vec_splat_u16(8)`
9	1	`vec_splat_u16(9)`
10	1	`vec_splat_u16(10)`
11	1	`vec_splat_u16(11)`
12	1	`vec_splat_u16(12)`
13	1	`vec_splat_u16(13)`
14	1	`vec_splat_u16(14)`
15	1	`vec_splat_u16(15)`
16	2	`vec_vaddubm(vec_splat_u16(8),vec_splat_u16(8))`
17	3	`vec_vandc(vec_splat_u16(-9),` `  vec_vadduhm(vec_splat_u16(-9),vec_splat_u16(-9)))`
18	2	`vec_vaddubm(vec_splat_u16(9),vec_splat_u16(9))`

19	3	`vec_vnor(`
		` vec_vadduhm(vec_splat_u16(-10),vec_splat_u16(-10)),`
		`  vec_vadduhm(vec_splat_u16(-10),vec_splat_u16(-10)))`

Algorithm 3.6 A function that generates arbitrary constant vectors, adapted from (App 2002*g*)

```
// Example Usage:
// typedef unsigned char uchar;
// typedef __vector unsigned char vector_uchar;
// vector_uchar v = splat<vector_uchar, uchar>(10);
template<typename V, typename T> V splat(T s) {
  // Load t into the vector.
  V result = vec_lde(0, &s);
  // Rotate the vector so that t is at position 0
  vector unsigned char rotate_mask = vec_lvsl(0, &s);
  result = vec_perm(result, result, rotate_mask);
  // Fill the result with the scalar at position 0 (which is s)
  return vec_splat(result, 0);
}
```

we then rotate `s` to a known position, which is 0 in this case. Then we use `vec_splat` to fill `result` with the scalar at 0. We had to perform the rotation instead of calculating the element to ask `vec_splat` to fill because `vec_splat` requires the position to be literal.

App (2002*g*) also discusses how to create arbitrary constant vectors for 16-byte aligned scalars and arbitrarily aligned scalars.

3.5.2 Unaligned loads

AltiVec VPUs are able to load memory directly only from aligned memory locations (Lai & McKerrow 2001, Mot 1999). While SSE and SSE2 are able to load unaligned memory locations, loading from aligned memory locations is faster (Adv 2002). This section discusses how to process unaligned memory locations with AltiVec.

App (2002*e*), Ollmann (2001), and Lai & McKerrow (2001) discuss how to load consecutive unaligned vectors efficiently. An unaligned vector can be loaded using something similar to Algorithm 3.7. To load an unaligned vector, two aligned vectors, starting from the first aligned address just before the unaligned address that is to be loaded, are loaded. The unaligned vector that we want to load would now be in these two aligned vectors. The `vec_lvsl` instruction generates a vector that specifies where the unaligned vector is located in the two aligned vectors, which when passed to the `vec_perm` instruction, extracts the unaligned vector from the two aligned vectors.

Using the function in Algorithm 3.7 for a contiguous array is inefficient because in subsequent calls, since `low` would be `high` from the previous call, it results in more loads than necessary. Algorithm 3.8 (App 2002*e*) shows how four consecutive unaligned vectors would be loaded. Using this technique, only one extra vector load is needed. When used in a loop, it is sensible to use `v0` as `vExtra`. This results in only one extra vector load

Algorithm 3.7 Loading an unaligned vector modified from (Ollmann 2001)

```
// Load a vector from an unaligned location in memory
__vector unsigned char LoadUnaligned(unsigned char *p_v) {
  __vector unsigned char permuteVector = vec_lvsl(0, p_v);
  __vector unsigned char low = vec_ld(0, p_v);
  __vector unsigned char high = vec_ld(16, 0);
  return vec_perm(low, high, permuteVector);
}
```

Algorithm 3.8 Efficient unaligned loads from (App 2002*e*)

```
vector float v0, v1, v2, v3, vExtra;
vector unsigned char fixAlignment;

// Load four unaligned vectors as five aligned vectors
v0 = vec_ld(0 * sizeof(vector float), ptr);
v1 = vec_ld(1 * sizeof(vector float), ptr);
v2 = vec_ld(2 * sizeof(vector float), ptr);
v3 = vec_ld(3 * sizeof(vector float), ptr);
vExtra = vec_ld(4 * sizeof(vector float), ptr);

// Use vec_perm to extract out the desired unaligned vectors
fixAlignment = vec_lvsl(0, ptr);
v0 = vec_perm(v0, v1, fixAlignment);
v1 = vec_perm(v1, v2, fixAlignment);
v2 = vec_perm(v2, v3, fixAlignment);
v3 = vec_perm(v3, vExtra, fixAlignment);
```

for the entire loop. Because the application will crash if the last `vExtra` includes memory that does not belong to it, to use this technique safely, there should be an extra vector at the end of the data.

For small arrays that require no looping, it is possible to load `vExtra` as the last element in the array. This allows the program to avoid reading memory that does not belong to the array and therefore prevent the possibility of access exceptions. This is presented in Algorithm 3.9, which was reproduced from App (2002*e*).

3.5.3 Unaligned stores

AltiVec VPUs are not only restricted to loading memory directly from aligned addresses, they can write memory directly only to aligned addresses (Lai & McKerrow 2001, Mot 1999). While SSE and SSE2 are able to write to unaligned memory addresses, writing to aligned memory addresses is faster (Adv 2002). This section discusses how to store vectors to unaligned memory locations with AltiVec.

Unaligned stores basically require writing to two vectors instead of one. The original

Algorithm 3.9 Efficient unaligned loads for small arrays requiring no loops from (App 2002*e*)

```
vector float v0, v1, v2, v3, vExtra;
vector unsigned char fixAlignment;

// Load four unaligned vectors as five aligned vectors
v0 = vec_ld(0 * sizeof(vector float), ptr);
v1 = vec_ld(1 * sizeof(vector float), ptr);
v2 = vec_ld(2 * sizeof(vector float), ptr);
v3 = vec_ld(3 * sizeof(vector float), ptr);
vExtra = vec_ld(4 * sizeof(vector float) - sizeof(float), ptr);

// Use vec_perm to extract out the desired unaligned vectors
fixAlignment = vec_lvsl(0, ptr);
v0 = vec_perm(v0, v1, fixAlignment);
v1 = vec_perm(v1, v2, fixAlignment);
v2 = vec_perm(v2, v3, fixAlignment);
v3 = vec_perm(v3, vExtra, fixAlignment);
```

vector needs to be combined with the original two vectors to generate two new vectors, which have the result vector grafted onto them at the correct places. These two new vectors are then written back. Because this technique requires writing to locations that are not used by the vector to be stored, it can lead to problems in threaded programs where another thread modifies the data in the vector just before the grafted vector is written. In addition, it is possible to try to read/write to locations that are not owned by the process, thereby leading to fatal crashes.

An algorithm describing how to perform unaligned stores is shown in Algorithm 3.10, which was adapted from Ollmann (2001) and published in Lai & McKerrow (2001). Note that the original used `vec_lvsl` instead of `vec_lvsr`. Since the algorithm seemed to fail when using the `vec_lvsl`, it was changed to `vec_lvsr`.

3.5.4 Handling edge conditions

When processing arrays with a VPU, there is a high chance that the data are unaligned, especially if the program is processing only a part of a larger array. In such cases, the number of elements in the data to be processed is not usually exactly divisible by the number of elements in the vector. The elements that fall out of alignment is what the author refers to as edge conditions. As Figure 3.4 illustrates, a contiguous array in memory has at most two edges.

App (2002*e*) suggests using the scalar processor to handle such edge conditions. This is because the scalar unit and vector unit in the G4 are able to operate independently of each other. Therefore, the edge conditions could be handled for free. It is also possible to

Algorithm 3.10 Unaligned store from (Lai & McKerrow 2001)

```
// Store a vector to an unaligned location in memory
void store(__vector unsigned char v, __vector unsigned char *p)
{
  // Load the surrounding areas
  __vector unsigned char low = vec_ld(0, p);
  __vector unsigned char high = vec_ld(16, p);
  // Prepare the constants that we need
  __vector unsigned char pv = vec_lvsr(0, (int *)p);
  __vector unsigned char oxFF = vec_splat_u8(-1);
  __vector unsigned char ox00 = vec_splat_u8(0);
  // Make a mask for which parts of
  // vector to swap out
  __vector unsigned char m = vec_perm(ox00, oxFF, pv);
  // Right rotate out input data
  v = vec_perm(v, v, pv);
  // Insert our data into the low and
  // high vectors
  low = vec_sel(low, v, m);
  high = vec_sel(v, high, m);
  // Store the two aligned result
  // vectors
  vec_st(low, 0, p);
  vec_st(high, 16, p);
}
```

Figure 3.4 A contiguous one-dimensional array can have at most two edges

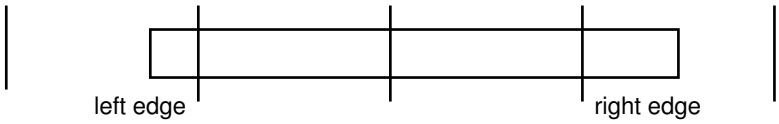
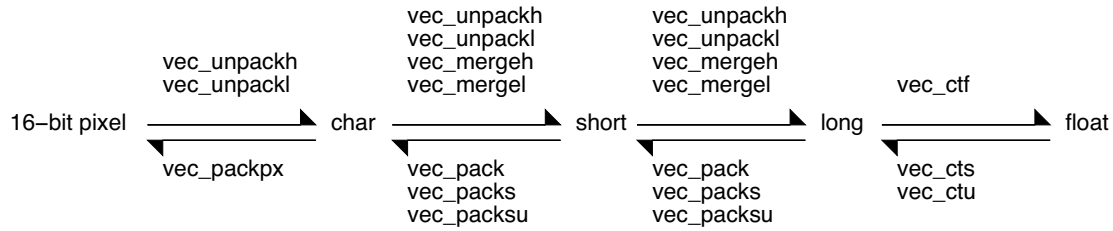


Figure 3.5 AltiVec type conversion functions from (App 2002*a*)



handle the edge conditions in the vector unit by copying the edge vector to a temporary work vector. The operation is then carried out on the temporary work vector, after which it is grafted back into the original array.

3.5.5 Type conversions

Vectors of different types in the VPU are usually all the same size. In AltiVec for example, all vectors are 128 bits long. However, the size of the elements in the different vectors is not constant. This means that the number of elements in vectors of different types is not consistent across types. AltiVec, for example, can have 16 characters, 8 short integers, 4 long integers or 4 floating-point scalars in a vector. When working with vectors of a single type, there is no problem. However, when trying to convert from one type to the other, this becomes an issue. Essentially a single vector conversion could create n vectors or vice versa. With respect to AltiVec, n is 1, 2, or 4.

AltiVec provides several functions to change the type of its vectors. Figure 3.5 from (App 2002*a*) illustrates the functions required for conversion.

3.6 Approaches to programming the vector processor

There are currently several methods of writing programs that utilise the VPU. These other methods, arranged in order of increasing conceptual level, include using the VPU directly, using vector-processor wrappers, vector libraries, high-level libraries, vectorising compilers and parallel languages.

Semantic vectorisation refers to being able to express an algorithm in a manner that uses vectors to produce the answer. Rizzoli et al. (1986) point out that "in order to fully exploit the capabilities of a vector hardware, any program architecture must be structured

accordingly". Semantic vectorisation allows programmers to structure their program in a manner that is easy for the VPU to execute.

3.6.1 Vector processor

VPUs can be programmed directly either through the use of assembly or a C interface, that maps one-to-one to existing vector processing instructions. Programming for the VPU directly provides the user with the most control. However, its low-level interface makes it more difficult for programmers to use.

Programming the VPU directly supports semantic vectorisation and results in highly efficient code that is not portable to other VPUs.

3.6.2 Vector-processor wrappers

Since using the VPU directly can be difficult, there are libraries around that wrap the VPU commands in a clearer, easier to use syntax. Vector-processor wrappers usually have a one-to-one correlation between instructions in the wrapper and instructions in the VPU and have the same constraints as the VPU that they are wrapping. An example of such wrappers can be found in part of Pixelglow's MacSTL library (Pix 2003), which wraps AltiVec commands in C++ style operators.

Programming for vector-processor wrappers is similar to programming for the VPU directly except the interface is usually simpler and more intuitive. Therefore using vector-processor wrappers also supports semantic vectorisation and results in code that is not portable to other VPUs. Since vector-processor wrappers are extremely thin wrappers, using vector-processor wrappers should produce highly efficient code.

3.6.3 Vector libraries

Vector libraries aim to provide functionality that allows vectors to be manipulated easily. Examples of such libraries are Blitz++ (Veldhuizen 2001), and C Vector Library (CVL) (Blelloch et al. 1995, Hardwick 1995, Blelloch et al. 1994). Vector libraries provide facilities to manipulate vectors in an easy-to-use package and support different vector-processor technologies through different implementations of the same interface — ports. Some of these libraries were developed with real VPUs in mind. For example, according to (Hardwick 1995), CVL was designed so that it can be efficiently implemented on a wide variety of parallel machines.

Vector libraries support cross-platform development and support semantic vectorisation. However, not all vector libraries are necessarily easy to vectorise, and so may result in less than optimal code.

3.6.4 High-level libraries

High-level libraries refer to libraries that offer facilities that conceptually have little to do specifically with a VPU. Examples of such libraries include image processing and data structures. While a lot of these libraries do not use the VPU at all, some use the VPU behind the scenes to provide speed benefits transparently. Different ports of the library could support different VPUs. VSIPL (*VSIPL website* 2001, Geo 2001) and VIGRA (Köthe 2001) are both image-processing libraries. However VIGRA does not use the VPU at all. VSIPL on the other hand has an AltiVec port, MMX port and many others. In fact, according to Jaenicke (1999), VSIPL was designed with AltiVec in mind.

While different implementations of the same library that target different vector-processor technologies can be created, creating these different ports in this manner means more work, both development-wise and maintenance-wise. This is because there are now many different versions of the library.

3.6.5 Vectorising compilers

Vectorising compilers automatically detect code that is suitable for parallel execution and generate a vectorised implementation. Languages that have constructs to express parallel operation are easier to vectorise automatically.

Most popular programming languages in use today however do not have such constructs. Despite this, there are a number of different vectorising compilers for these languages. Examples include the Intel C++ compiler (Walls & Fegreus 2002, *Optimizing Applications with Intel C++ and Fortran Compilers for Windows and Linux* 2003) and VAST/AltiVec (Cre 2003), which are vectorising compilers for MMX and AltiVec technologies respectively. However, to help the compiler, the programmer has to be careful of how the program is constructed. Bik et al. (2003) discuss how to structure programs so that they have a higher chance of being vectorised automatically.

3.6.6 Parallel languages

Parallel languages, like Nested Data-Parallel Language (NESL) (Blelloch et al. 1994), and extensions to C, such as Dataparallel C (Hatcher et al. 1991), CxC (Oberdorfer & Gutowski 2004), Multimedia C (MMC) (Bulic & Gustin 2003), among others are all attempts at allowing parallel operations to be specified explicitly by the programmer so that compilers can produce better vectorised programs automatically.

Some parallel languages, such as CxC, are generally aimed at SPMD (Single Program Multiple Data) problems (Oberdorfer & Gutowski 2004), while others, such as MMC, are aimed at providing language support for VPUs (Bulic & Gustin 2003). Parallel languages typically have the user programming virtual processor(s).

3.7 Effect of different issues on speedup

This section investigates the effect of alignment, prefetching and function complexity on the speedup of AltiVec programs. This investigation was published in Lai et al. (2002).

Lai et al. (2002) presented several different AltiVec algorithms. The fastest version did not handle any edges though. The fastest version presented that could handle edges is reproduced in Algorithm 3.7. This algorithm aligns the input image to the output image, thereby reducing the cost of unaligned loading. Because of this alignment, there is only a right edge. This right edge was computed using the scalar processor. For some operations, the edges can actually be computed with the VPU.

Algorithm 3.11: AltiVec transform functor

```
// T is the scalar type. eg. float
// V is the corresponding AltiVec vector type. eg. __vector float
// F is a functor for V
// SF is a functor for T
template<class T, class V, class F, class SF>
void transform(T* start, T* end, T* out, F f, SF sf)
{
  const int step = sizeof(V)/sizeof(T);
  V ov, iv, iv_xtr;
  __vector unsigned char ifix;
  int count = end - start;
  T *pi = start, *po = out;
#ifdef DST
  vec_dst(pi, 0x10010100, 0);
#endif
#ifdef UNALIGNED_LOAD
  // Do Initial load
  iv_xtr = vec_ld(0, pi);
  pi += step;
  ifix = vec_lvsl(0, pi);
#endif
  for(int i = 0; i < count; i += step)
  {
#ifdef DST
    vec_dst(pi, 0x10010100, 0);
#endif
#ifdef UNALIGNED_LOAD
    // Load unaligned
```

```
    iv = iv_xtr;
    iv_xtr = vec_ld(0, pi);
    pi += step;
    iv = vec_perm(iv, iv_xtr, ifix);
#else
    iv = vec_ld(0, pi);
    pi += step;
#endif
    ov = f(iv);
#ifdef UNALIGNED_STORE
    // store was discussed in Algorithm 3.10
    store((__vector unsigned char)ov, (__vector unsigned char*)po);
    po += step;
#else
    vec_st(ov, 0, po);
    po += step;
#endif
  }
#ifdef UNALIGNED_STORE
  // Handle right edge with scalar processor
  int starti = (count / step) *  step;
  for(int i = starti; i < count; ++i)
    out[i] = sf(pi[i]);
#endif
}
```

Algorithm 3.12: Optimised AltiVec transform function

```
// This vector loop will load unaligned data to the alignment of
// the output, so that only unaligned load is needed
// T is the scalar type. eg. float
// V is the corresponding AltiVec vector type. eg. __vector float
// F is a functor for V
// SF is a functor for T
template<class T, class V, class F, class SF>
void transform(T* start, T* end, T* out, F f, SF sf) {
  const int step = sizeof(V)/sizeof(T);
  V ov, iv, iv_xtr;
  __vector unsigned char ifix;
```

```
  int count = end - start;
  T *pi = start, *po = out;
#ifdef DST
  vec_dst(pi, 0x10010100, 0);
#endif
  // Load location = input offset +
  //                 (element count - output offset) % step
  // % step required because want values from 0 to
  // (scalar count - 1)
  int ioffset = ((unsigned long)pi & 0xf);
  int ooffset = ((unsigned long)po & 0xf);
  int roffset = ((step - ooffset) % step);
  if(roffset + ioffset > step)
    pi += step;
  // Do Initial load
  iv_xtr = vec_ld(0, pi);
  pi += step;
  ifix = vec_lvsl(roffset, pi);
  // Do front with scalar processor
  for(int i = 0; i < roffset; ++i)
    out[i] = sf(start[i]);
  po += roffset;
  // Do middle with vector processor
  for(int i = roffset; i < count; i += step) {
#ifdef DST
    vec_dst(pi, 0x10010100, 0);
#endif
    // Load unaligned
    iv = iv_xtr;
    iv_xtr = vec_ld(0, pi);
    pi += step;
    iv = vec_perm(iv, iv_xtr, ifix);
    // Do operation
    ov = f(iv);
    // Write aligned
    vec_st(ov, 0, po);
    po += step;
  }
  // Do end with scalar processor
```

```
  int starti = (count / step) *  step;
  for(int i = starti; i < count; ++i)
    out[i] = sf(start[i]);
}
```

Lai et al. (2002) only showed results without optimisation to avoid entangling the performance of the optimiser with the results. Unfortunately, since the rest of this thesis requires optimisation to attain sensible results and generally used a constant height of 960 instead of 128, the graphs are not directly comparable. So the figures have been re-timed with and without optimisations and the height changed to 960 when optimisations are on. All programs were compiled using Apple GCC 3.1 20021003. Without optimisation the compiler switch `-O0` (no optimisations) was used, while `-Os` (optimise for size) was used for the optimised version. All programs were executed 20 times, and the lowest time collected was used as the representative. The programs were executed several times because the execution time is affected by factors outside the program's control. The program could for example be interrupted by other processes. The lowest time collected was used as the representative because it is likely to be least affected by these other factors. Removing these other factors should provide for fairer comparisons between programs.

Figures 3.6, 3.7 and 3.8 show the effect of data alignment, prefetching instructions and the complexity of the functors on the speedup of on AltiVec programs. These figures show the speedup of different AltiVec programs compared to a scalar program that performs the same operation. The height of the images processed was fixed at 128 or 960. Only the width was varied. Because the image was represented as a one-dimensional array, the versions shown in the figures do not differentiate between rows and columns. Figures 3.6 and 3.7 used a functor that returns the value passed to it unchanged. Since the destination and source images were actually the same, the net effect is that the image was copied to itself. Figure 3.8 used different functors because it evaluates the effect of functor complexity on AltiVec programs.

The programs in Figure 3.6, which investigates the effect of data alignment, are detailed below:

Aligned Load and Aligned Store: This version assumes the source and destination images are aligned correctly at both ends. Since it performs only aligned loads and aligned stores, it does not handle edges and cannot handle all image sizes. This version is Algorithm 3.7 but without any special macros defined.

Unaligned Load and Aligned Store: This version uses unaligned loads with aligned stores. This version will load the source image correctly even if it is unaligned. Because it only performs aligned store, if the number of pixels is not divisible by 16 (since we were using `unsigned char`, and there are 16 `unsigned chars` in a `__vector`

`unsigned char`), then not all the pixels will be loaded. This version is Algorithm 3.7 with `UNALIGNED_LOAD` defined.

Aligned Load and Unaligned Store: This version performs aligned loading, but uses the unaligned store function discussed in Section 3.5.3. Since it only does aligned loading, it does not handle source images that are unaligned. However because it does unaligned store, the `end` pointer does not need to be aligned. It processes the remaining portion (the right edge) using the scalar processor. This version is Algorithm 3.7 with only `UNALIGNED_STORE` defined.

Unaligned Load and Unaligned Store: This version performs unaligned loading and also uses unaligned store. It is able to handle unaligned `start` and `end` pointers. It processes the remaining portion using the scalar processor. This version is Algorithm 3.7 with `UNALIGNED_LOAD` and `UNALIGNED_STORE` defined.

Unaligned Load and Unaligned Store (Optimised version): This version performs unaligned loading and also uses unaligned store. Unlike "Unaligned Load and Unaligned Store", which loads the unaligned source image to the alignment of the VPU, it loads the unaligned source image to the alignment of the destination image. Because of this, there is no need for an unaligned store. This however results in two leftover portions, one at the beginning and one at the end. Both of these portions are processed using the scalar processor. This version is Algorithm 3.7.

The programs in Figure 3.7, which investigates the effect of prefetching, are detailed below:

No DST (Aligned load and store): This is the same as the "Aligned Load and Aligned Store" version from Figure 3.6.

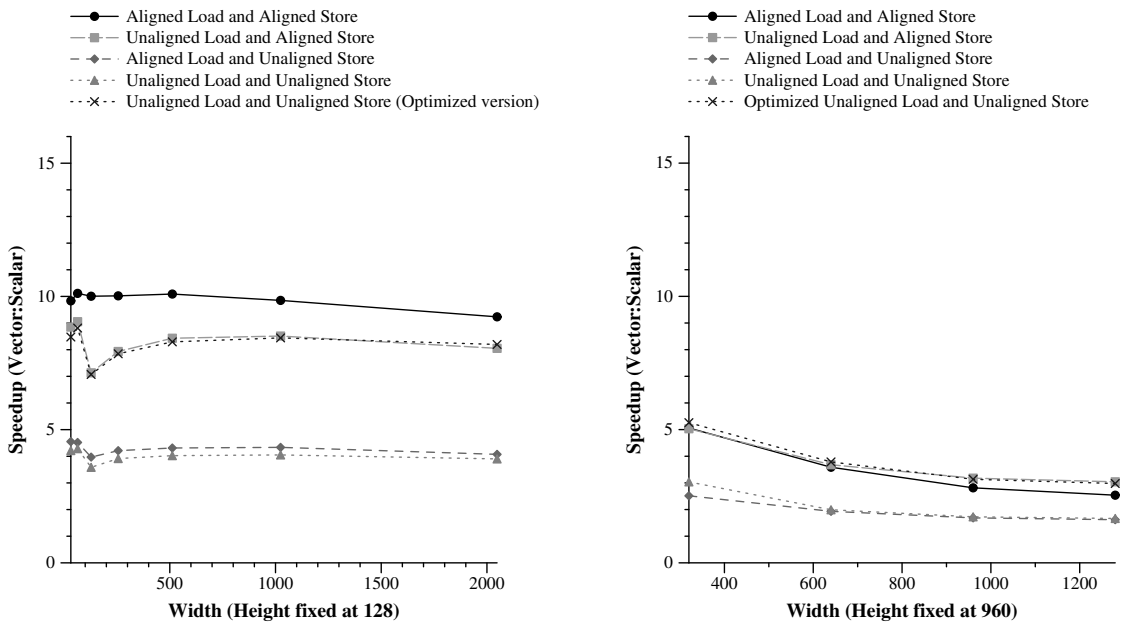
DST (Aligned load and store): This is the "Aligned Load and Aligned Store" version from Figure 3.6, but with prefetching on. This version is Algorithm 3.7 with only `DST` defined.

No DST (Optimised version): This is the same as the "Unaligned Load and Unaligned Store (Optimised version)" version from Figure 3.6.

DST (Optimised version): This is the "Unaligned Load and Unaligned Store (Optimised version)" version from Figure 3.8, but with prefetching on. This version is Algorithm 3.7 with `DST` defined.

The programs in Figure 3.8, which investigates the effect of function complexity, are detailed below. All variants used the "Aligned Load and Aligned Store" algorithm from Figure 3.6.

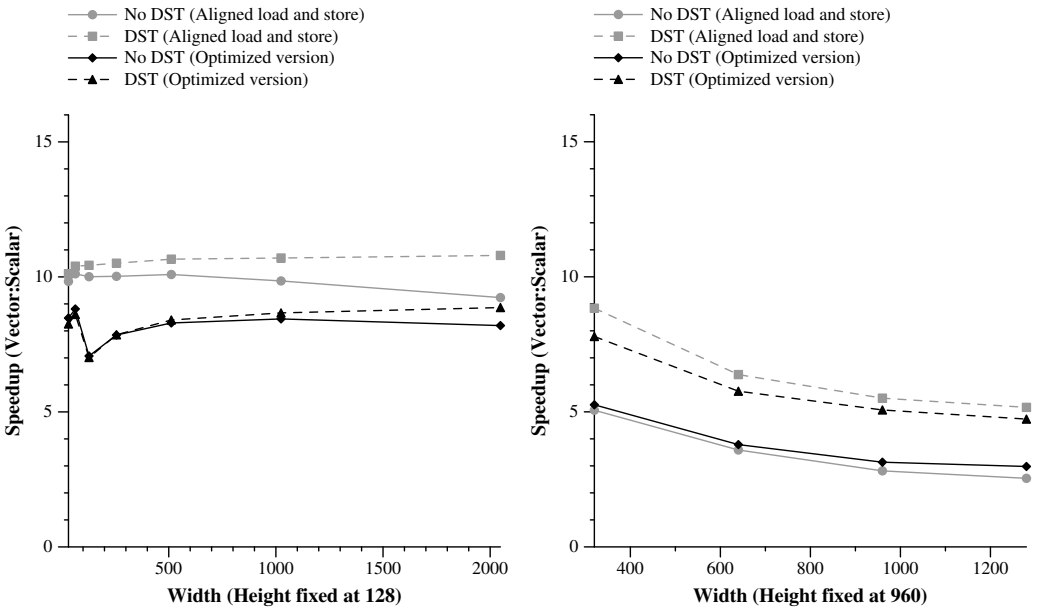
Figure 3.6 Effect of alignment on AltiVec transform functions



(a) Not Optimised

(b) Optimised for Size

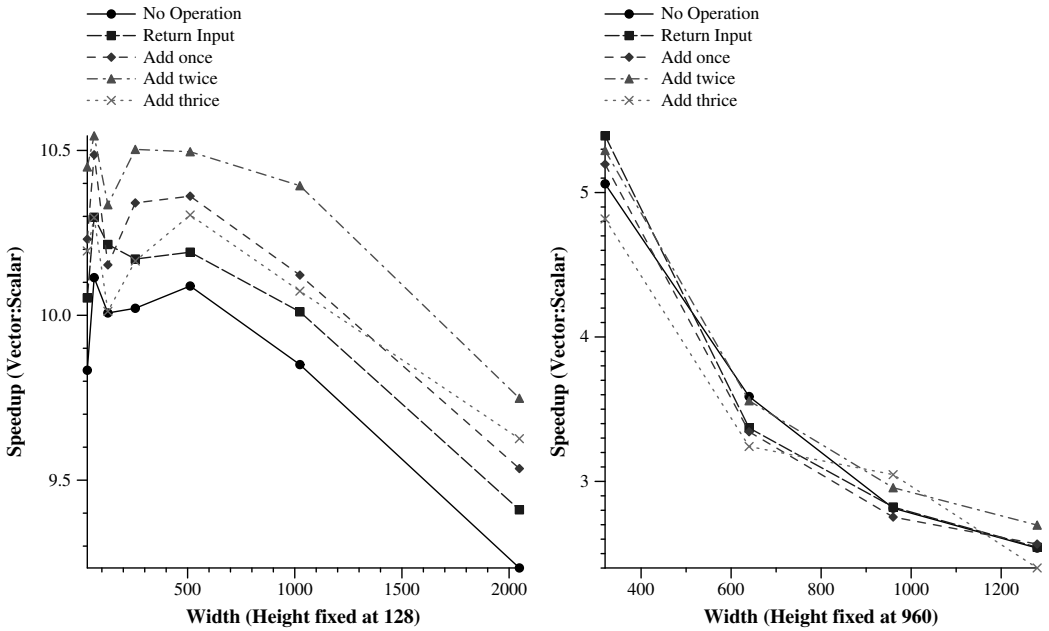
Figure 3.7 Effect of Data Stream Instructions on AltiVec transform functions



(a) Not Optimised

(b) Optimised for Size

Figure 3.8 Effect of function complexity on AltiVec transform functions



(a) Not Optimised

(b) Optimised for Size

No Operation: In this version, the functor was empty. It therefore returns an unknown value.

Return Input: In this version, the functor simply returned the input unchanged. This is the functor that was used in Figures 3.6 and 3.7.

Add Once: In this version, the functor added the input to itself once.

Add Twice: In this version, the functor added the input to itself twice.

Add Thrice: In this version, the functor added the input to itself three times.

Figure 3.6 shows that without optimisation, unaligned load and unaligned store both reduce the speed of the program. With optimisation however, unaligned load was actually faster than aligned load, which was unexpected. Unaligned store reduced the speed of the program, regardless of optimisations. Without optimisation, the optimised version from Algorithm 3.7 performed as expected — it was only a bit slower than Unaligned Load and Aligned Store.

Figure 3.7 shows that on the G4, adding prefetching, as suggested in Section 3.4.3, increases the speed of the AltiVec program, especially when optimisation was turned on. Without optimisations, prefetching increased the speedup ratio with increased image sizes. In addition, with optimisations, while aligned load was slower than unaligned load without prefetching, aligned load was faster than unaligned load when prefetching was

turned on. This suggests that unaligned load was faster than aligned load in Figure 3.6 because unaligned load waited less for data to be in the caches.

Figure 3.8 clearly shows that, without optimisation, speedup increases to a maximum and then trails off. With optimisation and a larger problem size, speedup appears to drop to a minimum. The speedup is lower for large data sizes, probably because the program becomes more dependent on memory. While increasing function complexity should reduce the memory-dependency of the algorithm thereby increasing speedups, Figure 3.8 shows mixed results. The speedup did not seem to be linearly proportional to function complexity.

In general, speedup never reached the theoretical maximum of 16. Speedups were less when optimisations were enabled, suggesting that scalar code benefited more from the optimisations performed by Apple GCC 3.1 20021003 than the AltiVec code. Furthermore, because the problem sizes were larger when optimisations were enabled, the optimised versions would have been more susceptible to delays in main memory.

3.8 Conclusion

In this chapter, a description of VPUs and issues pertaining to VPUs were presented. Issues covered were AltiVec detection, high-throughput computing, memory and cache management, vector processor specific algorithms, and different methods of programming VPUs were presented. The chapter concluded with a new investigation into the effect of alignment, prefetching and function complexity on AltiVec programs with Apple GCC 3.1 20021003.

Unlike scalar processors, which compute scalars one at a time, VPUs compute a number of scalars simultaneously. Desktop VPUs' vectors typically have fixed sizes. As a result, while vectors have fixed numbers of scalars, the number of scalars differs from one vector to another. In addition, desktop VPUs can typically only load and store data efficiently from and to aligned memory locations. Since VPUs have higher data bandwidth requirements than scalar processors, they are more susceptible to memory bandwidth limitations than scalar processors. To avoid memory being a bottleneck, the programmer can simply do more with the data currently loaded, or try to reduce memory usage. Ways to reduce memory usage are to use VPU-friendly data formats, to reduce optimisations that require lookup tables, to create smaller programs, to avoid globals and to generate constants through the use of instructions. Prefetching was also discussed as a method for loading data into the caches before it is used. This reduces the time spent loading data into the VPU. Six existing methods of programming the VPU were discussed: using the VPU directly, vector-processor wrappers, vector libraries, high-level libraries, vectorising compilers and parallel languages. Using the VPU directly, vector-processor wrappers, vector libraries, and some parallel languages allow for semantic vectorisation. Higher-

level solutions typically reduce or remove VPU constraints. For example, with vectorising compilers and high-level libraries, there is typically no need to worry about the size of vectors or aligned memory.

New to this thesis was an investigation into the effect of alignment, prefetching and function complexity on AltiVec programs with Apple GCC 3.1 20021003. Results show that unaligned stores lead to larger performance hits than unaligned loads. With optimisations, unaligned load can actually be faster than aligned load when prefetching was off. When prefetching, aligned load was clearly faster than unaligned load as expected, regardless of optimisations. Without optimisations, the highest speedup attained was about 11, while with optimisations, the highest speedup attained was about 9. The speedup attained was higher for smaller problem sizes when optimisations were on.

The VPU is difficult to use directly. Fixed vector sizes and memory requirements lead to programs that are more complicated. Not only are vector programs more difficult to write than scalar programs, they are also more susceptible to memory bandwidth limitations. Arranging data in a manner conducive for vector processing is important in overcoming memory bandwidth limitations. Since VPUs are difficult to use directly, the VPU has been wrapped in varying amounts of abstraction.

Chapter 4

Implementation Techniques

This chapter discusses existing implementation techniques that were used in this thesis. These implementation techniques affect the runtime performance and flexibility of the code. Some of the techniques in this section are provided as part of the Boost library (Boo 2003). Features missing were packaged into a utility library, called the CT (Compile-Time) library.

This chapter starts with a short introduction to generic programming, followed by techniques that allow for flexible code generation using templates. These techniques are typelists, tuples, template metaprogramming, tag dispatching, enablers, policies and expression templates. Details of how these techniques are implemented are generally omitted, because such details are not required for the purposes of this thesis.

4.1 Generic programming

Generic programming was introduced as a programming paradigm by Musser and Stepanov (Köthe 1999, Musser & Stepanov 1989, 1994). Generic programming focuses on obtaining abstract representations of efficient algorithms, data structures, amongst others (Musser & Stepanov 1989, Vandevoorde & Josuttis 2003). In C++, generic programming is sometimes defined as "programming with templates" (as opposed to object-oriented programming, which is "programming with virtual functions") (Vandevoorde & Josuttis 2003). Generic programming aims to design a framework that supports a multitude of use combinations (Vandevoorde & Josuttis 2003).

The most well-known library based on generic programming principles is the Standard Template Library (STL) (Vandevoorde & Josuttis 2003). STL provides a number of algorithms for a number of containers. Both algorithms and containers are templates. Algorithms are written in a generic way so that they can be applied to any container, instead of being a container's member functions. To enable this, STL introduces the concept of iterators which can be provided for any linear collection. STL groups the different

requirements into “concepts”. Examples of concepts specified by STL are “Input Iterator” and “Output Iterator”. As mentioned previously, VIGRA is an image-processing library based on generic programming principles. VIGRA extends the iterator design to two-dimensions.

According to Köthe (2000*c*), the generic programming approach overcomes the following limitations of traditional approaches to designing reusable software:

- Programming is more efficient, because algorithms are not repeated for different types.
- There is only a small performance penalty compared to an optimised algorithm.
- A smooth, incremental transition to generic programming is possible, because existing code is not broken while new iterators are incrementally implemented and adopted for existing data structures.
- Implementing reusable algorithms is comparable in difficulty to traditional programming. No complex inheritance hierarchies need to be designed.

Because generic programming depends on static polymorphism (Vandevoorde & Josuttis 2003), it is able to provide better performance than object-oriented programming, which is based on runtime polymorphism.

4.2 Typelists

A typelist is a type that represents a number of types. Czarnecki & Eisenecker (2000) used a typelist that contained integral constants (which were wrapped in a type) to implement a switch-statement for template metaprogramming. Alexandrescu (2001) used typelists in generic components such as Functor, Visitor and Tuple.

Typelists are easy to implement. Consider the following definitions:

```
template<typename headT, typename tailT>
struct typelist {
  typedef headT head;
  typedef tailT tail;
};
class null_typelist {};
```

Using this definition a typelist of char C++ types can easily be defined as follows:

```
typelist<char, typelist<unsigned char, typelist<signed char,
  null_typelist> > >
```

Table 4.1 Typelist metafunctions in CT

Expression	Return Type	Notes
`type_at<TL, i>::type`	Type	Returns type at index `i` in typelist `TL`. Compilation fails if not found
`type_at_non_strict< TL, i, D>::type`	Type	Returns type at index `i` in typelist `TL`. Return `D` if not found, or `void` if `D` is not passed in
`index_of<TL, X>::value`	`int`	Returns the index of `X` in the typelist `TL` Returns `-1`, if not found
`find_first<TL, E>::type`	Type	Returns first type in `TL` where `E<X>::value` is `true`
`length<TL>::value`	`int`	Returns number of types in `TL`
`append<TL, X>::type`	`typelist`	Result is `TL` with `X` at the end
`erase<TL, X>::type`	`typelist`	Result is `TL` without first `X`
`erase_all<TL, X>::type`	`typelist`	Result is `TL` without any `X`
`no_duplicates<TL>::type`	`typelist`	Result is `TL` without any duplicates
`apply<TL, E>::type`	`typelist`	Each type in `TL` is changed to `E<X>::type`
`keep_only<TL, E>::type`	`typelist`	Result is `TL` where all `E<X>::value` is `true`
`contains<TL, X>::value`	`bool`	Returns `true` if `X` is in `TL`
`any<TL, E>::value`	`bool`	Returns `true` if any `E<X>::value` is `true`
`all<TL, E>::value`	`bool`	Returns `true` if all `E<X>::value` is `true`

More information about typelist implementation can be found in Alexandrescu (2002). The Loki library written by Alexandrescu (2001) provides an example of how typelists can be implemented and some template metafunctions for retrieving types from the typelist.

For this thesis, typelists are provided by the CT library. CT provides several metafunctions for querying and using typelists. While most of these metafunctions are derived from the Loki library, `apply`, `keep_only`, `contains`, `any` and `all` are new to CT. The Loki library contains other metafunctions that CT did not provide because this thesis did not require those functions. The list of template metafunctions for operating on and querying typelists, provided by CT, is shown in Table 4.1.

4.3 Tuples

A tuple is a fixed-size collection of elements. An example of a tuple in C++ is `std::pair`. Some languages, like Perl and Python (van Rossum 2003), support tuples as part of the language. While C++ does not have native tuple support, tuple implementations are available from the Loki (Alexandrescu 2001) and Boost libraries. Loki's tuples are defined

using a typelist, while Boost tuples have their types passed in as separate arguments. For example, a tuple of a `char` and a `short` is `Loki::Tuple<TYPELIST_2(char, short)>` and `boost::tuple<char, short>` in Loki and the Boost tuple library respectively.

While tuples exist in both Loki and Boost, this thesis used tuples provided by CT. CT's tuples use typelists to define their elements, like Loki's, and also support operators, like Boost's. This was to allow CT's tuples to easily represent multi-channel pixels in the generic, vectorised, machine-vision library developed as part of this thesis. In the machine-vision library developed, channel types are represented as a typelist. For example, a `char` RGB image would be `vvis::base_image<TL>` where `TL` is `CT_TYPELIST3(char, char, char)`. Since CT's tuples define their members using a typelist, the pixel for this image can be represented by `ct::tuple<TL>`. Operator support is important because it reduces the number of functors that are needed. For example, the same expression `a+b` will also instantiate successfully even if `a` and `b` are `ct::tuples`.

The type `ct::tuple` inherits from `boost::tuple`. This gives `ct::tuple` the clean `boost::tuple` syntax for accessing elements and the logical and comparison operators that are supported by `boost::tuple`. The type `ct::tuple` was extended to support even more operators. Any operator performed on `ct::tuple` is the same as performing the operation on each of its elements. In addition, `ct::tuple` supports dereferencing, references and pointers. Dereferencing, references and pointers were needed in the generic, vectorised, machine-vision library to write algorithms that operate on any number of channels.

The operators in general were implemented using `meta::EFOR` (see Section 4.4). The `EFOR` command was used to generate the appropriate operation on each element of the tuple. The operators apply their operations orthogonally; each element is computed independently.

```
a()       a(v)      a(v1, v2 ...)
a = v     a = b
*a
++a       a++       --a      ++a
a += v    a += b    a + v    a + b
a -= v    a -= b    a - v    a - b
a *= v    a *= b    a * v    a * b
a /= v    a /= b    a / v    a / b
a %= v    a %= b    a % v    a % b
```

The type `ct::tuple` does not override the logical and comparison operators because they are provided by `boost::tuple`. Since tuples contain more than one element, logical and comparison operators can have more than one meaning. Does `a == b` return a tuple which contains the results of whether each members of the tuples `a` and `b` are equal, return a

single boolean indicating if all members of `a` and `b` are equal, or return a single boolean indicating if any corresponding members of `a` and `b` are equal? Boost's tuples' comparison and logical operators return a single boolean that indicates if all members of `a` and `b` are equal. For the operation to be meaningful with vectors, returning a vector of booleans is probably more meaningful. The type `ct::tuple` returns a single boolean that indicates if all members of `a` and `b` are equal, because it is based on `boost::tuple`.

4.4 Template metaprogramming

In 1994, Erwin Unruh presented a program that used the compiler to calculate prime numbers as error messages (Veldhuizen 2000). This program demonstrated that the C++ template mechanism can be used to perform computations at compile time. In fact, according to Czarnecki & Eisenecker (2000), C++'s templates when used with a number of other C++ features form a Turing-complete, compile-time sub-language of C++. Being Turing-complete, theoretically, any algorithm that is implementable in any other Turing-complete language like C++, C, and Java is also implementable using this sub-language of C++. Czarnecki & Eisenecker (2000) actually show how if-statements, switch-statements, for-loops, while-loops, and linked lists can be implemented in this sub-language. In practice however, technical limitations such as compile time and compiler limits can restrict what can be implemented.

The ability to perform computations during compile time is of little use practically, since the sub-language is not particularly easy to follow and the C++ compiler is not exactly a fast interpreter. This computation can however be used to conditionally generate code. This use of templates to generate code is referred to as template metaprogramming. Other methods of code generation during compile time include static metaprogramming techniques, which includes the use of the preprocessor, or extensions to the compilation process using tools like Open C++ (Chiba 1998*b*, 1995). Advantages that template metaprogramming has over the preprocessor are that variables in the program itself can be used, and more complex code can be generated. Template metaprogramming's advantage over extensions to the compilation process is that it is available to anyone with a C++ compiler that supports enough of the Standard C++; it does not require any additional tools or additional steps.

Veldhuizen (2000), Veldhuizen (1995*a*), and Czarnecki & Eisenecker (2000) all discuss template metaprogramming. Czarnecki & Eisenecker (2000) is particularly thorough, discussing the implementation of constructs such as if-statement, for-loop, while-loop, switch-statement and so on. Czarnecki & Eisenecker (2000)'s metaprogramming constructs are available in the Meta library. The example below uses the Meta library's for-statement, `EFOR`, to generate code that prints out the numbers from 0 to 2.

```
struct print {
  template<int i> struct Code {
    static void exec() {
      std::cout << i << std::endl;
    }
  };
}
meta::EFOR<0, meta::Less, 3, +1, print>::exec();
```

The `meta::EFOR` statement expands to:

```
std::cout << 0 << std::endl;
std::cout << 1 << std::endl;
std::cout << 2 << std::endl;
```

While this example may seem unremarkable, template metaprogramming is used later to achieve performance benefits and to manipulate tuples. In addition, because `EFOR` expands at compile-time, there are no overheads associated, unlike a `for` loop. CT contains a copy of the Meta library, with additional `EFOR`s.

Boost's MPL library (Gurtovoy & Abrahams 2002, Boo 2003) also provides some template metaprogramming constructs, namely the IF constructs. Because Boost libraries were the first preference, the author used the IF constructs from the Boost MPL library and the `EFOR` from Meta. IF constructs are used to make decisions at compile-time. They allow us to, for example, select different implementations using complex boolean logic with the help of tag dispatching (see Section 4.5) or enablers (see Section 4.6).

Templates that return values are referred to as template metafunctions. These values are either constant values or types. An example of a template metafunction would be the IF construct from `boost::mpl`.

4.4.1 Traits

Traits provide information about types at compile-time. An example of a traits template in C++ is `std::numeric_limits`, which provides information such as minimum and maximum values, and signedness of types. Traits are important when automating the generation of more complicated generic algorithms (Köthe 1999).

Traits allow a generic function to find other types, and to make decisions based on the characteristics of the type. For example, VIGRA uses a template called `vigra::-PromoteTraits` to decide what a given type should be promoted to. CT provides several promotion templates which are discussed in Section 4.4.2.

The Boost library includes a type traits library that provides information about types and includes template metafunctions that add qualifiers such as `const`, reference or pointer

to and from types, or remove them. The Mac OS X VecLib framework, which is part of the Accelerate framework in Mac OS 10.3, clashes with the type traits library because `vecLib.h` defines a macro named `check` which interferes with a `check` function used in `type_traits.hpp`. To solve this, CT includes a `boost_type_traits.hpp` header file that undefines the `check` macro before including the `type_traits.hpp` header file.

4.4.2 Type promotions

Templated programs sometimes need to promote types. For example, when summing elements, the sum's type should be larger than the operands to avoid overflow errors. Type promotion is required in template programming to prevent data from being truncated too early. VIGRA has its own type promotion template, `vigra::PromoteTraits` (Köthe 2001). `vigra::PromoteTraits` returns the appropriate return type of arithmetic arguments with two arguments.

CT provides several promotion template metafunctions — `promote`, `promote_signed` and `promote_float`. These return a larger type with the same characteristics, a larger signed type and a larger float type respectively. A single `promote` is inadequate because sometimes a signed or a float type is required. Saturated subtraction for example requires a larger signed type.

While `vigra::PromoteTraits` was specialised for each type, the promotion template metafunctions in CT operate by inspecting typelists. They return the first type larger than the current type in a number of predefined typelists. On failing to find a larger type, the original type is returned. Algorithm 4.1 shows how `promote` is implemented for a single type. `priv::find_larger` is a template metafunction that returns the first type in a given typelist with a larger size.

The metafunctions `promote_signed` and `promote_float` operate in a similar fashion, except they limit the typelist that they search for types to either signed or real types respectively. For example, `promote_signed` would set `result_tl` in Algorithm 4.1 to either `priv::signed_integer_tl` or `priv::real_tl` for integers and floats respectively. The typelist `result_tl` would never be set to `priv::unsigned_integer_tl`.

CT's promotion template metafunctions also accept typelists as template arguments. In such situations, the promotion template metafunction first determines the characteristics of the resultant type by inspecting the characteristics of all the types in the typelist. It then returns the the first type larger than the largest type in the typelist from the appropriate typelist. If there is no type larger than the largest type in the typelist, then it returns the largest type in the typelist.

Table 4.2 summarises the different promotion template metafunctions available in CT.

Algorithm 4.1 `promote` template metafunction for a single type

```
// List of types
namespace priv {
  // Signed integer types
  typedef ct::cons_tl<
    signed char,
    signed short int,
    signed int,
    signed long int>::type signed_integer_tl;
  // Unsigned integer types
  typedef ct::cons_tl<
    unsigned char,
    unsigned short int,
    unsigned int,
    unsigned long int>::type unsigned_integer_tl;
  // Real types
  typedef ct::cons_tl<float, double, long double>::type real_tl;
} // End of priv namespace

// promote template metafunction
template<typename T> struct promote {
public:
  typedef typename boost::mpl::if_c<
    boost::is_integral<T>::value, // Is T an integer?
    typename boost::mpl::if_c<
      std::numeric_limits<T>::is_signed, // Is T a signed integer?
      priv::signed_integer_tl,  // T is a signed integer
      priv::unsigned_integer_tl // T is an unsigned integer
      >::type,
    typename boost::mpl::if_c<
      boost::is_float<T>::value, // Is T a real?
      priv::real_tl, // T is a real
      null_type      // T is unknown -- cannot promote
      >::type
    >::type result_tl;
  typedef typename priv::find_larger<
    result_tl,  // Typelist to look for type in
    T      // Look for first type larger than T
    >::type promoted_type;
public:
  // Check if larger type found
  typedef typename boost::mpl::if_c<
    boost::is_same<promoted_type, null_type>::value,
    T,            // If NOT, then use T
    promoted_type // Otherwise use larger_type
    >::type type;
};
```

Table 4.2 Type promotion metafunctions provided by CT

Expression	Return Type	Notes
`promote<X>::type`	Type	Returns type larger than `X`.
`promote_signed<X>::type`	Type	Returns type larger than `X`. The return type is always signed.
`promote_float<X>::type`	Type	Returns type larger than `X`. The return type is always a float.

Type `X` can be a typelist.

4.5 Tag dispatching

Tag dispatching is a method of selecting implementations based on conditions at compile time. Tag dispatching works by specialising class template for different types called tags, which are usually just empty types. Consider the following example:

```
struct jump {};
struct crawl {};

template<typename tagT> struct do_move {
  static void exec() { ... }
};

template<> struct do_move<jump> {
  static void exec() {
    // We move by jumping here
  }
};

template<> struct do_move<crawl> {
  static void exec() {
    // We move by crawling here
  }
};

template<typename T> void move(T) {
  typedef boost::mpl::if_<can_jump<T>::value,
    jump, crawl>::type tag;
  do_move<tag>::exec();
}
```

Calling `move(X)` will call either `do_move<jump>::exec()` or `do_move<crawl>::exec()` depending on the result of the template metafunction `can_jump`.

Instead of tag dispatching, enablers (see Section 4.6) can also be used. Tag dispatching is more verbose, but is currently supported by more C++ compilers. Since tag dispatching requires an extra function call, if the compiler does not inline the second call, then the overhead of an additional function call is introduced.

4.6 Enablers

Hinnant et al. (2003) describe a method of enabling or disabling functions and classes based on computations that can be performed at compile-time. Enablers work because "erroneous" template definitions do not cause compilation errors. Instead the compiler is instructed to remove the erroneous prototype from the overload resolution list. Enablers operate by creating erroneous definitions deliberately to disable functions.

Both enabler and disabler templates are quite simple. Shown below is the enabler from the CT library.

```
template<bool B, typename T = void>
struct enable_if {
  typedef T type;
};
template<typename T>
struct enable_if<false, T> {};
```

The type `enable_if<false, T>::type` is invalid, because there is no type defined in `enable_if<false, T>`. The disabler is simply the `enable_if` template except `disable_if<true, T>` is undefined. Therefore

```
typename enable_if<false, void>::type func();
```

does not actually define the function `func()`. If however, `enable_if` was passed true, the function `func()` would become defined. For example, in the following example, func would be defined if the `sizeof(T)` was 1.

```
template<typename T>
  typename enable_if<sizeof(T) == 1, void>::type func();
```

The `enable_if` template can also be used to enable classes conditionally, as shown below. The

```
template<typename T, typename specializedT = void> class A;

template<typename T>
class A<T, typename enable_if<sizeof(T) == 1>::type> {
  // This is defined only if sizeof(T) == 1
};
```

Some compilers have trouble with enablers as is and will require some massaging to compile (Hinnant et al. 2003). Apple GCC 3.1 20021003, which was the main compiler used in this thesis requires each function that uses an enabler to be in a separate namespace. There is no need to place different classes in separate namespaces though. For example, the following will not compile in Apple GCC 3.1 20021003:

```
template<typename T>
  typename ct::enable_if<sizeof(T) == 1, void>::func() {
    // Implementation for types that have size of 1
    // goes here
  }
template<typename T>
  typename ct::enable_if<sizeof(T) == 2, void>::func() {
    // Implementation for types that have size of 2
    // goes here
  }
```

Instead, the user must write:

```
namespace a {
  template<typename T>
    typename ct::enable_if<sizeof(T) == 1, void>::func() {
      // Implementation for types that have size of 1
      // goes here
    }
}
namespace b {
  template<typename T>
    typename ct::enable_if<sizeof(T) == 2, void>::func() {
      // Implementation for types that have size of 2
      // goes here
    }
}
using a::func;
```

```
using b::func;
```

Combined with typelists, it becomes easy to enable functions for a range of types. This technique can be used to implement vector-processor implementations for abstract VPUs (see Chapter 5 for more information about abstract VPUs).

4.7 Policies

Policy-based class design aims to create complex classes through the use of many small classes (policies) (Alexandrescu 2001). Each policy is responsible only for one specific behavioural aspect, and it provides this aspect through a well-known interface. A single class can be composed from many policies. Since policies are usually designed to be orthogonal, it is possible to mix and match policies and thereby achieve a large number of behavioural sets using a small number of policies.

In Chapter 1, Policy-Based Class Design of Alexandrescu (2001), the idea of policies is introduced and different methods of implementation are discussed. Policies in this thesis are implemented as template parameters to the main class. The policies are usually inherited, though this is not always the case. When inheriting, the destructor of the policy class is protected. This is to enable the class to be non-virtual, so that the main class does not have to have additional cost, while allowing pointers to the base class (policy) to be handled correctly on deletion.

Take for example the following definition of an image. The `image` class inherits from the storage policy `storageP`. The details of how data in the image is stored is handled by the storage policy. The template specialisations `image<int, contiguous_storage>` and `image<int, gworld_storage>` would keep their data differently.

```
template<typename T, template<typename> storageP>
class image : public storageP<T> {
  ...
};

template<typename T>
class contiguous_storage {
private:
  ~contiguous_storage() { ... }
public:
  const T pixel(const int x, const int y) const { ... }
  ...
};
```

```
template<typename T>
class gworld_storage {
private:
  ~gworld_storage() { ... }
public:
  const T pixel(const int x, const int y) const { ... }
  ...
  GWorld gworld() { ... }
};
```

By using inheritance, it is possible to define different functions for different policies. For example, a storage policy that kept its data in a QuickTime GWorld would have a function to return a reference to that GWorld.

Instead of inheritance, policies can also be implemented using static functions. Under this scheme, the policies are `structs` which have public static functions. The main class then calls these functions. When using static functions however, the policy class cannot keep its own data. The image example can also be implemented using static functions, as illustrated below.

```
template<typename T, template<typename T> storageP>
class image {
  typedef storageP<T> storage_policy;
public:
  image(const int width, const int height)
  : _width(width), _height(height) {
    _data = storage_policy::allocate(width, height);
  }
  ~image() {
    storage_policy::deallocate(_data);
  }
  const T pixel(const int x, const int y) const {
    storage_policy::pixel(_data, _width, _height, x, y);
  }
  data_type gworld() {
    return storage_policy::gworld(_data);
  }
private:
  typename typename storage_policy::data_type _data;
  int _width, _height;
};
```

```
template<typename T>
struct contiguous_storage {
  typedef T* data_type;
  static T* allocate(const int width, const int height) {
    return new data_type[width*height];
  }
  static void deallocate(data_type data) {
    delete[] data;
  }
  static T pixel(data_type data, const int w, const int h,
  const int x, const int y) const {
    return data[y*w+x];
  }
};

template<typename T>
struct gworld_storage {
  typedef GWorldPtr data_type;
  static GWorldPtr allocate(const int width, const int height) {
    ...
  }
  static void deallocate() {
    ...
  }
  static T pixel(data_type data, const int w, const int h,
  const int x, const int y) {
    ...
  }
  static GWorldPtr gworld(data_type data) {
    return data;
  }
};
```

While this technique also allows different policies to support different functions, the function must be defined in the main class. Policies implement only the functions that they required. If the user tries to use a function that the current policy does not have an implementation for, then the code will fail to compile. In the previous example, if `a` is of type `image<char, contiguous_storage>`, then `a.gworld()` will cause a compilation error. If however, `a` is of type `image<char, gworld_storage>`, then `a.gworld()` will

succeed. Calling only `a.pixel(0, 0)` in both cases will not result in any compilation errors.

Policies can be more fine-grained, and they can control specific issues such as memory allocation. Policies should be orthogonal where possible so that they can be combined in different ways to produce a large number of possibilities from a small code set.

4.8 Expression templates

It is well known that increasing the number of elements in a vector increases overheads when function overloading is used (Veldhuizen 1995*b*, Blinn 2000). Consider the expression `R=A+B+C+D`. Using function overloading, the compiler will generate code similar to the following:

```
vvm::vector temp1 = A + B;
vvm::vector temp2 = temp1 + C;
vvm::vector temp3 = temp2 + D;
R = temp3;
```

Two reasons why evaluating expressions in this manner is slower than hand-coded programs are because more temporaries are used, and each statement contains its own for-loop (or its unrolled equivalent) (Veldhuizen 1995*b*, Blinn 2000).

The preferred execution method is to evaluate the entire expression for each element in the `vvm::vector`. Hand-coded C++ would look something like the following:

```
R.scalar(0) = A.scalar(0)+B.scalar(0)+C.scalar(0)+D.scalar(0);
R.scalar(1) = A.scalar(1)+B.scalar(1)+C.scalar(1)+D.scalar(1);
// And so on....
```

Expression templates is a technique that can help the compiler generate this faster implementation. It can generate the faster implementation because it delays evaluating expressions until the entire expression is used. With expression templates, `A+B+C+D` returns an object that represents the addition of `A`, `B`, `C` and `D`. When `R=<expression object>` is evaluated, the entire expression is evaluated simultaneously and the result is assigned to `R`. Since expression templates represent expressions as objects, they allow expressions to be passed to a function as an argument. This second feature allows users to create functors directly from expressions when using generic libraries.

More information on expression templates are provided by Veldhuizen (1995*b*), Blinn (2000), Becker (2003), and Köthe (2000*a*). The code involved in expression templates is discussed in more detail later when it is used.

4.9 Conclusion

In this chapter, generic programming was discussed briefly followed by uncommon C++ implementation techniques that were used in this thesis. These techniques are typelists, tuples, template metaprogramming, traits, type promotion, tag dispatching, enablers, policies and expression templates. While all the techniques were developed by others, the author introduced a few new template metafunctions for manipulating typelists, more operations for tuples and a type promotion implementation that uses the compiler to calculate the promoted type.

New template metafunctions, that were not part of the Loki library, were introduced for manipulating typelists. These template metafunctions are `ct::contains`, `ct::apply`, `ct::keep_only`, `ct::any` and `ct::all`. These additional template metafunctions were introduced to facilitate the creation of the generic, vectorised, machine-vision library. The metafunction `ct::contains` is just a short-hand for `ct::index_of`[1]`<TL, T> != -1`. The metafunctions `ct::apply` and `ct::keep_only` were used extensively to convert types when dealing with tuples. In the generic, vectorised library, developed as part of this thesis, `ct::apply` was used to change a typelist of scalars to a typelist of VVM's vectors. The metafunction `ct::keep_only` was used in the implementation of an abstract VPU to prune a typelist of fundamental types to typelists of unsigned integers, signed integers and floats. The metafunctions `ct::any` and `ct::all` were used to make decisions based on types in the typelist at compile-time. They made it easier for the generic, vectorised library to select algorithms based on how pixels are arranged in memory.

The type `ct::tuple` supports more operators than Loki's or Boost's tuple. In the generic, vectorised, machine-vision library that was developed as part of this thesis, `ct::tuple` is used to represent a multi-component pixel. Operators allow the creation of templated functors that can be instantiated for both single-channel and multi-channel images.

Typelists and template metaprogramming were used to calculate promoted types. Template metaprograms select larger appropriate types by iterating through typelists of possible answers. Using the compiler to calculate the promoted type leads to shorter, less error-prone code. It also makes it very easy to always return a signed or float promoted type, and to add more types to consider as candidates for promoted types.

[1] `ct::index_of` is derived from Loki's `Loki::IndexOf`. See Table 4.1 for more information.

Chapter 5

Abstract Vector Processor

There are many different desktop vector technologies currently available. These vector technologies have different instruction sets and associated costs as well as different vector sizes. Occasionally, different processors that support the same vector technology have different costs when executing the same instruction. For example, the AltiVec prefetch instructions are more likely to cause stalls on a G5 than on a G4 (App 2003*c*). These differences mean that programs must be written specifically for each vector technology, and sometimes require tuning for a specific VPU. If the program is to support multiple vector technologies, then it has to be reimplemented.

To solve this problem, and thus allow generic programming to be applied to low-level vector programs, we propose the abstract vector processing unit (VPU). The abstract VPU is a virtual VPU that represents a set of real VPUs, a virtual VPU that has an idealised instruction set and constraints common to the VPUs that it represents. The idealised instruction set and common constraints together allow programs that use the abstract VPU to be portable across the set of real VPUs being represented and to perform efficiently. The abstract VPU is suitable for programmers who want the performance advantages of using the VPU directly without being restricted to a particular VPU.

This chapter starts with a definition of what an abstract VPU is, followed by features that are the essence of an abstract VPU and discusses other features that would make the abstract VPU useful. This is wrapped up with a look at the differences between the abstract VPU and other ways of programming the VPU.

5.1 What is an abstract vector processor?

Real VPUs typically share many common constraints and instructions. For example, AltiVec, MMX and 3DNow! share constraints such as fixed vector sizes and faster access to aligned memory addresses, and share instructions such as saturated addition and subtraction. Instead of writing different programs for different VPUs, we propose writing a

single program for an abstract VPU, a virtual VPU that represents set of real VPUs with an idealised instruction set and common constraints.

The idealised instruction set makes the abstract VPU easier to use and portable. It also removes the need for programmers to handle instruction availability, and allows the programmer to express his logic using ideal VPU constructs, free from any real VPU's inadequacies. Instructions should be provided for all types. Low-level instructions, such as prefetching memory, should also be available. An idealised instruction set implies a scalar implementation for all functions. This scalar implementation is referred to as the emulation layer. Vector-processor implementations provide mappings for abstract VPU functions to real VPU instructions where applicable. An abstract VPU should enable appropriate vector-processor implementations when real VPUs are available. Since a vector-processor implementation is unlikely to provide mappings for all functions, expressions involving the use of both the emulation layer and the vector-processor implementation must be supported.

The abstract VPU should have the common constraints of the real VPUs that it represents. These constraints ensure that abstract VPU programs are comparable performance-wise to the real VPU programs. The more ambiguous the constraints, the more real VPUs the abstract VPU can represent. On desktop VPUs, examples of common constraints are short vectors, fixed vector sizes and efficient memory access only at aligned memory addresses. An abstract VPU should not try to represent a real VPU that has incompatible constraints because the resultant program is likely to be significantly slower than using the real VPU directly. For example, an abstract VPU with fixed vector sizes should not try to represent VPUs with variable vector sizes. VPUs that support variable vector sizes typically have larger start and stop overheads.

In order to be more useful, apart from having an idealised instruction set and common constraints, the abstract VPU should be easy to use, cater for undefined behaviour, have a performance measurement tool and be zero-cost. An easy-to-use interface improves programmer productivity, allowing more programmers to use VPUs. Undefined behaviour gives vector-processor implementations more choices, allowing them to map instructions more efficiently. A performance measurement tool allows the programmer to evaluate the cost of an abstract VPU implementation in his environment. Finally, zero-cost would allow abstract VPU programs to execute as fast as hand-coded VPU programs that perform the same operation in the same manner. Because an abstract VPU's constraints might be different from the real VPUs, an abstract VPU program might perform more work. For example, an abstract VPU that always processed 16 scalars simultaneously would still add 16 `int`s even if only 8 `int`s were needed. Since an AltiVec program would be able to add exactly 8 `int`s, the abstract VPU program adds an additional 8 `int`s. It is not expected that the abstract VPU will need to do much extra work. Zero-cost is important because programmers turn to the VPU for performance speedups.

The abstract VPU's idealised instruction set and common constraints allow programmers to write code that appears to target a set of real VPUs simultaneously.

5.2 Differences between the abstract vector processor and current methods

The differences between programming the VPU through the use of the abstract VPU and other existing techniques are discussed in this section. How each method differs from the abstract VPU is discussed.

5.2.1 Vector processor

Unlike the abstract VPU, programming the VPU directly results in code that is not portable to other vector technologies. The code however should be portable to other VPUs that implement the same vector technology, though some adjustments might be required to cater for differences in instruction sets and execution costs. Using the VPU directly allows the programmer to express the algorithm in VPU logic — semantic vectorisation.

Using the abstract VPU would be better, because a good implementation would be easier to use and allow code to be ported to different architectures with minimal losses in speed. Programming the VPU directly however might provide users access to instructions that the abstract VPU might not have and allows programmers to tune their programs specifically for a single VPU.

5.2.2 Vector-processor wrappers

Vector-processor wrappers place less emphasis than the abstract VPU on being cross-platform. The main aim is to provide a zero-cost, easy-to-use interface to the real VPU. While a vector-processor wrapper is likely to have the same instruction set and constraints as the real VPU, the abstract VPU has an idealised instruction set and more general, common constraints.

5.2.3 Vector libraries

The main difference between vector libraries and the abstract VPU is that vector libraries typically have no constraints. For example, even on the desktop computer, they are unlikely to force the programmer to use only aligned memory or fixed size vectors. While the abstract VPU is concerned with representing a set of real VPUs simultaneously, the vector library is concerned with making vector manipulation easier.

Vector libraries should be easier to use than the abstract VPU, because they have less constraints. However, the abstract VPU provides more fine-grained control. Writing hand-coded vector programs should be possible with an abstract VPU. For example, Lai et al. (2002) presented the same operation implemented in numerous ways to handle different alignment requirements. The operation can be implemented in as many ways using an abstract VPU, but would have only one implementation when using a vector library.

5.2.4 High-level libraries

Like the abstract VPU, programmers do not need to know if they are using a VPU or not. Unlike the abstract VPU however, users of such libraries would not be able to perform semantic vectorisation. While the library routines would be optimised, there is no support for the programmer who desires to use the VPU for routines that are not supported by the library, since it is outside the functionality that the library provides.

While different implementations of the same library that target different vector technologies can be created, creating these different ports involves both more development and maintenance. The abstract VPU could be used internally by libraries instead of hard coding for a specific VPU, to remove the need for different ports.

5.2.5 Vectorising compilers

Vectorising compilers change a scalar program to a vector program automatically. Like the abstract VPU, they can generate code that targets different VPUs (or no VPU) from the same source code. However, unlike the abstract VPU, they do not provide access to the VPU, nor are they likely to provide any vector-oriented routines. Because of this, vectorising compilers do not support semantic vectorisation. The programmer can however use one of the other methods discussed in this section to access the VPU.

5.2.6 Parallel languages

Parallel languages seek to provide the user with an easy syntax to express parallelism in their programs. The idea is to allow the user to focus more on the problem and less on details of how to pass data correctly between threads, between computers, to the VPU and so on.

The abstract VPU provides access to a virtual VPU that is representative of a number of VPUs. Unlike parallel languages, which can also map to multiple computers and processors, the abstract VPU only provides access to a VPU. With parallel languages, the user is typically insulated from the idiosyncrasies of real VPUs. The abstract VPU on the other hand is for programmers who wish to use the real VPU, but would prefer not to be locked to a single standard, and are looking for an easier-to-use alternative. Being

a VPU, it is expected to have constraints similar to the VPU. It is expected to have low-level instructions. An abstract VPU could have low-level instructions, like prefetching, and low-precision multiplications and divisions, which are unlikely to exist in a parallel language.

5.3 Conclusion

This chapter presented the abstract VPU. The abstract VPU is a new idea, introduced to allow programmers to write low-level, efficient, cross-platform VPU programs. This is required for generic vector programs. A description of the abstract VPU and how it differs from existing methods of programming the VPU was covered.

The abstract VPU represents a set of real VPUs with a virtual VPU that has an idealised instruction set and common constraints. In order to be more useful, apart from having an idealised instruction set and common constraints, the abstract VPU should be easy to use, cater for undefined behaviour, have a performance measurement tool and be zero-cost. The abstract VPU is suitable for programmers who want to use the VPU directly and also want to produce portable programs easily.

The idealised instruction set makes the abstract VPU easier to use and portable. It removes the need for programmers to handle instruction availability, and allows the programmer to express his logic using ideal VPU constructs, free from any real VPU's inadequacies — semantic vectorisation. An idealised instruction set implies a scalar implementation for all functions. This scalar implementation is referred to as the emulation layer. Vector-processor implementations provide mappings for abstract VPU functions to real VPU instructions where applicable. An abstract VPU should enable appropriate vector-processor implementations when real VPUs are available. Since a vector processor implementation is unlikely to provide mappings for all functions, expressions involving the use of both the emulation layer and the vector-processor implementation must be supported.

The abstract VPU has the common constraints of the real VPUs that it represents. These constraints ensure that abstract VPU programs are comparable performance-wise to the real VPU programs. The more ambiguous the constraints, the more real VPUs the abstract VPU can represent. An abstract VPU should not try to represent a real VPU that has incompatible constraints because the resultant program is likely to be significantly slower than using the real VPU directly. The abstract VPU's constraints ensure that any program written for it would be essentially a hand-coded vector program, since programs would be structured in a manner that is conducive to processing by the real VPUs represented by the abstract VPU.

The abstract VPU conceptually sits between vector-processor wrappers and vector libraries. The abstract VPU seeks to provide access to VPUs in a cross-platform manner.

While vector processor wrappers also provide access to the VPU, they are not designed to be cross-platform. Vector libraries, on the other hand, provide portability, but they do not provide direct access to the VPU. Since vector libraries typically do not preserve the constraints of the VPU, the programmer does not have fine-grained control over the execution of his program. High-level libraries and automatic vectorising compilers do not support semantic vectorisation, because the user does not control what vector instructions are generated. Parallel languages, like vector libraries, hide the idiosyncrasies of real VPUs from the programmer, and thus do not provide the programmer with as much control. Parallel languages also map to multi-threaded and multi-processor programs.

Chapter 6

Virtual Vector Machine's Design

In this chapter, the design of an abstract vector processor called Virtual Vector Machine (VVM) is discussed. VVM was designed to facilitate the creation of a generic, vectorised, machine-vision library. VVM is designed to be an API to enable programs that use the VVM API to use different VVM implementations where required. Different VVM implementations might have different runtime costs with different compilers, and might provide support for different VPUs.

In the interest of supporting the creation of a generic, vectorised library, VVM is templated, has constant scalar count, and provides trait information. Furthermore, where possible, VVM operators and functions perform the same operation as scalar operators and functions that have the same name. For example, `operator+` performs addition where overflow behaviour is undefined when applied to both scalars and VVM vectors. Most of the vector manipulation functions provided by VVM are derived from AltiVec. AltiVec was chosen as the most influential platform because of all the existing desktop vector-processor technologies available at the time of writing, it is the only one to have a dedicated vector processor unit. To support the creation of a machine-vision library, VVM also provides some "unnecessary" functions used commonly in machine-vision. An example of these "unnecessary" functions is the `madd` function which performs a multiplication and an addition as a single instruction. Since it involves only addition and multiplication, both of which are provided by VVM, `madd` does not allow the programmer to perform any new operations. The main advantage of such functions is speed. In `madd`'s case, AltiVec provides a dedicated multiply and add instruction which can be utilised by VVM. Despite its focus, because VVM tries to provide a more complete instruction set for ease of use, VVM also provides most of the functions from the `cmath` header file even though it is unlikely that any real VPU will provide such instructions.

This chapter discusses the VVM interface, including issues considered in its design.

6.1 Overview

Virtual Vector Machine (VVM) is an abstract VPU designed to be used as the VPU for a generic, vectorised, machine-vision library. VVM aims to represent desktop VPUs, such as AltiVec, MMX and 3DNow!. Being an abstract VPU, VVM has an idealised instruction set and constraints common to desktop VPUs. The characteristics of VVM are described in this chapter and possible implementations in the next.

VVM's functions are located in two different namespaces, `vvm` and `vvm::functional`, which are available when `vvm/vvm.h` and `vvm/vvm_functional.h` are included. The `vvm` namespace contains the bulk of the VVM specification. The VVM vector, functions and operators are all in this namespace. The `vvm::functional` namespace contains replacement comparison functors. Replacements are needed for Standard C++ comparison functors because Standard C++ library's comparison functors return a `bool`. The result of comparisons between vectors is a vector of booleans, which cannot be converted to a `bool`. Features found in the `vvm` namespace are not slated at replacing `std` functions.

VVM has three constraints common to desktop VPUs. These constraints are short vectors, fixed vector sizes and fast access only to aligned memory addresses. While the size of each VVM vector is fixed, unlike desktop VPUs, all VVM vectors have the same scalar count regardless of type. This constant scalar count is required for generic programming (Appendix C), and thus is required for the creation of a generic, vectorised library. Other features that support the creation of a generic, vectorised library include templated VVM vectors, traits, and consistent functions for both scalar and VVM vector operations. Because of the high cost of converting types in vector programs, VVM does not provide automatic type conversion; it supports only explicit type conversions.

Constant scalar count: Unlike real VPUs, such as AltiVec, MMX and 3DNow!, which have vectors that are all the same size, and therefore have different scalar counts for different vector types (Mot 1999, Mittal et al. 1997, Weiser 1996), all VVM vectors consist of the same number of scalars. The value of this fixed scalar count is not specified by VVM to allow more VPUs to be represented efficiently. A sensible value for the scalar count is the largest number of scalars in the VPU's vector, for example, 16 for AltiVec, 8 for MMX and 1 for the scalar processor. VVM has constant scalar count for ease of use and because it is required for generic programming. See Appendix C for more detail on why constant scalar count is necessary for generic programming. Constant scalar count also simplifies type conversions and makes VVM easier to understand. Examples of image processing operations that require type conversions are convolutions and equalisations.

Fast access to aligned memory addresses only: VVM can access only aligned memory addresses quickly. Like SSE and SSE2 (Adv 2002), VVM also provides slower

access to unaligned addresses. In AltiVec, unaligned memory can be accessed by loading aligned memory addresses and performing some transformations (Lai & McKerrow 2001, Ollmann 2001, App 2004*b*). Even though unaligned addresses are supported, it is still advisable to use only aligned addresses where possible. VVM provides access to unaligned addresses to allow implementations to use native VPU instructions. Using native VPU instructions is likely to provide better performance than using VVM instructions.

Templated VVM vectors: VVM's vectors are templated to support the easy creation of templated vector programs, which would be required for a generic, vectorised library.

Traits: Traits provide information about types at compile-time. An example of a traits template in C++ is `std::numeric_limits`, which provides information such as minimum and maximum values, and signedness of types. Traits are important when automating the generation of more complicated generic algorithms (Köthe 1999). Trait information can be used in the implementation of VVM itself. They are important for deriving the appropriate boolean type, since unlike scalar programs, vector programs have more than one boolean type. AltiVec, for example, has boolean vector types of differing sizes (Mot 1999). Promotion traits are important for writing templated code where promotion is necessary, such as in convolutions. Promotion traits were discussed in Section 4.4.2, and are a part of the CT library in this thesis.

Consistent functions where applicable: VVM has consistent functions for both scalar and VVM vector operations where applicable. Consistent functions make VVM easier to use and introduce fewer surprises. For example, `a + b` adds two VVM vectors and is undefined when the result overflows. Because VVM has consistent functions, in many situations, it is possible to write templated code that performs correctly when instantiated with a scalar or a VVM vector. For example, the following addition function works equally well if `T` is a scalar or a VVM vector.

```
template<typename T>
const T do_add(const T& a, const T& b) {
  return a + b;
}
```

Comparison operators in VVM however do not return a single boolean, because vector comparisons return a vector of booleans. In addition, unlike scalar code, `true` is converted to `~0` and not `1`. VVM maps `true` to `~0` because in vector programs, the results of comparisons are used as a mask. AltiVec comparison functions also return `~0` for `true` (Mot 1999). Because VVM comparisons return a vector of

booleans and `true` is converted to `~0`, template functions that use comparison operators cannot be instantiated for both scalars and VVM vectors. For example, the following program cannot be instantiated for VVM vectors.

```
template<typename T>
bool do_less(const T& a, const T& b) {
  return a < b;
}
```

Even if `do_less` returned `T`, scalar and VVM vector instantiations will return different values because `true` is converted to `~0` in VVM. To help scalar programs return VVM's value for `true` and `false`, VVM defines `VVM_TRUE` and `VVM_FALSE` which are the values that `true` and `false` are converted to in VVM respectively. `VVM_TRUE`, `VVM_FALSE` and other VVM preprocessor macros are discussed later in Section 6.17.

Operators are not only more intuitive, they also allow most Standard C++ functors to be used with VVM vectors.

Explicit type conversions only: Type conversions are discouraged in vector programs because type conversions have a pronounced effect on a vector program's performance. Because VPU vectors typically have different scalar counts, type conversions can change the maximum theoretical speedup. For example, in AltiVec, a `__vector unsigned char` has 16 scalars, and therefore has a theoretical 16-fold maximum speedup over `unsigned char`. A `__vector unsigned int` on the other hand only has 4 scalars and therefore AltiVec has only a 4-fold theoretical maximum speedup. Converting a vector's type can lead to changes in the theoretical maximum speedup.

6.2 Fundamental Scalar Types

VVM specifies a set of fundamental scalar types, which are listed in Table 6.1. Fundamental scalar types are types that vector-processor implementations must have supporting code for. It should be possible to create VVM vectors of all the fundamental scalar types and to manipulate them. While vector-processor implementations are not required to support all fundamental scalar types, it is recommended that they support as many as possible. The emulation layer in VVM must support all the fundamental scalar types. There are no fundamental VPU vector types because the VPU vector types that are available depends on the presence of real VPUs.

All of C++'s fundamental types (Inf 1998, basic.fundamental) except `bool`, are also VVM fundamental scalar types. Instead of `bool`, VVM has `bchar`, `bshort`, `bint` and so

Table 6.1 Fundamental VVM scalar types

Implementation Defined	Signed	Unsigned	Boolean
`char`	`signed char`	`unsigned char`	`bchar`
	`short int`	`unsigned short int`	`bshort`
	`int`	`unsigned int`	`bint`
	`long int`	`unsigned long int`	`blong`
	`float`		`bfloat`
	`double`		`bdouble`
	`long double`		`bldouble`
	`sint8`	`uint8`	`bint8`
	`sint16`	`uint16`	`bint16`
	`sint32`	`uint32`	`buint32`

on. This is to reduce the impact of conversion to and from the boolean types. Converting a VVM vector to its corresponding boolean vector type should not change the potential speedup.

In VVM, like in C++, 0 is converted to `false`, while any other value is converted to `true`. Unlike C++ however, converting `true` to an integer does not result in 1 (Inf 1998, conv.prom). In VVM, `true` is converted to a number where all the bits are 1s. This is because in vector processing, the result of comparisons is used as a bitmask to derive a new vector that contains the results of the comparison. A second reason for this behaviour is that this is the same behaviour as in AltiVec. To make it easier for scalar code to convert properly, VVM provides two preprocessors macros, `VVM_TRUE` and `VVM_FALSE` which are the integer values for `true` and `false` respectively. `VVM_TRUE`, `VVM_FALSE` and other VVM preprocessor macros are discussed later in Section 6.17.

Apart from additional booleans, VVM introduces `sint8`, `uint8`, `sint16`, `uint16`, `sint32`, `uint32` and corresponding booleans. These are integer types with a specific number of bits. These are building blocks for image pixels such as 8-bit grey-scale, and RGB colour pixels.

Pixel types such as 8-bit grey-scale, 16-bit grey-scale, and 24-bit colour were considered for fundamental scalar types for VVM. The grey-scale pixels in VVM would be simply a typedef of one of the existing fundamental scalar types. Colour pixels are more problematic because they are aggregate types. Colour pixel VVM types were avoided because colour pixels can be stored using different methods. VVM provides only the basic building blocks.

6.3 Traits

VVM provides a number of metafunctions for transforming types. These metafunctions are structured in a similar manner to the transformation type template metafunctions provided by Boost's type traits library.

```
namespace vvm {
  template<typename T> struct add_vector;
  template<typename T> struct remove_vector;
  template<typename T> struct to_bool;
  template<typename T> struct to_integer;
  template<typename T> struct to_float;
}
```

In addition, VVM provides information on the three different kinds of data that it uses: scalars, VPU vectors and VVM vectors. These three data types are referred to as `scalar`, `vpvector` and `vector` in VVM respectively. To get information on these three kinds of data, VVM uses three templates: `vvm::scalar_traits`, `vvm::vpvector_traits` and `vvm::vector_traits`. The `scalar_traits`, `vpvector_traits` and `vector_traits` trait templates take scalars, VPU vectors and VVM vectors as their template parameters respectively. In addition, they all have an `is_specialised` field. A value of `true` in this field indicates that the information provided by the trait template is valid.

```
namespace vvm {
  template<typename scalarT> struct scalar_traits;
  template<typename vpvectorT> struct vpvector_traits;
  template<typename vectorT> struct vector_traits;
}
```

Details on the type transformations and information provided by these three traits are presented in this section.

6.3.1 Type transformations

VVM provides two metafunctions, `add_vector` and `remove_vector`, respectively to allow the user to derive the appropriate VVM vector type for a scalar type and vice versa. In addition, it provides `to_bool`, `to_integer` and `to_float`, which respectively allow the user to derive the boolean, integer or float scalar type from a scalar type. When there is no appropriate answer, these templates should return `ct::null_type` instead of preventing compilation. The type transformations supported by VVM are detailed in Table 6.2.

These type transformation metafunctions are the preferred method of deriving the appropriate scalar or VVM vector type, because these type transformations are generally

Table 6.2 Template metafunctions for transforming types

Expression	Return Type	Notes
`add_vector<T>::type`	Type	Returns `vvm::vector<T>`
`add_vector< ct::tuple<TL> >::type`	Type	Returns `ct::tuple<ct::apply< TL, add_vector>::type>`
`remove_vector< vvm::vector<T> >::type`	Type	Returns `T`
`remove_vector< ct::tuple<TL> >::type`	Type	Returns `ct::tuple<ct::apply< TL, remove_vector>::type>`
`to_bool<T>::type`	Type	Returns `scalar_traits<T>:: bool_type`
`to_bool< ct::tuple<TL> >::type`	Type	Returns `ct::tuple<ct::apply< TL, to_bool>::type>`
`to_integer<T>::type`	Type	Returns `scalar_traits<T>:: integer_type`
`to_integer< ct::tuple<TL> >::type`	Type	Returns `ct::tuple<ct::apply< TL, to_integer>::type>`
`to_float<T>::type`	Type	Returns `scalar_traits<T>:: float_type`
`to_float< ct::tuple<TL> >::type`	Type	Returns `ct::tuple<ct::apply< TL, to_float>::type>`

smarter. For example, for `const` scalar types `add_vector` should return a valid VVM vector type while `vvm::vector<T>` might fail to compile because it is not in this specification. In addition, because `add_vector` will return `ct::null_type` when there is no appropriate type, it can be used to make compile-time decisions. Furthermore, these metafunctions make it easy to convert typelists of scalars to typelists of VVM vectors and vice versa. For example, it is possible to use the `ct::apply` template metafunction with `add_vector` as follows:

```
// TL is a typelist of scalars
typedef ct::apply<TL, vvm::add_vector>::type vectors_tl;
```

These type transformation metafunctions also convert `ct::tuples` correctly. In general, applying `OP<ct::tuple<TL>>::type` results in `ct::tuple<typename ct::apply<TL, OP>::type>`. Because VVM's type transformation metafunctions support `ct::tuples`, which are used to represent multi-channel pixels in the generic, vectorised, machine-vision library developed as part of this thesis, template functions can be written that instantiate correctly for both single-channel and multi-channel pixels. For example, the following `add` template function, which adds two VVM vectors, instantiates correctly if `scalarT` is a scalar or a `ct::tuple`.

```
template<typename scalarT>
```

```
const typename vvm::add_vector<scalarT>::type
  add(
    const typename vvm::add_vector<scalarT>::type a,
    const typename vvm::add_vector<scalarT>::type b) {
      return a + b;
    }
```

6.3.2 vvm::scalar_traits

The `vvm::scalar_traits` trait template provides information about VVM fundamental scalar types. It must be specialised for all the VVM fundamental scalar types listed in Table 6.1.

```
namespace vvm {
  // Replace vpvectorT and boolT with the appropriate types
  template<typename scalarT> struct scalar_traits {
    static const bool is_specialised = false;
    typedef boolT bool_type;
    typedef integerT integer_type;
    typedef floatT float_type;
    typedef vpvectorT vpvector_type;
 };
}
```

The `vvm::scalar_traits` trait template provides the corresponding boolean scalar, integer scalar, float scalar, and VPU vector types for a VVM fundamental scalar type.

bool_type: This refers to the corresponding boolean scalar type. This should be set to the appropriate VVM scalar boolean type. This is used to derive the appropriate boolean VPU or VVM vector type for operations like comparisons.

integer_type: This refers to an integer scalar type, preferably of the same size. If the original scalar type is an integer, then `integer_type` is equivalent to the original scalar type.

float_type: This refers a floating-point scalar type, preferably of the same size. If the original scalar type is a float, then `float_type` is equivalent to the original scalar type.

vpvector_type: This refers to the corresponding VPU vector type.

The `vpvector_type` type is used to determine the appropriate VPU vector type from the scalar. This vector type should have scalars that are the same size, the

same kind (either integer, boolean or float) and have the same signedness as the scalar, regardless of what its actual name is. For example, AltiVec has a `__vector signed int` that has scalars 4 bytes in length. If the compiler's `int` is only 2 bytes long, it should not be mapped to `__vector signed int`. Instead it should be mapped to `__vector signed short` which has scalars 2 bytes long.

Whether `char` is signed or unsigned is implementation dependant. The type returned by `char`'s `vpvector_type` should have the same signedness as `char`.

A VVM scalar type is either an integer or a float type, because `bool` is not a VVM fundamental scalar type. Thus either `integer_type` or `float_type` is the same type as the original type. The type returned by `integer_type` should never be identical to `float_type`.

6.3.3 vvm::vpvector_traits

The `vvm::vpvector_traits` trait template provides information on VPU vector types. It should be specialised for all VPU vector types that are referred to by specialisations of `vvm::scalar_traits`. VVM implementations do not need to provide `vvm::vpvector_traits` for VPU vector types that are not referenced by `vvm::scalar_traits`, because they are not be used by VVM.

```
namespace vvm {
  template<typename vpvectorT> struct vpvector_traits
  {
    static const bool is_specialised = false;
    typedef scalarT scalar_type;
    typedef vpvectorBoolT bool_type;
    enum { scalar_count = scalarCount };
  };
}
```

The `vvm::vpvector_traits` trait template provides the corresponding scalar type, boolean VPU vector type, and the number of scalars in the VPU vector type.

scalar_type: This refers to the corresponding scalar type.

bool_type: This refers to the corresponding boolean VPU vector type. This should be the same as `scalar_traits<scalar_traits<scalar_type>::bool_type>::vpvector_type`.

scalar_count: This refers to the number of scalars in the VPU vector. This can be calculated by dividing the size of the VPU vector by the size of the corresponding scalar,

since the corresponding scalar's size should be the same size as the scalars within the VPU vector.

This information is important when determining how many VPU vectors are contained within a VVM vector. The number of VPU vectors varies between different VVM vectors because the scalar count is fixed.

6.3.4 vvm::vector_traits

The `vvm::vector_traits` trait template provides information about VVM vectors. While `vvm::vectors` could have kept such information themselves, as much as possible, VVM tries to present `vvm::vectors` as a basic types to facilitate compiler optimisations. Moreover, using `vvm::vector_traits` to query `vvm::vector` characteristics is consistent with the behaviour of `vvm::scalar_traits` and `vvm::vector_traits`.

The `vvm::vector_traits` trait template should be specialised for all the standard `vvm::vectors`.

```
namespace vvm {
  template<typename vectorT> struct vector_traits {
    static const bool is_specialised = false;
    typedef scalarT scalar_type;
    typedef vpvectorT vpvector_type;
    typedef vector<scalar_traits<scalar_type>::bool_type>
      bool_type;
    typedef vector<scalar_traits<scalar_type>::integer_type>
      integer_type;
    typedef vector<scalar_traits<scalar_type>::float_type>
      float_type;
    enum { scalar_count, vpvector_count, step }
  };
}
```

scalar_type: This returns the type of the scalar that is contained in `vectorT`.

vpvector_type: This returns the type of the VPU vector that is contained in `vectorT`.

bool_type: This returns a `vvm::vector` that contains `scalar_traits<scalar_type>::bool_type` scalars.

integer_type: This returns a `vvm::vector` that contains `scalar_traits<scalar_type>::integer_type` scalars.

float_type: This returns a `vvm::vector` that contains `scalar_traits<scalar_type>::float_type` scalars.

scalar_count: This constant is the number of scalars in `vectorT`. Because VVM has constant scalar count, this value is actually constant for all `vvm::vectors`, and equal to `VVM_SCALAR_COUNT`.

vpvector_count: This constant is the number of VPU vectors that are in `vectorT`.

step: This is equal to the number of scalars in an individual `vpvector_count`. It can also be obtained through `vpvector_traits<vpvector_type>::scalar_count`.

6.4 The Vector

The design of `vvm::vectors` is discussed in this section. Access to individual scalars and VPU vectors within a `vvm::vector` is designed to be easy and efficient. VVM's vectors are designed to be simple, aiming to be more like intrinsic types like `int` rather than full-blown objects. As a result, `vvm::vectors` are not designed for inheritance, and thus have no virtual destructors. In addition, all data are kept on the stack and thus user-defined copy constructors and assignment operators are not necessary. Not providing user-defined copy constructors and assignment operator allows the compiler to more easily detect that `vvm::vectors` can be enregistered. Enregistering allows the `vvm::vector` to be loaded into a register and is important for reducing overheads. In Apple GCC 3.1 20021003, for example, it was not possible to avoid significant overheads when the copy constructor and assignment operator were defined.

A cost-effective and easy method of accessing individual scalars helps eliminate unsightly code such as `((unsigned char *)(&v1))[i]` from Lai & McKerrow (2001), which are a source of potential coding errors. While the cost of accessing individual scalars might as first glance not seem as important as the cost of accessing individual VPU vectors, it is important for the emulation functions of the VVM. Since no current VPU provides support for all VVM functions, it is likely that the emulation layer is required even when a real VPU is available. Therefore accessing scalars should be as fast as possible.

Since VVM vector-processor implementations need to access individual VPU vectors when performing operations, accessing individual VPU vectors should incur no significant overheads. If there are significant overhead involved in accessing individual VPU vectors, then the VVM implementation will have significant overheads as well.

```
namespace vvm {
  template<typename scalarT> class vector {
  private:
```

```
  typedef scalarT scalar_type;
  typedef typename scalar_traits<scalar_type>::vpvector_type
    vpvector_type;
public: // Constructors
  vector();
  vector(const vector& rhs);
  vector(const scalar_type& scalar);
public: // Operators
  vector& operator=(const vector& rhs);
  vector& operator=(const scalar_type& scalar);
public: // Access to data
  scalar_type& scalar(const int i);
  const scalar_type& scalar(const int i) const;
  vpvector_type& vpvector(const int i);
  const vpvector_type& vpvector(const int i) const;
};
typedef vchar vector<char>;
typedef vuchar vector<unsigned char>;
typedef vschar vector<signed char>;
typedef vshort vector<short>;
typedef vushort vector<unsigned short>;
typedef vint vector<int>;
typedef vuint vector<unsigned int>;
typedef vlong vector<long>
typedef vulong vector<unsigned long>;
typedef vfloat vector<float>;
typedef vdouble vector<double>;
typedef vbchar vector<bool_char>;
typedef vbshort vector<bool_short>;
typedef vbint vector<bool_int>;
typedef vblong vector<bool_long>;
typedef vbfloat vector<bool_float>;
typedef vbdouble vector<bool_double>;
typedef vbldouble vector<bool_ldouble>;
typedef vsint8 vector<sint8>;
typedef vuint8 vector<uint8>;
typedef vbint8 vector<bint8>;
typedef vsint16 vector<sint16>;
typedef vuint16 vector<uint16>;
```

```
  typedef vbint16 vector<bint16>;
  typedef vsint32 vector<sint32>;
  typedef vuint32 vector<uint32>;
  typedef vbint32 vector<bint32>;
}
```

The definition above is the minimum specification that VVM implementations must provide. VVM implementations can add extra functions and member variables that are needed to perform efficiently. Ideally, these extras should not be usable by the user. There is no requirement for the VVM vector to be a `class`. For example, a `union` can be used instead of a `class` to represent a `vvm::vector`; using a `union` should provide easy access to both scalar and VPU vector elements. The VVM specification leaves these implementation details open to enable VVM implementations to select the most efficient implementation method for their chosen environments.

For ease of use, `vvm::vectors` of VVM fundamental scalar types have a shorter name in addition to the full template name. For example, `vsint8` is a typedef for `vector<sint8>`.

vector(): This is the default constructor. For efficiency, the behaviour of the default constructor is to do nothing. This behaviour is consistent with basic types provided by C++ such as `int` and `char`. Since the default constructor does nothing, VVM implementations can omit this function.

vector(const scalar_type& scalar): This function sets all scalars in the `vvm::vector` to `scalar`. While VPUs typically provide fill functions, not all of the them are compatible with VVM's definition. In AltiVec for example, the value supplied to the fill function must be a literal; it must be set at compile-time. VVM does not have such restrictions because it tries to be easy to use and to provide an idealised instruction set. For VVM to have such a restriction seems excessive.

operator=(const scalar_type& scalar): This function sets all scalars in the `vvm::vector` to `scalar`. It performs the same operation as `vector(const scalar_type& scalar)`.

scalar(const int), vpvector(const int): These two functions provide read/write access to the scalars and VPU vectors contained in the `vvm::vector` respectively. Originally, scalar and VPU vector access was to be provided via a `.scalar[]` public field and a `.vpvector[]` public field respectively. For this scenario to work, either the VVM implementation implements the `vvm::vector` as a `union`, or it uses inner objects to provide `.scalar[]` and `.vpvector[]` access as illustrated in the code below:

```
template<typename scalarT> class vector {
private:
```

```
    struct _scalar_access {
      scalarT& operator[](int i);
      const scalarT& operator[](int i) const;
    };
    struct _vpvector_access {
      typename scalar_traits<scalarT>::vpvector_type&
        operator[](int i);
      const typename scalar_trait<scalarT>::vpvector_type&
        operator[](int i) const;
    };
  public:
    _scalar_access scalar;
    _vpvector_access vpvector;
  };
```

This alternative will be inefficient, because it increases the size of the `vvm::vector`; inner objects require a reference to the `vvm::vector`'s data. Furthermore, even if the inner objects had no member variables, according to Vandevoorde & Josuttis (2003), Chapter 16.2 The Empty Base Class Optimisation (EBCO), even empty classes might still not be of zero sizes. Larger `vvm::vectors` means less cache available for other important data, and longer times spent copying data.

Using the functions allows the VVM implementation more leeway in their implementation, especially since it should be easy for compilers to remove the cost of the function call via inlining, thereby removing any overheads.

6.5 Constants

As discussed in Section 3.5.1, it can be faster to generate constants using VPU instructions instead of loading them from memory. However, in order for a C++ library to generate such code without using any extra external tools, the library needs to know the constants that the user wants to generate at compile-time. Passing constants in via a parameter in a function does not give the library this information. Therefore, in the interest of supporting such constants, VVM provides the following function:

```
namespace vvm {
  template<typename scalarT, int value>
    vector<scalarT> constant();
}
```

While this feature is good for performance, the author feels that it makes the library less consistent since there are now two ways of generating constants, with subtle differences. It would be better, if VVM was able to generate instructions when the fill constructor is used. If VVM was implemented as a preprocessor, then it is likely that this is not required.

PixelGlow's MacSTL library (Pix 2003) provides such a feature. The C++ compiler is used to calculate the appropriate instructions for generating the required constant. Note that because the C++ compiler is not a fast interpreter, the compile-time might be drastically increased.

6.6 Memory functions

VVM provides functions to load and write `vvm::vectors` from and to aligned memory locations, to allocate and deallocate memory for `vvm::vectors` and to prefetch data.

```
namespace vvm {
  // Load and Store vectors
  template<typename scalarT>
    vector<scalarT> load(const scalarT* scalar_addr);
  template<typename scalarT>
    void store(const vector<scalarT>& v,
               scalarT* scalar_addr);
  template<typename scalarT>
    vector<scalarT> load_unaligned(const scalarT* scalar_addr);
  template<typename scalarT>
    void store_unaligned(const vector<scalarT>& v,
                         scalarT* scalar_addr);
  // New & Delete operators
  template<typename scalarT>
    vector<scalarT> *operator new(std::size_t count)
      throw(std::bad_alloc);
  template<typename scalarT>
    vector<scalarT> *operator new[](std::size_t count)
      throw(std::bad_alloc);
  template<typename scalarT>
    void operator delete(vector<scalarT> *addr) throw();
  template<typename scalarT>
    void operator delete[](vector<scalarT> *addr) throw();
  // Alignment
  template<typename scalarT>
```

```
    offset_t uoffset(scalarT* const p);
  template<typename scalarT>
    scalarT* align_prev(scalarT* p, const offset_t offset = 0);
  template<typename scalarT>
    scalarT* align_next(scalarT* p, const offset_t offset = 0);
  // Usage Types should be
  // read, read_transient, write, write_transient
  template<int channel, typename usage, typename scalarT>
    void prefetch(const vector<scalarT>* addr, const int count);
  template<int channel> void stop_prefetch();
    void stop_all_prefetch();
}
```

6.6.1 Loads and stores

```
template<typename scalarT>
  vector<scalarT> load(const scalarT* scalar_addr);
template<typename scalarT>
  void store(const vector<scalarT>& v,
             scalarT* scalar_addr);
template<typename scalarT>
  vector<scalarT> load_unaligned(const scalarT* scalar_addr);
template<typename scalarT>
  void store_unaligned(const vector<scalarT>& v,
                       scalarT* scalar_addr);
```

The `load` and `store` functions allow the user to load and store a `vvm::vector` from and to a scalar pointer. If the user has a `vvm::vector` pointer, the user can use the dereference `operator*` instead.

The `load` and `store` functions only work correctly for aligned memory locations. Loading from or storing to unaligned memory locations is undefined. To access unaligned memory locations, use `load_unaligned` and `store_unaligned` instead.

6.6.2 Alignment

```
template<typename scalarT>
  offset_t uoffset(scalarT* const p);
template<typename scalarT>
  scalarT* align_prev(scalarT* p, const offset_t offset = 0);
template<typename scalarT>
```

```
scalarT* align_next(scalarT* p, const offset_t offset = 0);
```

The alignment functions return how many bytes a pointer is off alignment by, and aligned pointers nearest to a specified pointer.

uoffset: This function returns the difference in bytes between `p` and the largest aligned pointer smaller than `p`. For aligned pointers, this function returns `0`.

align_prev: This function returns the largest aligned pointer that is less than or equal to `p + offset`. For aligned pointers, this function will return `p`.

align_next: This function returns the smallest aligned pointer that is greater than or equal to `p + offset`. For aligned pointers, this function will return `p`.

6.6.3 Allocation and deallocation

```
template<typename scalarT>
  vector<scalarT> *operator new(std::size_t count)
    throw(std::bad_alloc);
template<typename scalarT>
  vector<scalarT> *operator new[](std::size_t count)
     throw(std::bad_alloc);
template<typename scalarT>
  void operator delete(vector<scalarT> *addr) throw();
template<typename scalarT>
  void operator delete[](vector<scalarT> *addr) throw();
```

VVM implementations should ensure that the `new` and `delete` operators correctly with `vvm::vectors`. Fortunately, the compiler usually does this already, and therefore these functions are unlikely to need implementation. However, the onus is on the VVM implementation to make sure that the operators work as expected.

6.6.4 Prefetching

```
// Channels start from 0
// Usage Types should be
// read, read_transient, write, write_transient
template<int channel, typename usage, typename scalarT>
  void prefetch(const vector<scalarT>* addr, const int count);
template<int channel> void stop_prefetch();
void stop_all_prefetch();
```

VPUs require more data to be loaded before they can execute their instructions because each instruction operates on more data. This is a problem since, as discussed in Section 3.4, memory is usually the most significant bottleneck in a vector program. One of the methods for countering this is to prefetch the data so that it is in the caches before the VPU requires them.

Since such prefetch instructions are important to attaining and sustaining high speedup, VVM will need instructions for prefetching. For systems without such prefetch instructions, these prefetch instructions will do nothing, and ideally be optimised to nothing.

prefetch(const vector<scalarT>* addr, const int count): This function starts prefetching data. The prefetch instruction is designed to allow:

1. Allow easy specification of start address.

   The start address is needed to specify where the prefetch should begin. If the start address is not valid for the VPU, VVM implementations can start prefetching at any address close to the specified address instead.

2. Allow easy specification of number of bytes to prefetch.

   This specifies the number of bytes that should be prefetched from the start address. If the number of bytes to prefetch is not valid for the VPU, VVM implementation can fetch any number of bytes that is close to the number of bytes specified.

3. Allow easy specification of which channel to use to prefetch.

   For prefetching data, while VVM supports an unspecified number of channels, AltiVec only has four different channels. Thus in AltiVec, only the first four channels would have any effect. Other channels should result in an empty function.

   The channel to use to prefetch is determined by a template parameter type. This is allow implementations to easily generate no code for channels that do not exist. In addition, the channel count is specified by a template parameter because the AltiVec prefetch instruction requires the channel to be specified as a literal.

   Users should use prefetch channels from 0. This allows the user to actually use more of the channels provided by the real VPU. In addition, it allows the vector-processor implementation to number the channels internally in a manner that interferes least with the rest of the system. In MacOS X, for example, when running on an AltiVec-enabled microprocessor, the operating system uses AltiVec data stream channels starting from 3, then 2 and so on; the user should therefore start from AltiVec data stream channel 0 (App 2004*a*).

4. Allow easy specification of usage pattern.

   The usage pattern specifies what the data being loaded will be used for. These usage patterns are only hints. VVM specifies four different usage patterns, which are the same four usage patterns available in AltiVec:

   read_only: Use this when the program expects to only read from the address specified and will probably need to use it more than once.

   read_write: Use this when the program expected to read and write from the address specified and will probably need to use it more than once.

   read_only_transient: Use this when the program expects to only read the data once.

   read_write_transient: Use this when the program expects to read and write the data only once.

   Transient usage pattern indicates that the data would only be used once. In AltiVec, transient data are flushed straight to main memory instead of to the L2 cache (App 2004*a*). This frees more space in the caches for other programs.

In AltiVec, the prefetch instruction requires bitwise arithmetic. For this to be calculated at run-time is a waste of time if the values are known beforehand. The design therefore suggests that if possible, the instruction should be precomputed at compile-time.

The positions where the prefetching will start and stop is not exact, because in general, it is not really important. The idea is to get as much in as possible. As long as not too much is missed, or too much extra is loaded, there should be little impact.

stop_all_prefetch(): This function stops all prefetching. Whether these include channels that are not used by the current application is implementation-defined. Since it could stop channels that are not used by the application, it is advisable to not use this function frequently.

6.7 Input/Output functions

The input/output functions provided by VVM are quite basic. The `operator<<` function is expected to be useful when debugging programs.

```
namespace vvm {
  template<typename scalarT>
    std::ostream&
    operator<<(std::ostream& os, const vector<scalarT> &v);
  template<typename scalarT>
```

```
    std::istream&
    operator>>(std::istream& is, vector<scalarT> &v);
}
```

operator< <: The `operator<<` function should print out all scalars in a `vvm::vector` delimited by spaces. For `char` types, the integer value of the `char` should be printed, instead of the ASCII character. The `operator<<` function should not print any extra characters, like brackets before and after the `vvm::vector`, in order to allow `operator>>` to easily read the output back into a `vvm::vector`.

operator> >: The `operator>>` function should read in all the scalars using the `>>` operator.

6.8 Arithmetic functions

VVM will support all arithmetic operators, including a saturated addition and subtraction. Also included are `madd` and `nmsub` functions. From a functionality point of view, there is no reason for these functions to exist, since they can be readily computed using other functions. However, most VPUs have these functions, and compute these functions faster than if the composite functions were used instead.

```
namespace vvm {
  template<typename scalarT>
    const vector<scalarT> adds(const vector<scalarT>& lhs,
                               const vector<scalarT>& rhs);
  template<typename scalarT>
    const vector<scalarT> subs(const vector<scalarT>& lhs,
                               const vector<scalarT>& rhs);
  template<typename scalarT>
    const vector<scalarT> operator+(const vector<scalarT>& lhs,
                                    const vector<scalarT>& rhs);
  template<typename scalarT>
    const vector<scalarT> operator-(const vector<scalarT>& lhs,
                                    const vector<scalarT>& rhs);
  template<typename scalarT>
    const vector<scalarT> operator*(const vector<scalarT>& lhs,
                                    const vector<scalarT>& rhs);
  template<typename scalarT>
    const vector<scalarT> operator/(const vector<scalarT>& lhs,
                                    const vector<scalarT>& rhs);
```

```
template<typename scalarT>
  const vector<scalarT> operator%(const vector<scalarT>& lhs,
                                  const vector<scalarT>& rhs);

template<typename scalarT>
  vector<scalarT>& operator++(vector<scalarT>& rhs);
template<typename scalarT>
  vector<scalarT>  operator++(vector<scalarT>& lhs, int);
template<typename scalarT>
  vector<scalarT>& operator--(vector<scalarT>& rhs);
template<typename scalarT>
  vector<scalarT>  operator--(vector<scalarT>& lhs, int);

template<typename scalarT>
  vector<scalarT>& operator+=(vector<scalarT>& lhs,
                              const vector<scalarT>& rhs);
template<typename scalarT>
  vector<scalarT>& operator-=(vector<scalarT>& lhs,
                              const vector<scalarT>& rhs);
template<typename scalarT>
  vector<scalarT>& operator*=(vector<scalarT>& lhs,
                              const vector<scalarT>& rhs);
template<typename scalarT>
  vector<scalarT>& operator/=(vector<scalarT>& lhs,
                              const vector<scalarT>& rhs);
template<typename scalarT>
  vector<scalarT>& operator%=(vector<scalarT>& lhs,
                              const vector<scalarT>& rhs);

template<typename scalarT>
  const vector<scalarT>  operator-(const vector<scalarT>& lhs);
template<typename scalarT>
  const vector<scalarT>& operator+(const vector<scalarT>& rhs);
template<typename scalarT>
  const vector<scalarT> madd(const vector<scalarT>& v1,
                             const vector<scalarT>& v2,
                             const vector<scalarT>& v3);
template<typename scalarT>
  const vector<scalarT> nmsub(const vector<scalarT>& v1,
```

```
                                    const vector<scalarT>& v2,
                                    const vector<scalarT>& v3);
}
```

The addition and subtraction operators have undefined behaviour when the result overflows. The `adds` and `subs` functions perform saturated addition and subtraction respectively.

Originally, addition and subtraction operators were saturated, because saturated arithmetic is more common in image processing. However, because not all data can be processed efficiently using the vector processor, both scalar and `vvm::vector` operations are required. Thus, scalar and `vvm::vector` operations should have the same semantics. This means that either `vvm::vector` addition and subtraction should have undefined behaviour when the result overflows, or scalar addition and subtraction should be saturated.

Undefined behaviour on result overflows was chosen because it leads to more predictable behaviour for `vvm::vectors` and its operations. Making scalar addition and subtraction saturated requires representing scalar types as another type. This can lead to situations where the scalars within a `vvm::vector` are of a different type from the template parameter of `vvm::vector`, which would be unexpected. For example, the scalar elements of `vvm::vector<int>` would not be of type `int`.

6.9 cmath functions

Apart from basic arithmetic functions, VVM also provides functions from the Standard C++ `cmath` header file. Functions from the `cmath` library are also important to machine-vision, since some commonly used image-processing routines require operations from this library. Having these functions enables vector-processor implementations to perform these operations using the VPU where applicable. Even if all `cmath` functions were all processed using the scalar processor, these functions still simplify code and allow templated code to also use the `cmath` functions as well. VVM tries to provide all the `cmath` functions, with the same operations so that they perform in the same manner as scalar code.

```
namespace vvm {
  // The following should be defined for all integer types
  // Replace vint with vchar, vshort etc...
  vint abs(const vint& v);

  // The following should be defined for all real types
  // Replace vfloat with vdouble, vldouble
  vfloat abs(const vfloat& v);
```

```
  vfloat acos(const vfloat& v);
  vfloat asin(const vfloat& v);
  vfloat atan(const vfloat& v);
  vfloat atan2(const vfloat& v1, const vfloat& v2);
  vfloat ceil(const vfloat& v);
  vfloat cos(const vfloat& v);
  vfloat cosh(const vfloat& v);
  vfloat exp(const vfloat& v);
  vfloat fabs(const vfloat& v);
  vfloat floor(const vfloat& v);
  vfloat frexp(const vfloat& v,
    vector_traits<vfloat>::integer_type* exp);
  vfloat ldexp(const vfloat& v,
    const vector_traits<vfloat>::integer_type& exp);
  vfloat log(const vfloat& v);
  vfloat log10(const vfloat& v);
  vfloat modf(const vfloat& v, vfloat* iptr);
  vfloat pow(const vfloat& v, const vfloat& exp);
  vfloat pow(const vfloat& v,
    const vector_traits<vfloat>::integer_type& exp);
  vfloat sin(const vfloat& v);
  vfloat sinh(const vfloat& v);
  vfloat sqrt(const vfloat& v);
  vfloat tan(const vfloat& v);
  vfloat tanh(const vfloat& v);
}
```

In the interests of being succinct, variations of each function is not listed. For example, `abs` should be defined for all signed and unsigned integer `vvm::vectors`, excluding boolean `vvm::vectors`. The `acos`, `asin`, `atan`, and so on functions should be defined for all floating-point `vvm::vectors`.

A VVM implementation might just provide these functions through the emulation layer. On Mac OS X, there is a VecLib framework which actually contains such functions, but implemented using AltiVec were appropriate. Other libraries, like Itanium Vector Math Library (Hew 2003), also provide such functions for other vector-processor technologies.

6.10 Bitwise functions

Bitwise operators only apply to integer and boolean `vvm::vector` types. Trying to use bitwise operators on floating-point types should result in a compilation error.

```
namespace vvm
{
  // Applies only to integer and boolean vvm::vector types
  template<typename scalarT>
    vector<scalarT> operator~(const vector<scalarT>& rhs);
  template<typename scalarT>
    vector<scalarT> operator^(const vector<scalarT>& lhs,
                              const vector<scalarT>& rhs);
  template<typename scalarT>
    vector<scalarT>& operator^=(vector<scalarT>& lhs,
                                const vector<scalarT>& rhs);
  template<typename scalarT>
    vector<scalarT> operator&(const vector<scalarT>& lhs,
                              const vector<scalarT>& rhs);
  template<typename scalarT>
    vector<scalarT>& operator&=(vector<scalarT>& lhs,
                                const vector<scalarT>& rhs);
  template<typename scalarT>
    vector<scalarT> operator|(const vector<scalarT>& lhs,
                              const vector<scalarT>& rhs);
  template<typename scalarT>
    vector<scalarT>& operator|=(vector<scalarT>& lhs,
                                const vector<scalarT>& rhs);
  template<typename scalarT>
    vector<scalarT> operator>>(const vector<scalarT>& lhs,
                               const unsigned short shift);
  template<typename scalarT>
    vector<scalarT> operator>>(const vector<scalarT>& lhs,
                               const vector<scalarT>& rhs);
  template<typename scalarT>
    vector<scalarT> operator<<(const vector<scalarT>& lhs,
                               const unsigned char shift);
  template<typename scalarT>
    vector<scalarT> operator<<(const vector<scalarT>& lhs,
                               const vector<scalarT>& rhs);
```

```
  template<typename scalarT>
    vector<scalarT>& operator>>=(vector<scalarT>& lhs,
                                 const unsigned char shift);
  template<typename scalarT>
    vector<scalarT>& operator>>=(vector<scalarT>& lhs,
                                 const vector<scalarT>& rhs);
  template<typename scalarT>
    vector<scalarT>& operator<<=(vector<scalarT>& lhs,
                                 const unsigned char shift);
  template<typename scalarT>
    vector<scalarT>& operator<<=(vector<scalarT>& lhs,
                                 const vector<scalarT>& rhs);
}
```

There are two versions of the left shift and right shift operations The first versions shifts the entire `vvm::vector` by an `unsigned char` number of bits, while the second version shifts the scalars in a `vvm::vectors` by the corresponding scalars in a second `vvm::vector`.

The first version's right-hand side is an `unsigned char`. This shift function shifts the entire `vvm::vector`, as a single integer, by the specified number of bits. The amount shifted is specified using a `unsigned char` because it is unlikely that the user needs to shift more than 256 bits.

The second version's right-hand side is a `vvm::vector`. In this case, the scalars are shifted by the corresponding scalar in the right-hand `vvm::vector`. Because different `vvm::vectors` consist of different VPU vectors, which are likely to consist of different number of scalars, the right-hand `vvm::vector` is not represented by `vector<unsigned char>`. It is instead `vector<scalarT>` to allow easier mapping to VPU functions.

Shift operations are only valid for positive values. When asked to shift negative values, the behaviours of these shift operators is undefined. This behaviour is consistent with Standard C++ (Inf 1998, Section 5.8). The result of shifting by too many bits is also undefined.

6.11 Logical functions

Logical operators apply to all `vvm::vectors`, including floating-point types. Any non-zero value is assumed to be `true`. This behaviour is consistent with the C++ Standard (Inf 1998, Section 4.12).

```
namespace vvm
{
  template<typename scalarT>
```

```
    vector<scalar_traits<scalarT>::bool_type>
      operator!(const vector<scalarT>& rhs);
  template<typename scalarT>
    vector<scalar_traits<scalarT>::bool_type>
      operator&&(const vector<scalarT>& lhs,
                 const vector<scalarT>& rhs);
  template<typename scalarT>
    vector<scalar_traits<scalarT>::bool_type>
      operator||(const vector<scalarT>& lhs,
                 const vector<scalarT>& rhs);
}
```

6.12 Comparison functions

The return value of comparison operators in VVM is different from the comparison operators in C++. Instead of returning a `bool`, comparison operators in VVM return a boolean `vvm::vector`, because the comparison results of two vectors cannot be expressed as a single boolean. The exact boolean `vvm::vector` is derived by querying `scalar_traits` for the appropriate boolean type. For more information on obtaining the corresponding boolean type through `scalar_traits`, see Section 6.3.

```
namespace vvm {
  template<typename scalarT>
    vector<scalar_traits<scalarT>::bool_type>
      operator>(const vector<scalarT>& lhs,
                const vector<scalarT>& rhs);
  template<typename scalarT>
    vector<scalar_traits<scalarT>::bool_type>
      operator<(const vector<scalarT>& lhs,
                const vector<scalarT>& rhs);
  template<typename scalarT>
    vector<scalar_traits<scalarT>::bool_type>
      operator>=(const vector<scalarT>& lhs,
                 const vector<scalarT>& rhs);
  template<typename scalarT>
    vector<scalar_traits<scalarT>::bool_type>
      operator<=(const vector<scalarT>& lhs,
                 const vector<scalarT>& rhs);
  template<typename scalarT>
```

```
    vector<scalar_traits<scalarT>::bool_type>
      operator==(const vector<scalarT>& lhs,
                 const vector<scalarT>& rhs);
    template<typename scalarT>
    vector<scalar_traits<scalarT>::bool_type>
      operator!=(const vector<scalarT>& lhs,
                 const vector<scalarT>& rhs);
}
```

6.13 Predicates

The VVM predicates presented in this section come from AltiVec. Unlike comparisons, predicates return a single `bool`. Predicates of the "any" kind OR the results of the comparison while predicates of the "all" kind AND. Predicates of the "any" kind return `true` if any of the values in the two input `vvm::vectors` match the comparison operation specified. Predicates of the "all" kind return `true` only if all the values in the two input `vvm::vectors` match the comparison operation specified.

The `eq`, `ge`, `gt`, `le` and `lt` suffixes stand for "equal to", "greater than or equal to", "greater than", "less than or equal to" and "less than" respectively. The `ne`, `nge`, `ngt`, `nle` and `nlt` suffixes stand for "not equal to", "not greater than or equal to", "not greater than", "not less than or equal to" and "not less than" respectively.

```
namespace vvm {
  // All predicates
  template<typename scalarT>
    bool all_eq(const vector<scalarT>& lhs,
                const vector<scalarT>& rhs);
  template<typename scalarT>
    bool all_ge(const vector<scalarT>& lhs,
                const vector<scalarT>& rhs);
  template<typename scalarT>
    bool all_gt(const vector<scalarT>& lhs,
                const vector<scalarT>& rhs);
  template<typename scalarT>
    bool all_le(const vector<scalarT>& lhs,
                const vector<scalarT>& rhs);
  template<typename scalarT>
    bool all_lt(const vector<scalarT>& lhs,
                const vector<scalarT>& rhs);
```

```
template<typename scalarT>
  bool all_ne(const vector<scalarT>& lhs,
              const vector<scalarT>& rhs);
template<typename scalarT>
  bool all_nge(const vector<scalarT>& lhs,
               const vector<scalarT>& rhs);
template<typename scalarT>
  bool all_ngt(const vector<scalarT>& lhs,
               const vector<scalarT>& rhs);
template<typename scalarT>
  bool all_nle(const vector<scalarT>& lhs,
               const vector<scalarT>& rhs);
template<typename scalarT>
  bool all_nlt(const vector<scalarT>& lhs,
               const vector<scalarT>& rhs);
// Any predicates
template<typename scalarT>
  bool any_eq(const vector<scalarT>& lhs,
              const vector<scalarT>& rhs);
template<typename scalarT>
  bool any_ge(const vector<scalarT>& lhs,
              const vector<scalarT>& rhs);
template<typename scalarT>
  bool any_gt(const vector<scalarT>& lhs,
              const vector<scalarT>& rhs);
template<typename scalarT>
  bool any_le(const vector<scalarT>& lhs,
              const vector<scalarT>& rhs);
template<typename scalarT>
  bool any_lt(const vector<scalarT>& lhs,
              const vector<scalarT>& rhs);
template<typename scalarT>
  bool any_ne(const vector<scalarT>& lhs,
              const vector<scalarT>& rhs);
template<typename scalarT>
  bool any_nge(const vector<scalarT>& lhs,
               const vector<scalarT>& rhs);
template<typename scalarT>
  bool any_ngt(const vector<scalarT>& lhs,
```

```
                const vector<scalarT>& rhs);
  template<typename scalarT>
    bool any_nle(const vector<scalarT>& lhs,
                const vector<scalarT>& rhs);
  template<typename scalarT>
    bool any_nlt(const vector<scalarT>& lhs,
                const vector<scalarT>& rhs);
}
```

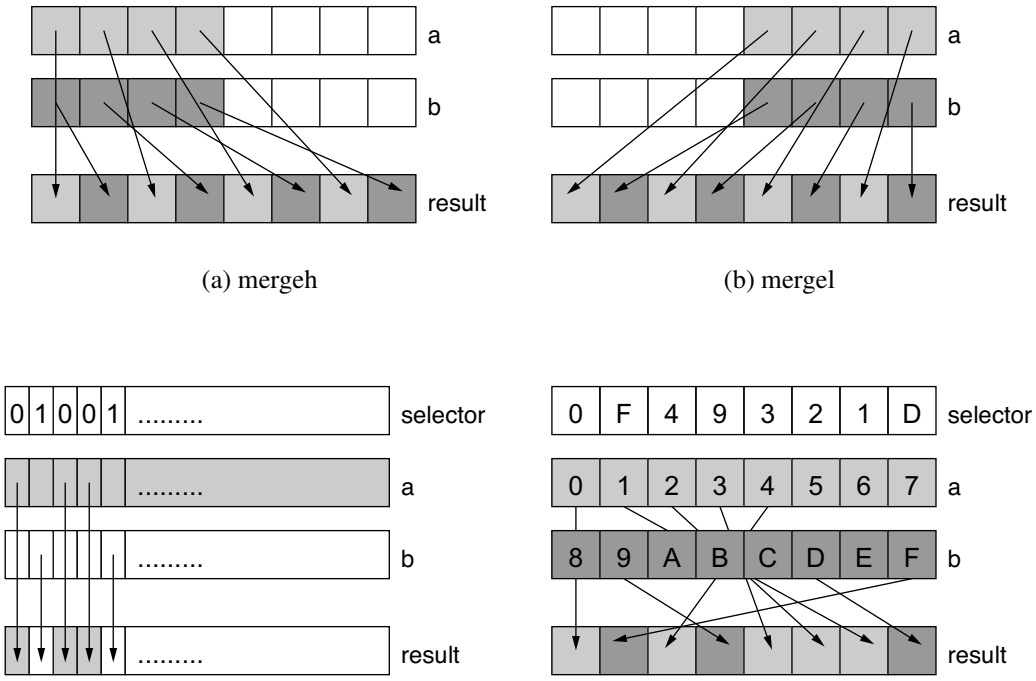
6.14 Vector processor specific functions

VVM contains vector functions which are modelled after AltiVec instructions. In the absence of a real VPU, these functions perform little work since each `vvm::vector` would have only one scalar.

```
namespace vvm {
  template<typename scalarT>
    vector<scalarT> lvsl(const scalarT* const p,
                         const offset_t offset = 0);
  template<typename scalarT>
    vector<scalarT> lvsl(const vector<scalarT>* const p,
                         const offset_t offset = 0);
  template<typename scalarT>
    vector<scalarT> lvsr(const scalarT* const p,
                         const offset_t offset = 0);
  template<typename scalarT>
    vector<scalarT> lvsr(const vector<scalarT>* const p,
                         const offset_t offset = 0);
  template<typename scalarT>
    vector<scalarT> mergeh(const vector<scalarT>& a,
                           const vector<scalarT>& b);
  template<typename scalarT>
    vector<scalarT> mergel(const vector<scalarT>& a,
                           const vector<scalarT>& b);
  template<typename scalarT>
    vector<scalarT> select(
      const vector_traits<vector<scalarT> >::bool_type,
      const vector<scalarT>& a,
      const vector<scalarT>& b)
```

```
  template<typename scalarT>
    vector<scalarT> perm(const vector<scalarT>& selector,
                         const vector<scalarT>& a,
                         const vector<scalarT>& b);
  template<typename scalarT>
    vector<scalarT> max(const vector<scalarT>& a,
                        const vector<scalarT>& b);
  template<typename scalarT>
    vector<scalarT> min(const vector<scalarT>& a,
                        const vector<scalarT>& b);
  template<typename scalarT>
    typename ct::promote<scalarT>::type
      sum(const vector<scalarT>& a);
}
```

Figure 6.1 VVM vector processor specific functions



(a) mergeh (b) mergel

(c) select (d) perm

lvsl: This generates a `vvm::vector` with ascending scalars that start from `vvm::uoffset(p + offset)`. Equation 6.1 summarises `lvsl`'s operation where s_0 is the most significant element. The value of s_0 is directly proportional to `vvm::uoffset(p + offset)`.

$$\begin{aligned} s_0 &= \texttt{vvm} :: \texttt{uoffset}(\texttt{p} + \texttt{offset}) \\ s_i &= (s_{i-1} + 1) \end{aligned} \tag{6.1}$$

lvsr: `lvsr` is similar to `lvsl` except the most significant element s_0 is `VVM_SCALAR_COUNT - vvm::uoffset(p + offset)`. The value of s_0 is inversely proportional to `vvm::uoffset(p + offset)`. Equation 6.2 summarises `lvsr`'s operation.

$$\begin{aligned} s_0 &= \texttt{VVM\_SCALAR\_COUNT} - \texttt{vvm} :: \texttt{uoffset}(\texttt{p} + \texttt{offset}) \\ s_i &= (s_{i-1} + 1) \end{aligned} \tag{6.2}$$

mergeh: The most significant scalars in `vvm::vectors a` and `b` are interleaved to produce the desired result. For a graphical representation, see Figure 6.1(a).

mergel: The least significant scalars in `vvm::vectors a` and `b` are interleaved to produce the desired result. For a graphical representation, see Figure 6.1(b).

select: The `select` function is used to obtain the results of comparisons. For example, to obtain the results of `a > b`, you would write:

```
select(a > b, a, b);
```

The `select` function's behaviour is illustrated in Figure 6.1(c). It selects each bit in the `vvm::vectors` individually because it is modelled after the `vec_sel` AltiVec command. It is possible to implement `select` using bitwise operations if the current VPU does not have a comparable command.

perm: The permutation operation is used to select scalars from two `vvm::vectors`. The `selector` contains which scalar from `vvm::vector a` or `b` should be placed at the corresponding position in the result. Figure 6.1(d) shows how the `perm` function operates graphically.

Scalars in the `selector` which have larger values than twice the constant scalar count, results in undefined values at those positions.

Note that the `perm` function can be used to produce the same results as `mergel`, `mergeh` and sometimes `select`. Where possible, the user should use one of the other vector functions. The `perm` function should be used as a last resort because it has less VPU support. AltiVec is the only vector-processor technology available on the desktop at the time of writing that supports the permutation operation. Even when AltiVec is enabled, the `perm` function is easy to implement only for `char` types. This is because the AltiVec `vec_perm` function only operates in the same manner as VVM's for `char` types.

Unlike AltiVec's `vec_perm` instruction, which will let the user rearrange the bytes of scalars in its VPU vectors, VVM's `perm` does not. It only allows scalars in the `vvm::vectors` to be rearranged.

max: The `max` function returns a `vvm::vector` whose scalars are the larger of corresponding scalars from `vvm::vectors` `a` and `b`. The `max` function comes from the `vec_max` AltiVec operation. The `max` function is equivalent to:

```
select(a > b, a, b)
```

min: The `min` function returns a `vvm::vector` whose scalars are the smaller of corresponding scalars from `vvm::vectors` `a` and `b`. The `min` function comes from the `vec_min` AltiVec operation. The `min` function is equivalent to:

```
select(a < b, a, b)
```

sum: The `sum` function returns the sum of all the scalars within the `vvm::vector` `a`.

6.15 Type conversions

Type conversion is handled using the `vector_cast` template function. The template parameter accepted by this template function is the `vvm::vector` that the user wants to convert to.

```
namespace vvm {
  template<typename toVectorT, typename fromVectorT>
    toVectorT vector_cast(const fromVectorT& from);
}
```

The emulation layer is expected to provide a `vector_cast` that will convert from `vvm::vector` by invoking `static_cast` on each scalar element. Therefore it will be possible to change any `vvm::vector` type to another. Vector-processor implementations can provide type conversions that operate on VPU vectors. The exact conversions that are supported is not defined.

Because `vector_cast` accepts `vvm::vectors` as template parameters, it is not valid to write `vector_cast<float>(a)`. Instead the programmer should write `vector_cast<vector<float> >(a)`. This is to keep `vector_cast` consistent with other cast templates like `static_cast`, `reinterpret_cast` and `const_cast`. In all these casts, the template parameter is the type that we want to convert to.

6.16 Functors

In the Standard C++ library, all comparison functors return `bool`. These functors cannot be instantiated with `vvm::vectors`, because the result of `vvm::vector` comparisons is a `vvm::vector` of booleans. Because Standard C++ comparison functors cannot be instantiated with `vvm::vectors`, VVM will also provide substitute comparison functors. The VVM comparison functors are in a different namespace, `vvm_functional`, and in a different header file, `vvm/functional.h`, from the rest of VVM. This is to enable the user to specify that the namespace is to be used in preference to the Standard C++ library. VVM comparison functors can be instantiated for both scalars and `vvm::vectors`.

```
namespace vvm::functional {
  template<typename scalarT> struct equal_to;
  template<typename scalarT> struct not_equal_to;
  template<typename scalarT> struct less;
  template<typename scalarT> struct less_equal;
  template<typename scalarT> struct greater;
  template<typename scalarT> struct greater_equal;
}
```

6.17 VVM settings

VVM allows the user to obtain information about the current VVM implementation and the current vector-processor implementation. In addition, it provides a few user-configurable options. All settings are available at compile-time via macros because the information provided never changes during a single compilation, and allows VVM itself or the user to make compile-time decisions. Moreover, it is less likely for macros to be set to the wrong value than global constant variables. If global constant variables were used, there is a chance that the variables are compiled differently from the rest of the program, thereby leading to incorrect values. VVM settings are provided either for the purposes of making compile-time decisions, or for display to the user.

VVM is an active library (for more information about active libraries, see Veldhuizen & Gannon (1998)), because it helps the compiler compile itself. For example, the constant scalar count is not defined by VVM, but is instead determined by the current vector-processor implementation. VVM code will need to change if constant scalar count changed. VVM users could also decide to call different functions depending on information about their vector-processor implementation. In many cases, just knowing which VPU is available is enough to make these decisions.

6.17.1 Information

VVM provides a variety of information to the user. Some of this information can be displayed on the screen, while others are suitable for making compile-time decisions using the preprocessor.

VVM_REVISION: This is the revision number of the VVM specification that is implemented by the VVM implementation. VVM implementations implementing the specification described in this chapter should set `VVM_REVISION` to 1.

```
#define VVM_REVISION 1
```

Subsequent versions would have 2, 3 and so on. While there is no guarantee that code using revision 1 will run correctly in revision 2, it is a strong possibility. All programs should check that the VVM_REVISION is greater than or equal to the revision number that they are using.

VVM_INFO_IMPLEMENTATION: This should be set to the name of the VVM implementation.

VVM_INFO_AUTHOR: This should be set to a human-readable string describing the author that created this VVM implementation.

VVM_INFO_COMPANY: This should be set to a human-readable string describing the company that created this VVM implementation.

VVM_INFO_VERSION: This should be set to a human-readable string describing the version of the VVM implementation.

VVM_INFO_BUILD: This should be set to the version of the VVM implementation as a number. This number must be larger for subsequent versions.

VVM_VECTOR_PROCESSOR: The `VVM_VECTOR_PROCESSOR` macro will be defined only if there are any active vector-processor implementations. This indicates whether a real VPU that is supported is available.

VVM_<Platform>: To detect what the active vector-processor implementation is, the program can check if one of the macros listed in Table 6.3 is defined. Some vector-processor technologies are subsets of other vector-processor technologies. For example, SSE is an extension of MMX. If the vector-processor implementation supports SSE, then it also supports MMX. For VPUs that are not listed, the VVM implementation is allowed to define any macro that it sees fit, as long as the macro follows the `VVM_`<Platform> convention. VVM implementations may also define `VVM_` macros for individual processors, as long as they also define architecture macros as required.

Table 6.3 VVM macros signalling availability of VPUs

Macro	Vector Processor Architecture	Examples
VVM_ALTIVEC	AltiVec	Motorola G4, IBM Power4
VVM_MMX	MMX	Intel Pentium
VVM_3DNOW	3DNow!	AMD Athlon
VVM_SSE	SSE	
VVM_SSE2	SSE2	

VVM_INFO_VECTOR_PROCESSOR: The `VVM_INFO_VECTOR_PROCESSOR` macro is a human-readable string containing the name of the VPU that is currently active. In the absence of a vector-processor implementation, it should be set to `"None"`. This is used by the programs using VVM to tell the user which VPU is active. It is not expected that the program makes compile-time decisions using this information.

VVM_SCALAR_COUNT: This macro evaluates to the number of scalars in all `vvm::vectors`.

VVM_PREFERRED_SCALAR_COUNT: This is the preferred scalar count for the current vector-processor implementation. VVM implementations set this value. In general, if `VVM_USER_SCALAR_COUNT` is not defined or is not a sensible value, `VVM_SCALAR_COUNT` will be set to this instead.

VVM_TRUE: This macro evaluates to the integer scalar value for `true`. This value is `~0`.

VVM_FALSE: This macro evaluates to the integer scalar value for `false`. This value is `0`.

6.17.2 User-configurable options

VVM provides a few user-configurable options. These options should be set before the first VVM header file is included.

VVM_USER_NO_VECTOR_PROCESSOR: If this macro is defined, the VVM implementation should not enable any vector-processor implementations, even if they are available. Using the scalar processor regardless of the vector processor allows a scalar version to be built on platforms that have vector-processor support.

VVM_USER_SCALAR_COUNT: The user sets this to his preferred scalar count. The VVM implementation can decide which values are acceptable. This is because, some values of scalar counts are impractical. For example, setting the scalar count to 17 for AltiVec is bad. There is no way to represent 17 elements as VPU vectors

in AltiVec without wasting space. In addition, there could also be constraints in the way the functions are implemented. The VVM implementation should raise an error if `VVM_USER_SCALAR_COUNT` is invalid.

6.18 Conclusion

This chapter presented the design issues and rationale behind the Virtual Vector Machine (VVM), a new abstract VPU designed to be used as the VPU for a generic, vectorised, machine-vision library. VVM aims to represent desktop VPUs, such as AltiVec, MMX and 3DNow!. Being an abstract VPU, VVM has an idealised instruction set and constraints common to desktop VPUs. VVM has three constraints common to desktop VPUs: short vectors, fixed vector sizes and fast access only to aligned memory addresses. While the size of a `vvm::vector` is fixed, unlike desktop VPUs, all `vvm::vectors` have the same scalar count regardless of type. This constant scalar count is important for generic programming, which is required for the creation of a generic, vectorised library. Other features that support the creation of a generic, vectorised library include templated `vvm::vectors`, traits, and consistent functions for both scalar and `vvm::vector` operations. Because of the high cost of converting types in vector programs, VVM does not provide automatic type conversion; it only supports explicit type conversions.

While all `vvm::vectors` contain the same number of scalars, the value of this fixed scalar count is not specified by VVM to allow more VPUs to be represented. Sensible values for scalar counts are 16 for AltiVec, 8 for MMX and 1 for the scalar processor. VVM has constant scalar count for ease of use and because this is important for generic programming (Appendix C). Having different scalar counts per `vvm::vector` makes it difficult to perform type conversions, which are necessary when performing some operations like convolutions and equalisations.

VVM can access only aligned memory address quickly. Like SSE and SSE2 (Adv 2002), VVM also provides slower access to unaligned addresses. In AltiVec, unaligned memory can be accessed by loading aligned memory addresses and performing some transformations (Lai & McKerrow 2001, Ollmann 2001, App 2004*b*). Even though unaligned addresses are supported, it is still advisable to use only aligned addresses where possible.

Templated `vvm::vectors` makes creating templated vector programs easier. Templated vector programs are required for a generic, vectorised library.

Traits are important when automating the generation of more complicated generic algorithms (Köthe 1999). Trait information can be used in the implementation of VVM itself. They are also important for deriving the appropriate boolean type, since unlike scalar programs, vector programs use more than one boolean type. Promotion traits are important for writing templated code where promotion is necessary, such as in convolu-

tions.

VVM has consistent functions for both scalar and `vvm::vector` operations where applicable. This makes VVM easier to use and introduces less surprises; it also allows the same templated code to be instantiated with a scalar or a `vvm::vector`. Furthermore, they also allow most Standard C++ functors to be used with `vvm::vectors`.

Type conversion is discouraged in vector programs because the effect of these operations on a vector program's speed is quite pronounced. Type conversions will change the type of VPU vectors in a `vvm::vector`, which changes the maximum theoretical speedup. However, as mentioned previously, type conversions are important to some image-processing operations. The compromise was to make the user specify type conversions explicitly.

Chapter 7

Virtual Vector Machine's Implementation

In Chapter 5, the abstract VPU was presented. In Chapter 6, a real abstract VPU, called Virtual Vector Machine (VVM), was specified. In this chapter, how the VVM specification can be implemented using only Standard C++, and the costs associated are discussed. Using only Standard C++ makes the implementation more portable across different compilers and makes it easier for users to use, since the user does not have to perform any special additional steps to compile the program. Since VVM is an abstract VPU, knowing how implementable it is provides insights into how implementable any abstract VPU is. In particular it highlights the performance costs of an abstract VPU implemented using only Standard C++.

This chapter starts with the implementation of VVM fundamental scalar types and VVM's traits. This is followed by an investigation into the costs of performing a VVM operation, and switching between the emulation layer and the active vector-processor implementation. The implementation of type conversion functions and function objects is then discussed. This chapter closes with a presentation of the results from a performance measurement tool on several VVM implementations created as part of this thesis.

7.1 Fundamental scalar types

The implementation of VVM fundamental scalar types is discussed in this section. Most VVM fundamental scalar types are straightforward to implement, because they are fundamental C++ types, and thus require no implementation. Types like `sint8`, `sint16`, and `sint32`, can be implemented as typedefs, once we can find types with the appropriate characteristics. Finding scalar types with the appropriate characteristics is discussed later in Section 7.2.2. Therefore, the only VVM fundamental scalar types worth discussing in this section are the boolean scalar types.

Boolean scalar types can be implemented using simple wrappers.

```
template<typename scalarT> class boolean {
  scalarT _value;
public:
  boolean() {}
boolean(const scalarT value) : _value(value) {
}
  operator scalarT() const {
    return _value;
  }
};
typedef boolean<char> bchar;
typedef boolean<short int> bshort;
typedef boolean<int> bint;
typedef boolean<long int> blong;
typedef boolean<float> bfloat;
typedef boolean<double> bdouble;
typedef boolean<long double> bldouble;
typedef boolean<int8> bint8;
typedef boolean<int16> bint16;
typedef boolean<int32> bint32;
```

Using typedefs for boolean scalars is inappropriate, because typedefs prevent AltiVec boolean VPU vectors from being mapped correctly. Because a typedef is treated as the original type in C++, it would not be possible to provide different trait information for the typedef and the original type. For example, the following code will not compile.

```
typedef signed char bchar;
template<> struct scalar_traits<signed char> {
  // ..
  typedef __vector signed char vpvector_type;
};
// This will fail to compile because bchar is a typedef
// of signed char
template<> struct scalar_traits<bchar> {
  // ...
  typedef __vector bool char vpvector_type;
};
```

What is required for VVM boolean scalars are types that act like the original types, are the same size as the original types, but are distinct types from the original types. The `boolean`

wrapper fulfils these requirements. The conversion operator included in the definition of the `boolean` wrapper allows the boolean scalar to act like the original type.

7.2 Traits

VVM provides three trait templates `scalar_traits`, `vpvector_traits` and `vector_traits` that provide information on scalars, VPU vectors and `vvm::vectors` respectively. Information provided by these traits include scalar-to-VPU vector and its inverse mapping, the number of elements in the VPU vector and the corresponding integer, real and boolean types.

7.2.1 Mapping scalars to VPU vectors

At first glance, mapping scalars to VPU vectors for AltiVec seems easy; scalar types can be converted to AltiVec vector types by simply adding the `__vector` modifier. For example, under this mapping scheme, `unsigned char` and `unsigned short` map to `__vector unsigned char` and `__vector unsigned short` respectively. This simple mapping scheme has three major problems. The first is the question of what `char` maps to. The second is that since the size of scalar types in C++ is not fixed, the scalar type could be mapped to a VPU vector with a different scalar element size. Some compilers, like xlC for example, have different sizes for fundamental types when given different compiler switches. For example, in xlC, `-ldbl128` will increase the size of `long doubles` from 64 bits to 128 bits (IBM 2002). The same issue arises when mapping from VPU vectors to scalars. In addition, this simple mapping can result in a less than optimal mapping. For example, if both `long int` and `int` are 32 bits long, then they can be both mapped to `__vector signed int`.

The first problem is easy to solve since `std::numeric_limits<char>::is_signed` returns `true` or `false` if `char` is signed or unsigned respectively. However because this information is known at compile-time and not at preprocess-time, the preprocessor cannot be used to decide whether `char` maps to `__vector signed char` or `__vector unsigned char`. Instead, template metaprogramming (refer to Section 4.4 for more information) is required. The following code illustrates how to select the appropriate VPU vector for `char`.

```
template<> struct scalar_traits<char> {
  // ... Other fields go here
  typedef boost::mpl::if_c<
    std::numeric_limits<char>::is_signed,
    __vector signed char,
    __vector unsigned char>::type vpvector_type;
```

```
};
```

The second problem is more difficult to solve. Ideally, scalar types should be matched to VPU vectors that are of the same type and scalar size. For example, `signed char` should map to a VPU vector whose scalars are signed integers and are 8 bits long. The type can be signed integer, unsigned integer, reals or booleans. By using these two characteristics as template parameters, a VPU vector with the desired characteristics can be found at compile-time. The type is well-known, except in the case of `char` which is either a signed or unsigned integer. The scalar size can be determined at compile-time using the `sizeof` operator. The following code illustrates how this works:

```
// Enumeration of type types
enum { signed_int, unsigned_int, real, boolean };

// If no appropriate VPU vector, just use an appropriate scalar
template<int kind, int scalar_size> struct find_vpvector {
  typedef typename find_scalar<kind, scalar_size>::type type;
};

// An entry for each VPU vector for AltiVec
template<> struct find_vpvector<signed_int, 1> {
  typedef __vector signed char type;
};
template<> struct find_vpvector<unsigned_int, 1> {
  typedef __vector unsigned char type;
};
template<> struct find_vpvector<boolean, 1> {
  typedef __vector bool char type;
};
template<> struct find_vpvector<real, 4> {
  typedef __vector float type;
};
// And so on ...
template<> struct scalar_traits<signed char> {
  // Other fields go here ...
  typedef typename find_vpvector<
    signed_int, sizeof(signed char)>::type vpvector_type;
};
```

From the above code, `scalar_traits<signed char>::vpvector_type` is `__vector signed char` since the kind is `signed_int` and `sizeof(char)` is 1. In addition, if there is

no appropriate VPU vector type, the default `find_vpvector` returns an appropriate scalar using the `find_scalar` template metafunction, which is discussed in the next section.

7.2.2 Mapping VPU vector to scalar types

We can try to map VPU vectors to scalar types using the inverse operation of the process described in the previous section. In this case, instead of explicitly specifying the scalar size, the scalar size should be `sizeof(scalarT)`. The size cannot be explicitly specified because the size is implementation defined. However, each specialised `vpvector_traits` will know the explicit size that is required, because it is specified in AltiVec.

```
template<int kind, int scalar_size> struct find_scalar;

// Some example specialisations
template<>
struct find_scalar<signed_int, sizeof(short)> {
  typedef short type;
}
template<>
struct find_scalar<unsigned_int, sizeof(unsigned short)> {
  typedef unsigned short type;
}
template<>
struct find_scalar<signed_int, sizeof(int)> {
  typedef int type;
}
template<>
struct find_scalar<unsigned_int, sizeof(unsigned int)> {
  typedef unsigned int type;
}
template<>
struct find_scalar<signed_int, sizeof(long)> {
  typedef long type;
}
// ...
```

Unfortunately, this implementation does not work because some of the scalar types have the same size. This leads to duplicate entries. For example, in Apple GCC 3.1 20021003, `int`s and `long int`s are both 4 bytes. Therefore, they will both end up specialising

find_scalar<signed_int, 4> and find_scalar<unsigned_int, 4>. This of course leads to compilation problems.

Trying to use the scalar-to-VPU vector mapping instead to calculate from the VPU vector-to-scalar mapping, might result in a vpvector_traits implementation.

```
template<>
struct vpvector_traits<scalar_traits<char>::vpvector_type> {
  static const bool is_specialised = true;
  typedef char scalar_type;
  enum {
    scalar_count = sizeof(scalar_traits<char>::vpvector_type) /
                   sizeof(char)
  };
};
```

Unfortunately this also leads to the same problem. The crux of the problem is that the scalar-to-VPU vector mapping is a many-to-one mapping.

Instead of trying to use scalar characteristics or the vpvector_type as template parameters, it is possible to implement a template metaprogram that will iterate through a list of types until it finds the appropriate type. This is where typelists come in. The template metafunction find_scalar for signed integers can be easily implemented as follows, assuming signed_integer_types is a typelist that contains only scalars that are signed integers.

```
// Find scalar type that matches type and size
template<int kind, int size> struct find_scalar;
template<int size> struct find_scalar<signed_int, size> {
  typedef typename ct::find_first<
    signed_integer_types,
    ct::has_sizeof<size>::func, int>::type type;
};
```

In the preceding code snippet, the find_first command will find the first type in signed_integer_types that is the same size as size. If there is no type that satisfies this criterion, then it will return int. The default is important when mapping bool_type, integer_type and float_type.

The same style can be repeated for unsigned integers, real numbers and booleans, with an appropriate default type. The template metafunction find_scalar for boolean scalars should probably return ct::null_type if no type is found, because the default boolean type should be the original type.

7.2.3 bool_type, integer_type, float_type

The types `bool_type`, `integer_type` and `float_type` return corresponding boolean, integer and real types. The VVM specification says that where possible, these types should be the same size as the original. This is important in reducing type conversion costs.

Fortunately, the `find_scalar` template from Section 7.2.2 makes it easier to fulfil VVM's request. Using `find_scalar`, the `bool_type`, `integer_type` and `float_type` for `int` can be written as

```
template<> struct scalar_traits<int> {
  // Other fields go here
  typedef typename find_scalar<boolean,
                               sizeof(int)>::type bool_type;
  typedef int integer_type;
  typedef typename find_scalar<real,
                               sizeof(int)>::type float_type;
};
```

There is no need to use `find_scalar` for `integer_type`, because `int` is already an integer type. Likewise, for `float`, `find_scalar` would not be used for `float_type`. Because the `find_scalar` template metafunction returns a default type if there is no appropriate type, there is no need to check whether an appropriate type was found.

Note that for `bool_type`, if there is no appropriate type, then the original type should be used. This means that `find_scalar<boolean, T>::type` should probably return `ct::null_type` if no type was found to facilitate the detection of no appropriate boolean types. The type `bool_type` can be implemented as follows.

```
typedef typename find_scalar<boolean,
                             sizeof(scalarT)>::type _bool_scalar;
typedef boost::mpl::if_c<
  boost::is_same<_bool_scalar, ct::null_type>,
  scalarT, // Not found
  _bool_scalar>::type bool_type;
```

7.3 Performing an operation

The operation implemented in this section is the addition operator whose function prototype is as follows:

```
template<typename scalarT> const vector<scalarT>
  operator+(const vector<scalarT>&, const vector<scalarT>&);
```

In order to process a user's request, the abstract VPU has to decide whether the function should be executed by the active vector-processor implementation or the emulation layer. In order to make this decision, VVM requires trait information. After making this decision, VVM performs the actual operation. In order for VVM to be zero-cost, making the decision and performing the operation must have no overheads. The implementation and costs of performing the operation are discussed in this section. The implementation and costs of switching between the emulation and the active vector-processor implementation are covered in the next section.

The overheads of a function depend on the compiler and the function prototype. Functions that accept a reference argument to the result are easier for compilers to inline than functions that return their results, because functions that return their results return a copy which has to be created, copied to the real result variable and destroyed. Most functions and operators however return their result. The addition operator discussed in this section returns its result.

This section starts with a description of the test harness used. The hand-coded versions of a scalar and VPU vector operations are then covered. Two different methods of implementation, function overloading and expression templates, and their costs are then discussed.

7.3.1 The test harness

To measure the execution time of the functions, a test harness was written. The test harness adds an input array of a fixed size with itself a set number of times. The number of elements in the array is fixed. Number of operations refers to the number of times each element is added to itself. The test harness looks like the following:

```
T* r = new T[count];
T* a = new T[count];
timing_t start, stop;
start = timenow(); // Timer starts
for(int i = 0; i < count; ++i) {
  // Insert expression here. Examples are:
  //  r[i] = a[i] + a[i];
  //  r[i] = vec_add(a[i], a[i]);
}
stop = timenow();  // Timer ends
delete[] a;
delete[] r;
```

The test harness applies the operation to an array to prevent the compiler from optimising the for-loop away. For extra protection against excessive compiler optimisations, the test

harness could write the output to disk. For the graphs shown in this section however, this step was not necessary; the same results were obtained with or without writing to disk.

When comparing the speed of expressions involving `vvm::vectors` with only a single element, `T` was `signed char` and `__vector signed char` for scalar and VPU vector expressions respectively. The type `char` was chosen because in AltiVec, when using the preferred scalar count, the only `vvm::vectors` to have a single VPU vector are the `vvm::vectors` based on `char`. There was no particular reason for choosing `signed char` over `unsigned char`.

When comparing the speed of expressions involving `vvm::vectors` with more than one element, `T` was `signed int` and `__vector signed int` for scalar and VPU vector expressions respectively. The type `int` was chosen because in AltiVec, when running with the preferred scalar count of 16, `vvm::vectors` based on `int` have the largest number of VPU vectors.

Programs were compiled using both Apple GCC 3.1 20021003 and Apple GCC 3.3 20030304, with the `-Os` (optimise for size) switch. All programs were timed 20 times, and the lowest value was taken as the result.

7.3.2 The hand-coded reference versions

There are two different hand-coded versions — one for scalar expressions and one for VPU vector expressions. In a real abstract VPU, there will also be expressions that involve both scalar and VPU vector elements. However this facet was not investigated because its speed is expected to be in-between the speeds of scalar and VPU vector only expressions.

For scalar expressions, the expression is simply `r[i] = a[i] + a[i]`, `r[i] = a[i] + a[i] + a[i]`, `r[i] = a[i] + a[i] + a[i] + a[i]` and so on. For VPU vector expressions, the expression is `r[i] = vec_add(a[i], a[i])`, `r[i] = vec_add(vec_add(a[i], a[i]), a[i])`, `vec_add(vec_add(vec_add(a[i], a[i]), a[i]), a[i])` and so on.

Note that because `vvm::vector<int>` consists of 16 scalars and 4 VPU vectors, the count was multiplied by 16 and 4 for scalar and VPU vector expressions respectively. This is to ensure that the hand-coded version performs the same amount of work.

7.3.3 Operations on vvm::vectors with a single element

For `vvm::vectors` that contain only one element, a zero-cost `operator+` should be possible if the compiler removes the cost of the function call completely. The `operator+` function returns an object by value. According to Meyers (1996, Item 20), by-value returns imply the creation of a new object to hold the return value (and appropriate calls to the constructor and destructor for that object). The overheads of this function call therefore include not only the function call itself, but also the creation of a new object, its construction and destruction, and the copying of the new object to the destination. Meyers

(1996, Item 20) points out that while some functions, including the `operator+`, simply must return objects, it is possible to reduce the cost of these objects if the compiler supports return value optimisation. Return value optimisation is a common optimisation that removes the cost of the return object completely, by replacing it with the appropriate destination object. According to Meyers (1996, Item 20), the best way to help the compiler perform this optimisation is to use an unnamed return value. Two different `operator+` versions were implemented and timed. The first used an unnamed return value while the second used a named return value.

Unnamed return value

The addition operator implemented using unnamed return values is as follows:

```
// Scalar version
template<typename scalarT> inline const vector<scalarT>
  operator+(const vector<scalarT>& a, const vector<scalarT>& b) {
    return vector<scalarT>(a.scalar(0) + b.scalar(0));
  }
// VPU vector version (AltiVec)
template<typename scalarT> inline const vector<scalarT>
  operator+(const vector<scalarT>& a, const vector<scalarT>& b) {
    return vector<scalarT>(vec_add(a.vpvector(0), b.vpvector(0)));
  }
```

Named return value

Instead of using an unnamed return value, we can use a named return value. This implementation cannot be used if `scalarT` is `const`, because this implementation modifies the return value after it is created.

```
// Scalar version
template<typename scalarT> inline const vector<scalarT>
  operator+(const vector<scalarT>& a, const vector<scalarT>& b) {
    vector<scalarT> ret;
    ret.scalar(0) = a.scalar(0) + b.scalar(0);
    return ret;
  }
// VPU vector version (AltiVec)
template<typename scalarT> inline const vector<scalarT>
  operator+(const vector<scalarT>& a, const vector<scalarT>& b) {
    vector<scalarT> ret;
```

```
    ret.vpvector(0) = vec_add(a.vpvector(0), b.vpvector(0));
    return ret;
}
```

Results

Figures 7.1(a) and 7.1(b) show that using an unnamed return value is slower than a named return value when using Apple's GCC 3.1 20021003. The unnamed return value version was at worst within 80% slower than a hand-coded program for expressions involving up to five additions when operating on `vvm::vectors` with a single scalar element in scalar mode. The overheads of the named return value version on the other hand were within 1% for all cases tested in both scalar and AltiVec mode.

Figures 7.1(c) and 7.1(d) show that when Apple GCC 3.3 20030304 was used, the performance of returning named or unnamed return values were not significantly different. However, both versions had significant overheads when the number of additions in the expression was greater than two in scalar mode. Both versions were within 60% slower than hand-coded programs in scalar mode. In AltiVec mode, the results were similar to those obtained with Apple GCC 3.1 20021003; both versions were within 1% for all cases tested.

These results suggest that it is possible to perform the operation with minimal overheads, when there is only one scalar or VPU vector in a `vvm::vector` with Apple GCC 3.1 20021003, but not Apple GCC 3.3 20030304. When there is no active vector-processor implementation, `vvm::vectors` will consist of only one scalar. When AltiVec is active, only `vvm::vectors` for `char`, `signed char` and `unsigned char` will consist of only one VPU vector.

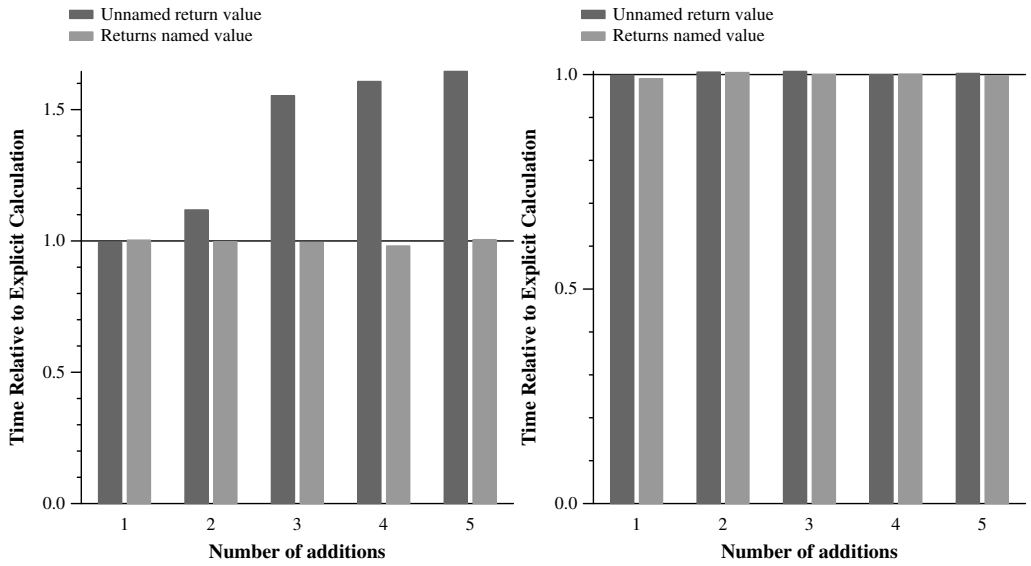
7.3.4 Function overloading

Since VVM has constant scalar count, `vvm::vectors` do not always consist of a single element. Furthermore, the number of elements in a `vvm::vector` is not usually known at programming-time; it is known at compile-time. Three different methods of handling one or more scalars or VPU vectors were investigated. The first method uses a for-loop to iterate over all the elements. The second version unrolls the for-loop at compile-time using template metaprogramming. The third version returns an unnamed `vvm::vector` and performs the addition operation during the construction of the `vvm::vector`.

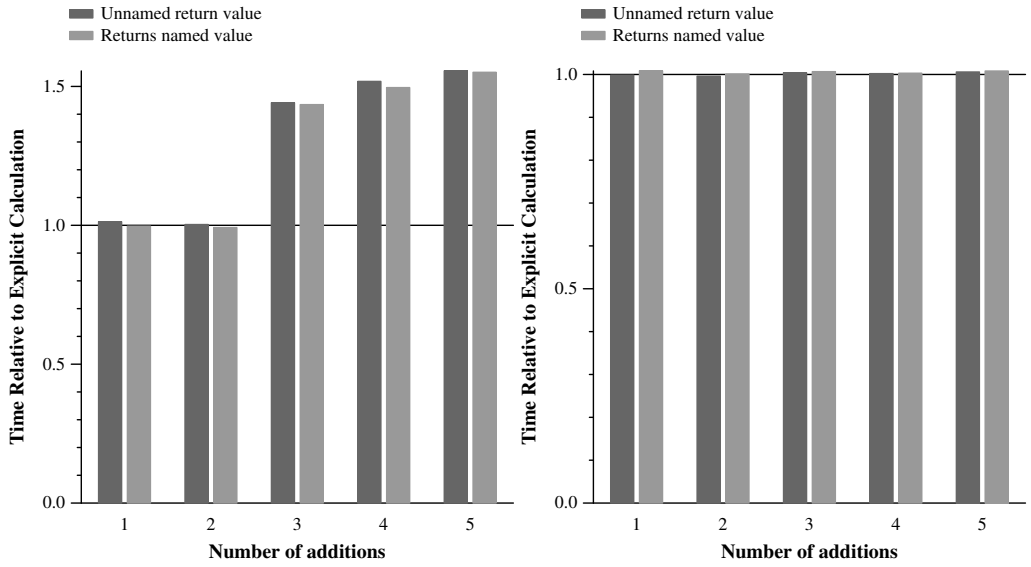
for-loop

The addition operator from Section 7.3.3 can be extended to handle `vvm::vectors` with more than one element by using a for-loop to iterate over all the elements. Hopefully, the

Figure 7.1 Performance of `operator+` based on function overloading that operates only on `vvm::vectors` with a element



(a) 1 scalar per vvm::vector (GCC 3.1) (b) 1 vp::vector per vvm::vector (GCC 3.1)



(c) 1 scalar per vvm::vector (GCC 3.3) (d) 1 vp::vector per vvm::vector (GCC 3.3)

compiler will notice that the number of times around the loop is low and constant, and thus unroll the for-loop automatically. The for-loop version is as follows:

```
// scalar version
template<typename scalarT> inline const vector<scalarT>
  operator+(const vector<scalarT>& a, const vector<scalarT>& b) {
    vector<scalarT> ret;
    for(int i = 0;
        i < vector_traits<vector<scalarT> >::scalar_count;
        ++i) {
      ret.scalar(0) = a.scalar(0) + b.scalar(0);
    }
    return ret;
  }
// VPU vector version (AltiVec)
template<typename scalarT> inline const vector<scalarT>
  operator+(const vector<scalarT>& a, const vector<scalarT>& b) {
    vector<scalarT> ret;
    for(int i = 0;
        i < vector_traits<vector<scalarT> >::vpvector_count;
        ++i) {
      ret.vpvector(0) = vec_add(a.vpvector(0), b.vpvector(0));
    }
    return ret;
  }
```

Figure 7.2 shows that unfortunately the compiler does not remove the for-loop for us. In fact it does not even remove the for-loop when there is always only a single iteration — if it did, this version would have been zero-cost for `vvm::vectors` with a single element.

Unrolled for-loop

Since the compiler is not removing the for-loop for us, we have to do it ourselves. If we knew exactly how many elements there were at program-time, then we can simply write code such as:

```
// scalar version
template<typename scalarT> inline const vector<scalarT>
operator+(const vector<scalarT>& a, const vector<scalarT>& b) {
  vector<scalarT> ret;
  ret.scalar(0) = a.scalar(0) + b.scalar(0);
```

```
  ret.scalar(1) = a.scalar(1) + b.scalar(1);
  ret.scalar(2) = a.scalar(2) + b.scalar(2);
  // ...
  ret.scalar(N) = a.scalar(N) + b.scalar(N);
  return ret;
}
// VPU vector version
template<typename scalarT> inline const
vector<scalarT>
operator+(const vector<scalarT>& a, const vector<scalarT>& b) {
  vector<scalarT> ret;
  ret.vpvector(0) = vec_add(a.vpvector(0), b.vpvector(0));
  ret.vpvector(1) = vec_add(a.vpvector(1), b.vpvector(1));
  ret.vpvector(2) = vec_add(a.vpvector(2), b.vpvector(2));
  // ...
  ret.vpvector(N) = vec_add(a.vpvector(N), b.vpvector(N));
  return ret;
}
```

We would know the exact number of elements if we knew the exact mapping between scalars and VPU vectors and the scalar count at program-time. As discussed in Section 7.2, this unfortunately is something that we do not know. Fortunately, the C++ template mechanism provides us with a method for unrolling the for-loop completely without knowing the exact number of elements at program-time; the number only needs to be known at compile-time. The trait information in Section 7.2 provides this information. If the number of elements had been unknown even at compile-time, then it would not be possible to unroll the loop completely.

Unnamed return value

This version returns an unnamed return value, performing the addition while it is being created. The version tested has the number of elements hard-coded for simplicity. Since the number of elements is not really known until compile-time, this approach would not be adequate for a real implementation.

The unnamed return value tested looks like the following code:

```
// scalar version
template<typename scalarT>
inline const vector<scalarT>
operator+(const vector<scalarT>& a, const vector<scalarT>& b) {
```

```
  return vector<scalarT>(a.scalar(0) + b.scalar(0),
                         a.scalar(1) + b.scalar(1),
                         a.scalar(2) + b.scalar(2),
                         // ...
                         a.scalar(N) + b.scalar(N));
}
// VPU vector version
template<typename scalarT>
inline const vector<scalarT>
operator+(const vector<scalarT>& a, const vector<scalarT>& b) {
  return vector<scalarT>(vec_add(a.vpvector(0), b.vpvector(0),
                         vec_add(a.vpvector(1), b.vpvector(1),
                         vec_add(a.vpvector(2), b.vpvector(2),
                         // ...
                         vec_add(a.vpvector(N), b.vpvector(N));
}
```

Note that the code actually uses a constructor that is not required in VVM. In addition, if the scalar type and the VPU vector type are the same, as in scalar mode, then the constructor can only be defined once, since the two constructors would have the same method signature.

To create a version that supports an unknown number of elements at program-time, but known at compile-time, the user can write code similar to the following:

```
// n is the number of elements
template<int n> struct do_add;
// For 1 scalar in vvm::vector
template<> struct do_add<1> {
  static vector<scalarT>
    exec(const vector<scalarT>& a, const vector<scalarT>& b) {
      return vector<scalarT>(a.scalar(0) + a.scalar(0));
    }
};
// For 2 scalars in vvm::vector
template<> struct do_add<2> {
  static vector<scalarT>
    exec(const vector<scalarT>& a, const vector<scalarT>& b) {
      return vector<scalarT>(a.scalar(0) + a.scalar(0),
                             a.scalar(1) + a.scalar(1));
    }
```

```
};
// And so on
template<typename scalarT>
inline const vector<scalarT>
operator+(const vector<scalarT>& a, const vector<scalarT>& b) {
  return do_add<vector_traits<vector<scalarT> >::scalar_count
    >::exec(a, b);
}
```

The template `do_add` is specialised for all the possible number of elements accepted by the constructor, and the appropriate specialisation is picked up at compile-time. The `vvm::vector` also needs to have constructors that accept the entire range of possible element counts. The constructors do not actually have to be compilable code, since the real number of elements might be less. The code has to be careful to only use the constructor with the correct element count. This is similar to the constructor provided for `boost::tuple`.

Results

Figures 7.2(a) and 7.2(b) show that unrolling the for-loop using template metaprogramming introduced no significant overheads for `vvm::vectors` with a single scalar or VPU vector with Apple's GCC 3.1 20021003.

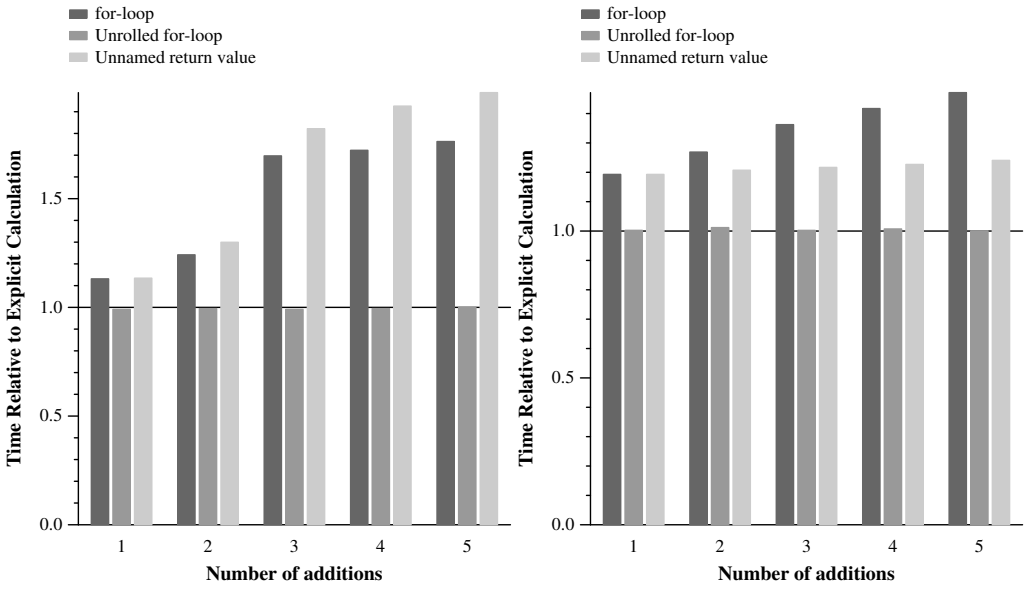
Figures 7.2(c) and 7.2(d) show the same performance behaviour for the unrolled for-loop version as for the versions discussed in Section 7.3.3 when Apple GCC 3.3 20030304 is used. Like the results obtained previously, the unrolled for-loop had significant overheads only in scalar mode and when the number of additions in the expression was greater than two. Like the results obtained for Apple GCC 3.1 20021003, the fastest version was the unrolled for-loop.

Figure 7.3 shows that as expected, as the number of elements in the `vvm::vector` increases, the overheads increase. At worst, for AltiVec's preferred scalar count of 16, using `vvm::vectors` with no AltiVec support would be about 70% slower than a hand-coded scalar program for a single addition for the fastest version, the unrolled for-loop. When the number of elements in the `vvm::vector` is greater than one, the overheads increase as the number of additions in the expressions increase. The results obtained for Apple GCC 3.1 20021003 were similar to the results obtained for Apple GCC 3.3 20030304.

7.3.5 Expression templates

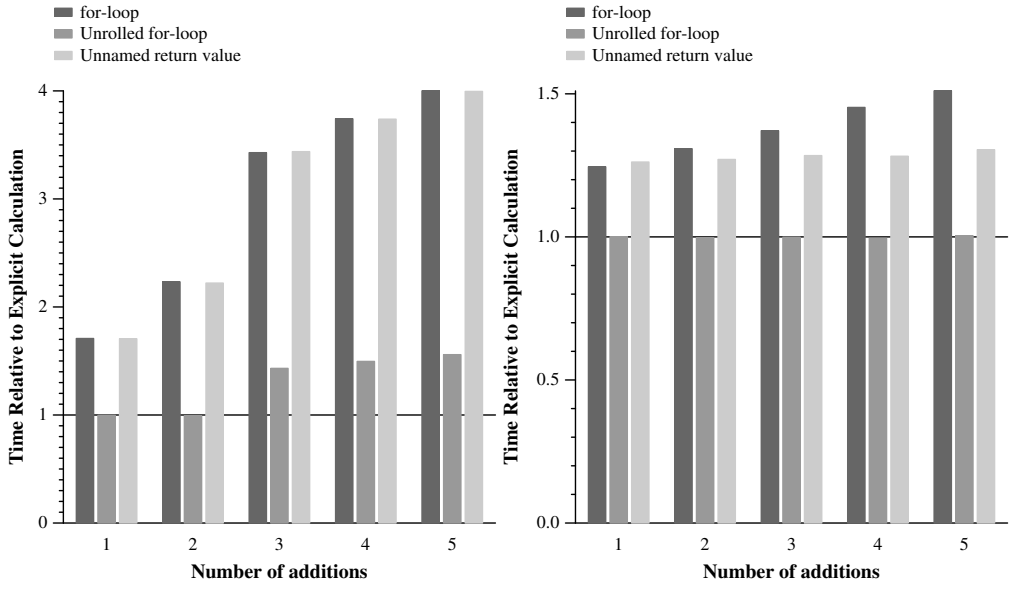
The reason why increasing the number of elements in a `vvm::vector` increases overheads when using function overloading is well known (Veldhuizen 1995*b*, Blinn 2000). Con-

Figure 7.2 Performance of `operator+` based on function overloading, when operating on only `vvm::vectors` with a single element



(a) 1 scalar per vvm::vector (GCC 3.1)

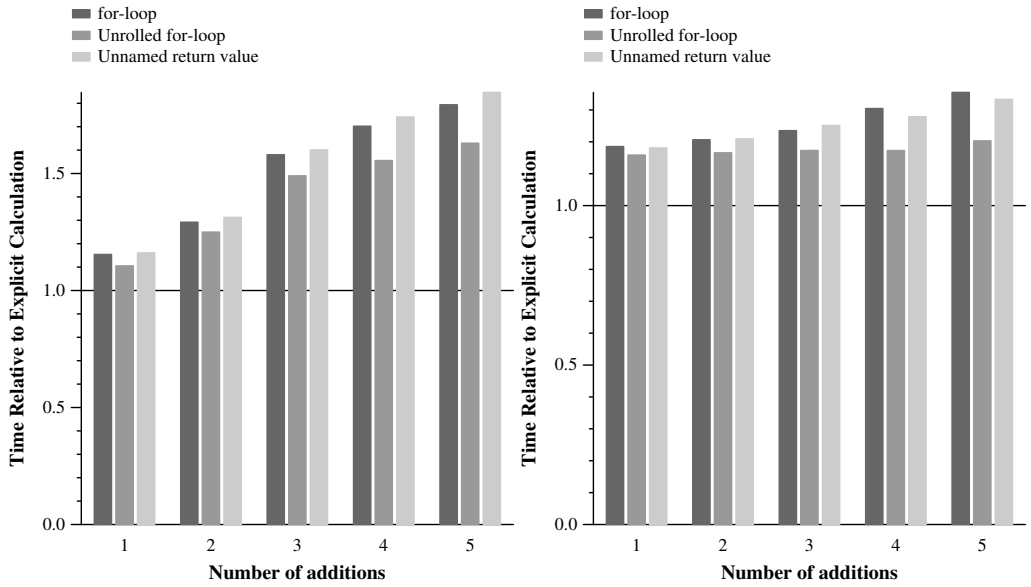
(b) 1 vp::vector per vvm::vector (GCC 3.1)



(c) 1 scalar per vvm::vector (GCC 3.3)

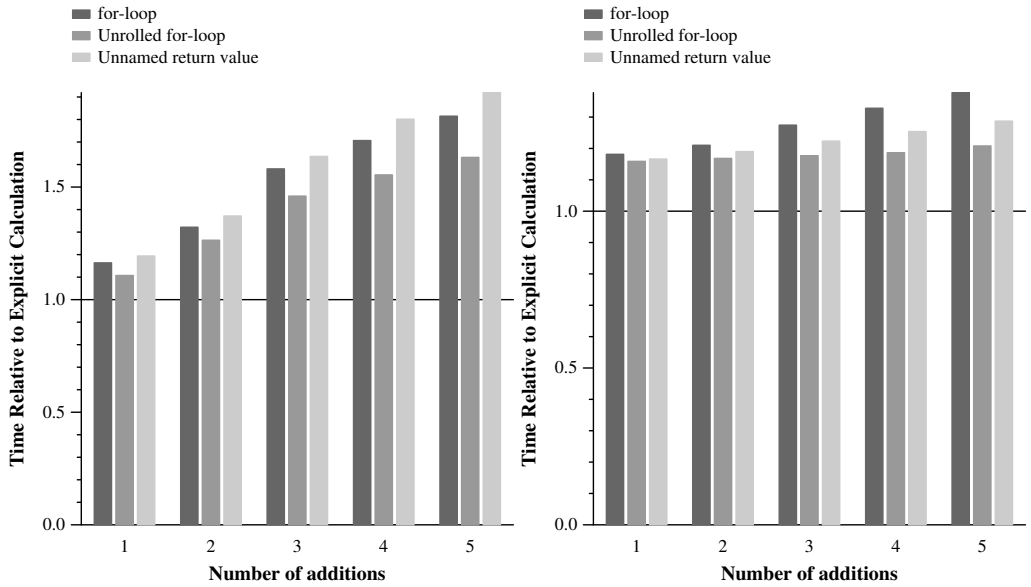
(d) 1 vp::vector per vvm::vector (GCC 3.3)

Figure 7.3 Performance of `operator+` based on function overloading, when operating on `vvm::vector<int>` with a constant scalar count of 16



(a) 16 scalars per vvm::vector (GCC 3.1)

(b) 4 vp::vectors per vvm::vector (GCC 3.1)



(c) 16 scalars per vvm::vector (GCC 3.3)

(d) 4 vp::vectors per vvm::vector (GCC 3.3)

sider the expression R=A+B+C+D. Using function overloading, the compiler will generate code similar to the following:

```
vvm::vector temp1 = A + B;
vvm::vector temp2 = temp1 + C;
vvm::vector temp3 = temp2 + D;
R = temp3;
```

Evaluating expressions in this manner is slower than hand-coded programs because more temporaries are used, and each statement contains its own for-loop (or its unrolled equivalent) (Veldhuizen 1995*b*, Blinn 2000).

The preferred execution method is to evaluate the entire expression for each element in the vvm::vector. Hand-coded C++ would look something like the following:

```
R.scalar(0) = A.scalar(0)+B.scalar(0)+C.scalar(0)+D.scalar(0);
R.scalar(1) = A.scalar(1)+B.scalar(1)+C.scalar(1)+D.scalar(1);
// And so on....
```

Expression templates is a technique that can help the compiler generate this faster implementation. It can generate the faster implementation because it delays evaluating expressions until the entire expression is used. With expression templates, A+B+C+D returns an object that represents the addition of A, B, C and D. When R=<expression object> is evaluated, the entire expression is evaluated simultaneously and the result is assigned to R. Since expression templates represent expressions as objects, they allow expressions to be passed to a function as an argument. This second feature allows users to create functors directly from expressions when using generic libraries.

Expression templates cannot be used efficiently for all functions because it is inefficient to pass a non-constant reference as a parameter. This is because the function that performs the evaluation is a const function (Blinn 2000, Veldhuizen 1995*b*). Being const, it cannot call non-const functions, and therefore does not support functions that take non-const reference arguments. Removing the const means that all fields cannot be const either, and this results in a slower expression template implementation, because of the extra copying that is required when the expression object is created. Examples of functions that cannot be implemented efficiently using expression templates include operator++ (both pre- and post- versions), operator-=, operator+=, amongst others.

When applying expression templates to VVM, it is important to remember that an abstract VPU expression can involve only scalars, only VPU vectors or a combination of both, that the number of elements in a vvm::vector is small and the number of elements in a vvm::vector is known at compile-time.

Expression templates however decompose an expression to a single type. For example, `A+B` becomes `A.scalar(i)+B.scalar(i)` or `A.vpvector(i)+B.vpvector(i)`. While types that can convert to and from this single type are allowed, converting a single VPU vector to a single scalar and vice versa is not possible, since they have different scalar counts. VPU vectors however can be considered to be an array of scalars. Therefore the expression template implementation for an abstract VPU should decompose a `vvm::vector` expression to an expression consisting of VPU vectors.

The number of elements in a `vvm::vector` is expected to be small. For such vectors, different papers have different performance results. Blinn (2000) showed that for short vectors (in Blinn's paper, there were three `floats` in a vector), expression templates can be faster than hand-coded C. Blinn (2000) said that his final version generated "optimal code on all examples" that were attempted. Veldhuizen (1995*b*) presented a more modest picture. Veldhuizen (1995*b*)'s implementation focused on expression templates involving long vectors, and the results from his paper show that for short vectors (less than 20), the performance of expression templates is quite poor when compared with hand-coded C. Veldhuizen (1995*b*)'s implementation only approached zero-cost for vectors with about a 1,000 elements. Veldhuizen (1995*b*)'s implementation never managed to run faster than hand-coded C despite the same assembly code being generated (after some compiler options and tuning performed by Veldhuizen). The poor performance of expression templates for short vectors was because of the time spent in the expression object constructors. Like Veldhuizen (1995*b*), Bassetti et al. (1998) also said that expression templates perform worse than hand-coded C. However the reason given for their poor performance was because more registers were required by the expression templates version.

Since the number of elements in a `vvm::vector` is always known at compile-time (and is a small number), we can unroll the for-loop using template metaprogramming. We cannot unroll the loop during programming, because the constant scalar count changes depending on the availability of real VPUs. Knowing the number of elements in a `vvm::vector` at compile-time gives any hand-coded version for VVM an edge over other expression template libraries, like Blitz++. While Blitz++'s `TinyVector` allows the user to specify the number of elements at compile-time, Blitz++ does not use this information to unroll the loop, since the same algorithms also cater for vectors where the number of elements is not known at compile-time.

Because different papers suggest different costs for short vectors when using expression templates, three different expression template versions were evaluated. The first version was a hand-coded version based on Veldhuizen (1995*b*)'s paper, the second uses Blitz++'s expression template engine and the final version was based on Blinn (2000)'s paper.

Expression templates based on Veldhuizen (1995*b*)'s paper

Like all expression template implementations, the first expression template implementation, based on Veldhuizen (1995*b*)'s paper, has a template, `_e`, that represents an expression. The `_e` template is shown below:

```
template<typename T> class _e {
  const T _expr;
public:
  typedef typename T::scalar_type scalar_type
  typedef typename T::return_type return_type;
  _e(T t) : _expr(t) {
  }
  return_type vpvector(int i) const {
    return _expr.vpvector(i);
  }
};
```

Unlike classes used to represent expressions by Veldhuizen (1995*b*) and Blinn (2000), `_e` has two additional types, `scalar_type` and `return_type`. The type `scalar_type` refers to the scalar type of operands in the current expression. It is used to decide the specialisation of the class representing the operation. The type `return_type` refers to the VPU vector type being returned. This is important for VVM because not all operators, namely the logical operators, return the same type as their arguments.

The template `_e` is specialised for a `vvm::vector`. This specialisation stores a constant reference to a `vvm::vector` instead of a copy as in the general version. The program will run without this specialisation, but will be slower, because of the extra copy. The general expression cannot use a reference because the result of an operation might be a temporary object. The addition operation for example returns a temporary object. The template `_e` specialised for a `vvm::vector` is shown below:

```
template<typename scalarT> class _e<vector<scalarT> > {
  const vector<scalarT>& _v;
public:
  typedef scalarT scalar_type;
  typedef typename scalar_traits<scalarT>::vpvector_type
    return_type;
  _e(const vector<scalarT>& v) : _v(v) {
  }
  return_type vpvector(int i) const {
    return _v.vpvector(i);
```

```
  }
};
```

The template `_binary_expr` is used to represent binary expressions. Other templates are also required for unary and ternary expressions in this implementation.

```
template<typename opT, typename leftT, typename rightT>
  class _binary_expr {
    const leftT _left;
    const rightT _right;
  public:
    typedef typename opT::scalar_type scalar_type;
    typedef typename opT::vector_type vector_type;
    typedef typename opT::return_type return_type;
    _binary_expr(const leftT& left, const rightT& right)
    : _left(left), _right(right) {
    }
    return_type vpvector(int i) const {
      return opT::evaluate(_left.vpvector(i),
                           _right.vpvector(i));
    }
  };
```

In addition to adding a specialisation for `vvm::vector`, `vvm::vector` also requires the following two additional functions — a constructor and an `operator=` that accept `_e`.

```
template<typename T>
  vector<scalarT>(const _e<T>& a);
template<typename T>
  vector<scalarT>& vector<scalarT>::operator=(const _e<T>& a);
```

These additional functions assign the expression to the `vvm::vector`. This assignment process starts the code generation from the expression template. The conversion constructor and `operator=` can be implemented as follows, using the `meta::EFOR` template metaprogram to unroll the for-loop.

```
template<typename scalarT, typename destT, typename arg1T>
  struct _copy_vpvector {
    template<int i> struct Code {
      static void
        exec(destT& dest, arg1T arg1) {
```

```
          dest.vpvector(i) = arg1.vpvector(i);
        }
    };
  };
template<typename scalarT> template<typename T>
  vector<scalarT>::vector(const _e<T>& a) {
    meta::EFOR2<0, meta::Less, vpvector_count, +1,
        _copy_vpvector<scalarT, vector<scalarT>, const _e<T>&>
      >::exec(*this, a);
  }
template<typename scalarT> template<typename T> vector<scalarT>&
  vector<scalarT>::operator=(const _e<T>& a) {
    meta::EFOR2<0, meta::Less, vpvector_count, +1,
        _copy_vpvector<scalarT, vector<scalarT>, const _e<T>&>
      >::exec(*this, a);
    return *this;
  }
```

Instead of a single `operator+`, four different versions are required (Blinn 2000). These cater for all possible permutations of adding a `vvm::vector` and an expression. For ternary operators, eight different versions would be required. Langer & Kreft (2003) needed to use only one `operator+`, because the arguments are automatically promoted to `doubles` during the addition. Since type conversion is undesirable for vector programs, this technique is not suitable. Note the use of `leftT::scalar_type` to select the appropriate `_add` specialisation.

```
#define VaV _binary_expr<_add<scalarT>, \
  _e<vector<scalarT> >, _e<vector<scalarT> > >
#define EaV _binary_expr<_add<scalarT>, \
  _e<leftT>           , _e<vector<scalarT> > >
#define VaE _binary_expr<_add<scalarT>, \
  _e<vector<scalarT> >, _e<rightT>           >
#define EaE _binary_expr<_add<typename leftT::scalar_type>, \
  _e<leftT>           , _e<rightT>           >
template<typename scalarT> inline const _e<VaV>
operator+(const vector<scalarT>& a, const vector<scalarT>& b) {
  return _e<VaV>(VaV(a, b));
}
template<typename scalarT, typename leftT> inline const _e<EaV>
operator+(const _e<leftT>& a, const vector<scalarT>& b) {
```

```
  return _e<VaE>(VaE(a, b));
}
template<class scalarT, typename rightT> inline const _e<VaE>
operator+(const vector<scalarT>& a, const _e<rightT>& b) {
   return _e<VaE>(VaE(a, b));
}
template<typename leftT, typename rightT> inline const _e<EaE>
operator+(const _e<leftT>& a, const _e<rightT>& b) {
  return _e<EaE>(EaE(a, b));
}
```

The template _add represents the addition operation. It is the class that actually performs the addition. The template _add can be implemented as follows:

```
// Scalar version
template<typename scalarT> struct _add {
  typedef scalarT scalar_type;
  typedef const typename scalar_traits<scalarT>::vpvector_type
    return_type;
  typedef typename scalar_traits<scalarT>::vpvector_type
    vpvector_type;
  struct do_add {
    template<int i> struct Code {
      static inline void
        exec(return_type& ret, const vpvector_type& a,
             const vpvector_t& b) {
          reinterpret_cast<scalar_type*>(&ret)[i] =
            reinterpret_cast<const scalar_type*>(&a)[i] +
            reinterpret_cast<const scalar_type*>(&b)[i];
        }
    };
  };
  static inline return_type
  evaluate(const vpvector_type& a, const vpvector_type& b) {
    return_type ret;
    meta::EFOR3<0, meta::Less,
      vpvector_traits<vpvector_type>::scalar_count,
      +1, do_add>::exec(ret, a, b);
    return ret;
  }
```

```
};

// VPU vector version
template<typename scalarT> struct _add {
  typedef scalarT scalar_type;
  typedef const typename scalar_traits<scalarT>::vpvector_type
    return_type;
  typedef typename scalar_traits<scalarT>::vpvector_type
    vpvector_type;
  static inline return_type
  evaluate(const vpvector_type& a, const vpvector_type& b) {
    return vec_add(a, b);
  }
};
```

The `_add` implementation described above will fail to compile currently if both scalar and VPU vector versions are included at the same time. Switching between the emulation layer and the active vector-processor implementation automatically is covered later in Section 7.4.

Blitz++

To create a version based on Blitz++, `vvm::vector` was derived from Blitz++'s `TinyVector`. Since it derives from `TinyVector`, there is no need to keep VPU vector instances anymore. `TinyVector`'s `operator[]` provides access to VPU vectors. Scalars can be accessed by retrieving the first VPU vector and performing some pointer arithmetic. Note that the data in `TinyVector` could not be used directly since it is a private data member and that the scalar access methods assume that the data allocated by `TinyVector` is contiguous.

```
template<typename scalarT> class vector
: public blitz::TinyVector<
  typename scalar_traits<scalarT>::vpvector_type,
  vector_traits<vector<scalarT> >::vpvector_count
> {
  // Other normal vvm::vector stuff goes here
public:
  inline vpvector_type& vpvector(const int i) {
    return operator[](i);
  }
```

```
  inline const vpvector_type vpvector(const int i) const {
    return operator[](i);
  }
  inline scalar_type& scalar(const int i) {
    return (reinterpret_cast<scalarT*>(&vpvector(0)))[i];
  }
  inline const scalar_type scalar(const int i) const {
    return (reinterpret_cast<const scalarT*>(&vpvector(0)))[i];
  }
public:
  template<class P_expr>
  inline vector(blitz::_bz_VecExpr<P_expr> expr) {
    _bz_assign(expr, blitz::_bz_update<T_numtype,
      _bz_typename P_expr::T_numtype>());
  }
  template<class P_expr>
  inline vector& operator=(blitz::_bz_VecExpr<P_expr> expr) {
    _bz_assign(expr, blitz::_bz_update<T_numtype,
       _bz_typename P_expr::T_numtype>());
    return *this;
  }
};
```

Inheriting from `TinyVector` is all that VVM needs when it runs without a VPU. When a VPU, like AltiVec, is active, instead of creating an object that represents an addition and calling `vec_add`, this version wraps AltiVec types in a wrapper class and overloads `operator+`. The wrapper class is required because operator overloading of AltiVec vectors is not allowed. The wrapper and its `operator+` can be implemented as follows:

```
template<typename T> struct _wrapper {
  T value;
};
typedef _wrapper<__vector signed char> _vpvector_schar;
// Insert _wrapper typedefs for other AltiVec types here
inline const _vpvector_schar
operator+(const _vpvector_schar a, const _vpvector_schar b) {
  return vec_add(a.value, b.value);
}
// Insert operator+ for other types here
```

Expression templates based on Blinn (2000)'s paper

The hand-coded version based on Blinn (2000)'s paper is almost identical to the hand-coded version based on Veldhuizen (1995*b*)'s paper. The major difference is that Blinn (2000) did not use a class that represents a binary expression.

The `operator+` definitions would therefore look like the following:

```
#define VaV \
  _add<scalarT, _e<vector<scalarT> >, _e<vector<scalarT> > >
#define EaV \
  _add<scalarT, _e<leftT>           , _e<vector<scalarT> > >
#define VaE \
  _add<scalarT, _e<vector<scalarT> >, _e<rightT> >
#define EaE \
  _add<typename leftT::scalar_type, _e<leftT>, _e<rightT> >
template<typename scalarT> inline const _e<VaV>
operator+(const vector<scalarT>& a, const vector<scalarT>& b) {
  return _e<VaV>(VaV(a, b));
}
template<typename scalarT, typename leftT> inline const _e<EaV>
operator+(const _e<leftT>& a, const vector<scalarT>& b) {
  return _e<EpV>(EpV(a, b));
}
template<typename scalarT, typename rightT> inline const _e<VaE>
operator+(const vector<scalarT>& a, const _e<rightT>& b) {
  return _e<VaE>(VaE(a, b));
}
template<typename leftT, typename rightT> inline const _e<EaE>
operator+(const _e<leftT>& a, const _e<rightT>& b) {
  return _e<EaE>(EaE(a, b));
}
```

Without the binary expression class, the addition class needs to handle the arguments itself.

```
template<typename scalarT, typename leftT, typename rightT>
class _add {
  const leftT _left;
  const rightT _right;
public:
  typedef typename scalar_traits<scalarT>::vpvector_type
```

```
    return_type;
  inline _add(const leftT& l, const rightT& r)
  : _left(l), _right(r) {
  }
  inline return_type vpvector(const int i) const {
#if defined VVM_ALTIVEC
    return vec_add(_left.vpvector(i), _right.vpvector(i));
#else
    return _left.vpvector(i) + _right.vpvector(i);
#endif
  }
};
```

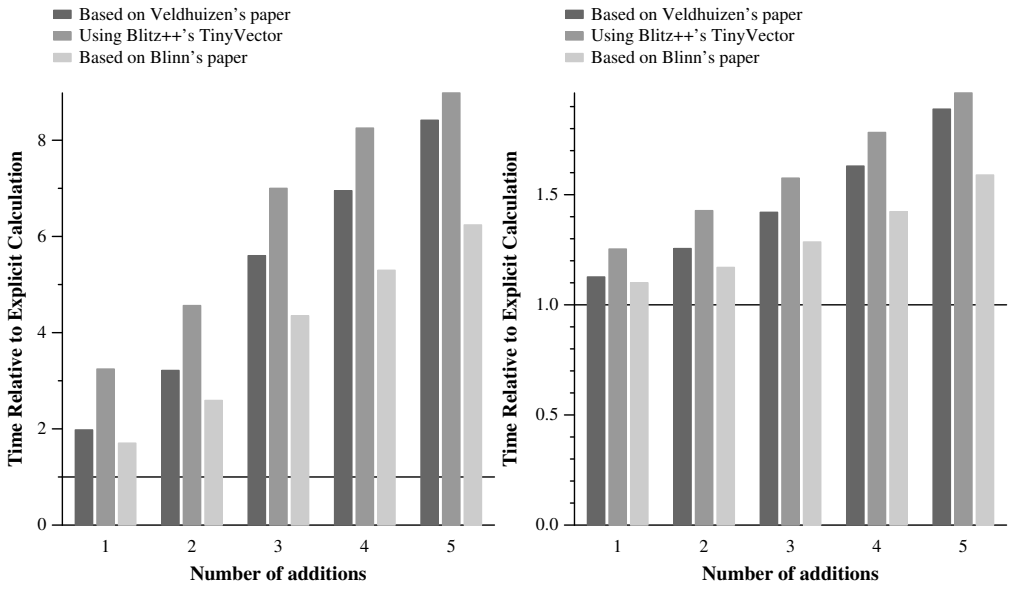
Results

Figures 7.4 and 7.5 show that in general the version based on Blinn (2000)'s paper was fastest, followed by the version based on Veldhuizen (1995*b*)'s paper, and lastly Blitz++'s `TinyVector`. The version based on Veldhuizen (1995*b*)'s paper was within 20% slower than a hand-coded program when the number of elements was four or sixteen for expressions with one to five additions. For the same kind of expressions, `vvm::vectors` with a single scalar were about 8 times slower. This behaviour is consistent with the trend shown by Veldhuizen (1995*b*) where the expression templates perform badly on short vectors but quickly become comparable to the hand-coded C version with increasing vector sizes. As expected, the hand-coded expression templates were faster than Blitz++, probably because the hand-coded versions unrolled the for-loop using template metaprogramming.

Blinn (2000) states that expression templates were faster than the hand-coded version for simple expressions to about zero-cost for more complex expressions, for a vector that had only three elements. According to Blinn (2000), this was because the expression templates removed more temporaries and used registers more efficiently. Veldhuizen (1995*b*)'s implementation however only approached zero-cost for vectors with about a 1,000 elements. However, it should be noted that Veldhuizen (1995*b*)'s implementation is for long vectors and uses iterators to iterate through the elements in the vector. VVM's expression templates speedup should therefore be similar to Blinn (2000)'s implementation. The results obtained however do not correlate with the results from Blinn (2000). Blinn (2000)'s expression templates were faster than hand-coded programs for simple expressions, and slightly slower for more complex expressions, for a vector that has only three elements. One possible reason for the difference is the compiler. Unfortunately, Blinn (2000) did not specify the compiler that he used.

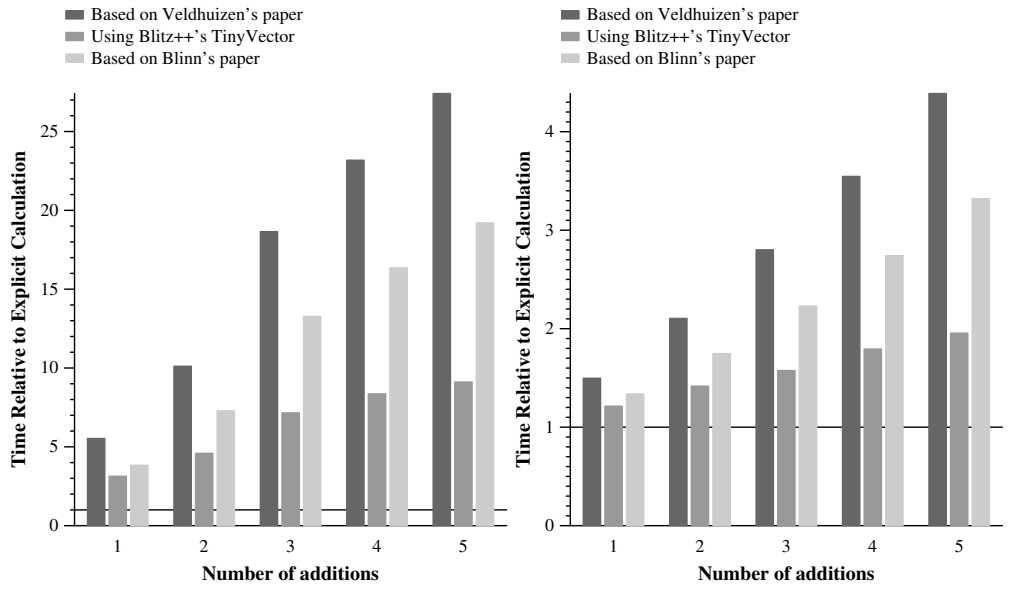
Expression templates cannot be used to create a VVM implementation that has no significant overheads when compiled with Apple GCC 3.1 20021003 or Apple GCC 3.3

Figure 7.4 Performance of `operator+` based on expression templates, when operating on `vvm::vectors` with a single element



(a) 1 scalar per vvm::vector (GCC 3.1)

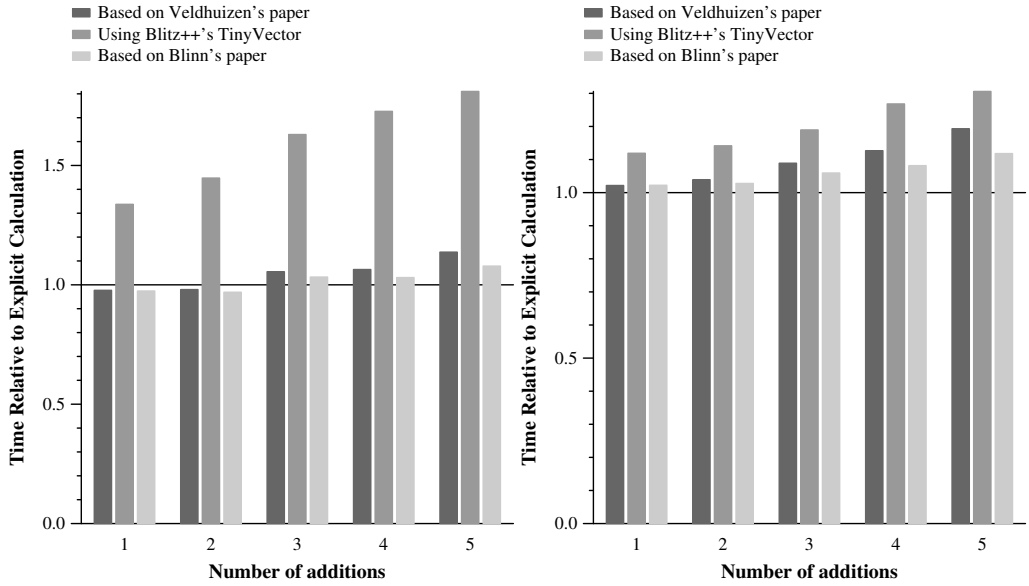
(b) 1 vp::vector per vvm::vector (GCC 3.1)



(c) 1 scalar per vvm::vector (GCC 3.3)

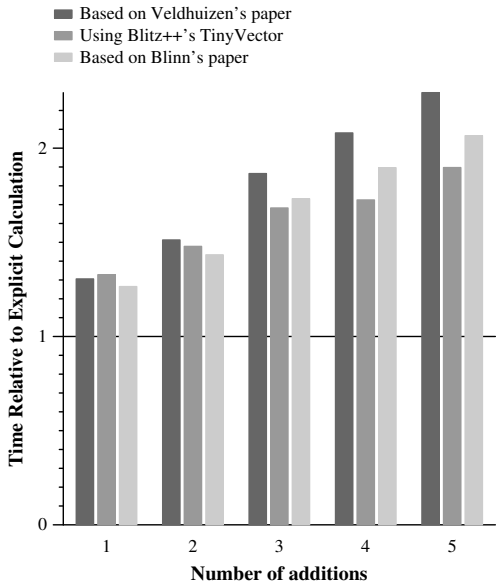
(d) 1 vp::vector per vvm::vector (GCC 3.3)

Figure 7.5 Performance of `operator+` based on expression templates, when operating on `vvm::vector<int>` with a constant scalar count of 16

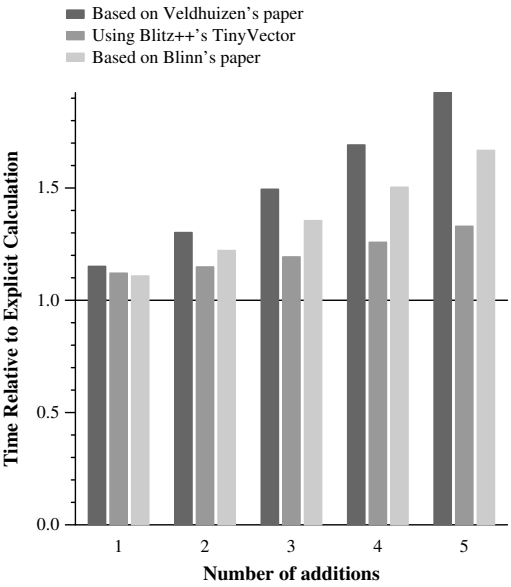


(a) 16 scalars per vvm::vector (GCC 3.1)

(b) 4 vp::vectors per version (GCC 3.1)



(c) 16 scalars per vvm::vector (GCC 3.3)



(d) 4 vp::vectors per version (GCC 3.3)

20030304. An implementation based on expression templates has particularly significant overheads when processing `vvm::vectors` that have only a single element. As the number of elements in a `vvm::vector` increases, the overheads decrease. This behaviour is the opposite of function overloading. An implementation based on function overloading has no significant overheads when processing `vvm::vectors` with a single element. As the number of elements in a `vvm::vector` increases, the overheads increase.

7.4 Switching between the emulation layer and the active vector-processor implementation

Up until now, we have been providing two different implementations, one scalar and one VPU vector, for the same operation. The scalar version is part of the emulation layer while the VPU vector version is the active vector-processor implementation version. While preprocessor macros appear to be the most appropriate technique for determining whether a function should be executed by the emulation layer or the active vector-processor implementation, in VVM's case, the preprocessor cannot make this choice because the exact mapping between VPU vectors and scalars is not known until compile-time. Instead of using the preprocessor, template specialisation can be used to select between different implementations automatically.

When using template specialisation, the emulation layer is usually the general template and vector-processor implementations specialise this general template. This way, anything that is not specialised is automatically handled by the emulation layer, which is exactly the required behaviour. The most obvious way to implement this is to specialise the appropriate VVM function as follows:

```
// Vectorised Addition available for ints
template<> vector<int>
operator+(const vector<int>& lhs, const vector<int>& rhs) {
  vector<int> ret;
  for(int i = 0;
      i < vector_traits<vector<int> >::vpvector_count;
      ++i) {
    ret.vpvector(i) = vec_add(lhs.vpvector(i), rhs.vpvector(i));
    }
  return ret;
}
// Scalar Addition for doubles
template<> vector<double>
operator+(const vector<double>& lhs, const vector<double>& rhs) {
```

```
  vector<double> ret;
  for(int i = 0;
      i < vector_traits<vector<double> >::scalar_count;
      ++i) {
    ret.scalar(i) = lhs.scalar(i) + rhs.scalar(i);
  }
}
```

The above approach fails for the following reason: since the scalar-to-VPU vector mapping is unknown at programming-time, the programmer cannot be expected to know what scalar types to specialise. For example, if `long` is of the same length as `int`, then it can also use vectorised addition. However, since the size of `long` is not known until compile-time, the programmer cannot decide whether it should be a vector or scalar addition.

Four different switching methods are discussed — function, operator, template and enabler switching. All programs were compiled using Apple GCC 3.1 20021003 and Apple GCC 3.3 20030304, with the `-Os` (optimise for size) and timed 20 times. The lowest values were used as the representative. Note that for brevity, none of the techniques used in Section 7.3 to speed up the execution are used.

7.4.1 Function switching

Instead of specialising functions in the VVM specification, vector-processor implementations can use function templates which accept a VPU vector type as a template parameter. This extra function call provides an additional level of indirection. The emulation layer would provide the general function template, while vector-processor implementations would specialise the function template for their VPU vectors. Note that using a scalar as a template parameter does not help since it runs into the same problems as specialising the VVM functions directly. VPU vector types will be used instead. The VVM addition function therefore becomes something like the following:

```
template<typename scalarT> vector<scalarT>
operator+(const vector<scalarT>& lhs, const vector<scalarT>& rhs) {
vector<scalarT> ret;
  for(int i = 0;
      i < vector_traits<vector<scalarT> >::vpvector_count;
      ++i) {
    ret.vpvector(i) = _add(lhs.vpvector(i), rhs.vpvector(i));
  }
  return ret;
}
```

```
// Emulation layer
template<typename vpvectorT> vpvectorT
_add(const vpvectorT a, const vpvectorT b) {
  typedef typename vpvector_traits<vpvectorT>
    ::scalar_type scalar_type;
  vpvectorT ret;
  for(int i = 0;
      i < vpvector_traits<vpvectorT>::scalar_count;
      ++i) {
    reinterpret_cast<scalar_type*>(&ret)[i] =
      reinterpret_cast<scalar_type*>(&a)[i] +
      reinterpret_cast<scalar_type*>(&b)[i];
  }
  return ret;
}
// AltiVec vector-processor implementation
template<> __vector signed int
_add(const __vector signed int a, const __vector signed int b) {
  return vec_add(a, b);
}
// Scalar processor implementation
template<> int _add(const int a, const int b) {
  return a + b;
}
```

There are now three different versions of `_add` that operate on different VPU vector types. A vector-processor implementation switching technique would have at least the emulation layer and the AltiVec versions.

1. Emulation layer

   Since the emulation layer might be used both whether a VPU is available or not, the addition must support both scalar and VPU vector scalar types. To support both scalar and VPU modes, the addition operation treats the element type as VPU vectors, but perform its operations using scalar operations.

2. AltiVec vector-processor implementation

   The AltiVec vector-processor implementation uses AltiVec commands (in this case, `vec_add`) to perform its operations, and would be specialised for applicable AltiVec vectors only.

3. Scalar processor implementation

   The scalar processor implementation is not a necessity because the emulation layer uses the scalar processor. The scalar processor implementation however can increase the performance of VVM in scalar mode.

7.4.2 Operator switching

Instead of trying to adding an extra level of indirection, `operator+` can be used directly. In this case, scalars call `operator+` directly, while VPU vectors need to be wrapped and have `operator+` defined. VPU vectors need to be wrapped because C++ does not allow operators to be overloaded for base types.

The following code shows how operator switching can be applied to the addition operator, implemented using function overloading:

```
template<typename scalarT> inline const vector<scalarT>
operator+(const vector<scalarT>& lhs, const vector<scalarT>& rhs) {
  vector<scalarT> ret;
  meta::EFOR3<0, meta::Less,
    vector_traits<vector<scalarT> >::vpvector_count,
    +1, _add<scalarT> >::exec(ret, lhs, rhs);
  return ret;
}
// Scalar implementation
template<typename scalarT> struct _add {
  template<int i> struct Code {
    static inline void
    exec(vector<scalarT>& ret, const vector<scalarT>& a,
         const vector<scalarT>& b) {
      ret.vpvector(i) = a.vpvector(i) + b.vpvector(i);
    }
  };
};
// Wrapper class for VPU vectors
template<typename T> struct _wrapper {
  T value;
  inline _wrapper() {}
  inline _wrapper(const T& v) : value(v) {}
};
// AltiVec implementation
typedef _wrapper<__vector signed int> vpvector_sint;
```

```
inline const vpvector_sint
operator+(const vpvector_sint a, const vpvector_sint b) {
  return vpvector_sint(vec_add(a.value, b.value));
}
// Emulation layer
template<typename vpvectorT> inline const _wrapper<vpvectorT>
operator+(const _wrapper<vpvectorT> a,
          const _wrapper<vpvectorT> b) {
  typedef typename vpvector_traits<_wrapper<vpvectorT> >
    ::scalar_type scalar_type;
  _wrapper<vpvectorT> ret;
  for(int i = 0;
      i < vpvector_traits<_wrapper<vpvectorT> >::scalar_count;
      ++i) {
    reinterpret_cast<scalar_type*>(&ret)[i] =
      reinterpret_cast<const scalar_type*>(&a)[i] +
      reinterpret_cast<const scalar_type*>(&b)[i];
  }
  return ret;
}
```

In this version, the general `_add` template is actually the scalar processor implementation. AltiVec and emulation functions will pass through this function. AltiVec specialisations are provided by specialising `operator+` for the appropriate VPU vector wrapper type. The emulation layer is found in the `operator+` for the wrapper with an unknown VPU vector type. In this implementation, all vector-processor implementations would have to use the same wrapper type.

Figures 7.6 and 7.7 show that this version does not incur any additional overheads. However, this method makes it difficult to provide custom behaviour for scalar types. For example, it would be difficult to provide a `operator%` for float types if needed. While we can define an `operator%` for float types if we wrapped the scalar types, wrapping can make it more difficult for the user to use the scalars or VPU vectors in a `vvm::vector` directly. In particular, the scalars returned via the `scalar()` access method would not behave identically to the scalar type of the `vvm::vector`, which would be confusing.

7.4.3 Template switching

In this version, we add a VPU vector type as a template parameter to the addition class. Despite being able to derive a scalar type from a VPU vector type, the scalar type is still passed to the addition class as a template parameter, because the VPU vector-to-scalar

mapping is a one-to-many relationship. Thus the addition class will not know exactly what `vvm::vector` types to accept.

The following code shows how template switching can be applied to the addition operator, implemented using function overloading:

```
template<typename scalarT> const vector<scalarT>
operator+(const vector<scalarT>& a, const vector<scalarT>& b) {
  typedef typename vector_traits<vector<scalarT> >
    ::vpvector_type vpvector_t;
  vector<scalarT> ret;
  meta::EFOR3<0, meta::Less,
    vector_traits<vector<scalarT> >::vpvector_count,
    +1, _add<scalarT, vpvector_t> >::exec(ret, a, b);
  return ret;
}
// Emulation
template<typename scalarT, typename vpvectorT> struct _add {
  template<int i> struct Code {
    static void exec(vector<scalarT>& dest,
    const vector<scalarT>& a, const vector<scalarT>& b) {
      for(int j = 0;
          j < vpvector_traits<vpvectorT>::scalar_count;
          ++j) {
        reinterpret_cast<scalarT*>(&dest.vpvector(i))[j] =
          reinterpret_cast<const scalarT*>(&a.vpvector(i))[j] +
          reinterpret_cast<const scalarT*>(&b.vpvector(i))[j];
      }
    }
  };
};
// AltiVec version
template<typename scalarT>
struct _add<scalarT, __vector signed int> {
  template<int i> struct Code {
    static void exec(vector<scalarT>& dest,
    const vector<scalarT>& a, const vector<scalarT>& b) {
      dest.vpvector(i) = vec_add(a.vpvector(i), b.vpvector(i));
    }
  };
};
```

```
// Scalar version
template<typename scalarT> struct _add<scalarT, int> {
  template<int i> struct Code {
    static void exec(vector<scalarT>& dest,
    const vector<scalarT>& a, const vector<scalarT>& b) {
      dest.vpvector(i) = a.vpvector(i) + b.vpvector(i);
    }
  };
};
```

The general `_add` class is the emulation layer's version, while `_add<scalarT, int>` and `_add<scalarT, __vector signed int>` are the scalar processor implementation's and AltiVec vector-processor implementation's versions respectively. This example illustrates how you can use the VPU vector type to determine the appropriate specialised class instead of the scalar type.

While template switching provides good performance and avoids wrapping, specialising operations for template switching can be tiresome because each type must be specialised individually.

7.4.4 Enabler switching

Enabler switching uses enablers to decide which version of the addition operation will be invoked. Using enablers allow us to specialise the operation on `vvm::vector` instead of the operation on VPU vector. Enablers make it easier to specialise for a number of types simultaneously, make it easier to provide specialisations for operations that do not apply the same operation to all the VPU vectors, and allow the emulation layer to operate on scalars. The other switching techniques discussed require the emulation layer to operate on VPU vectors because the specialisations are applied to the addition of VPU vectors and not to the addition of `vvm::vectors`. Furthermore, because the emulation layer operates on scalars, there is no need to create a scalar processor implementation version.

The following code shows how enabler switching can be applied to the addition operator, implemented using function overloading.

```
namespace priv {
  // Emulation layer
  template<typename scalarT, typename specializedT = void>
  struct add {
    static vector<scalarT>
    exec(const vector<scalarT>& a, const vector<scalarT>& b) {
      // Perform Emulation addition here
```

```
    }
  };
  // AltiVec vector-processor implementation
  template<typename scalarT> struct add<scalarT,
    typename ct::enable_if<
      ct::contains<
        altivec_types,
        typename scalar_traits<scalarT>::vpvector_type
      >::value
    >::type
  > {
    static vector<scalarT>
    exec(const vector<scalarT>& a, const vector<scalarT>& b) {
      // Perform AltiVec addition here
    }
  };
} // End of priv namespace
template<typename scalarT>
vector<scalarT>
operator+(const vector<scalarT>& a, const vector<scalarT>& b) {
  return priv::add<scalarT>::exec(a, b);
}
```

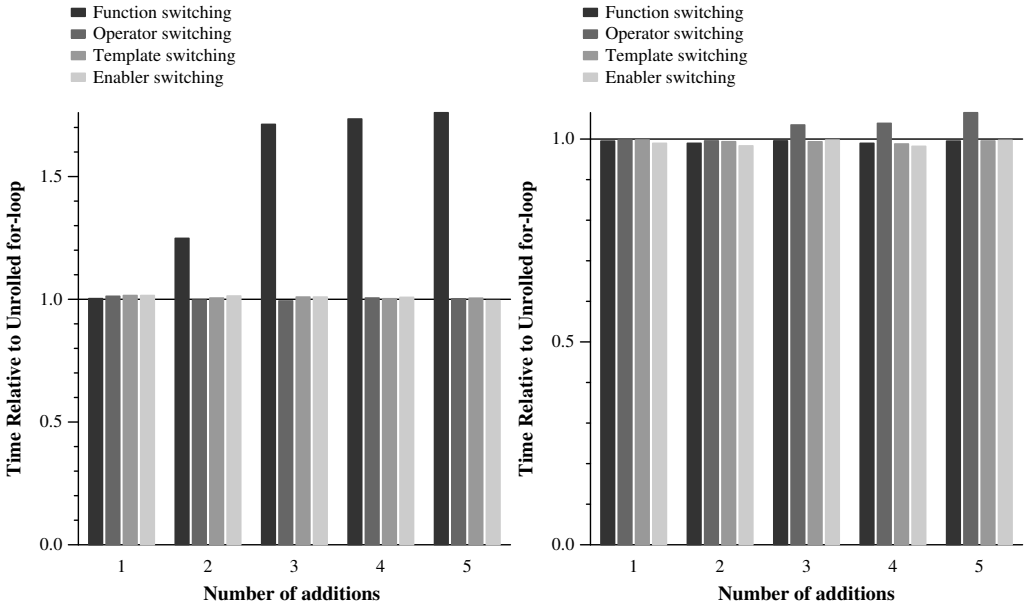
It is also possible to specialise on other parameters and to make more complex decisions. For example, it is easy to specialise the `prefetch` operation only for channels `0` to `3` for AltiVec.

7.4.5 Results

Figure 7.6 shows the cost of switching between the emulation layer and the active vector-processor implementation when operating on `vvm::vectors` with only one element. Figures 7.6(a) and 7.6(b) are for Apple GCC 3.1 20021003 in scalar and AltiVec mode respectively. Figures 7.6(c) and 7.6(d) are for Apple GCC 3.3 20030304 in scalar and AltiVec mode respectively.

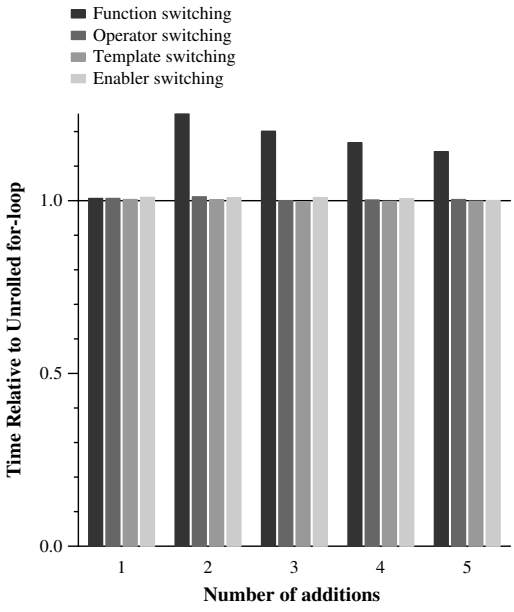
Figures 7.6 and 7.7 show that, on both Apple GCC 3.1 20021003 and Apple GCC 3.3 20030304, operator, template and enabler switching incur no significant overheads. Function switching incurred no significant overheads except when applied to `vvm::vectors` with a single element in scalar mode. In such situations, a program based on function switching was at worst 80% and 30% slower than a program without any switching when Apple GCC 3.1 20021003 and Apple GCC 3.3. 20030304 were used respectively.

Figure 7.6 Cost of switching between the emulation layer and the active vector-processor implementation when operating on `vvm::vectors` with a single element

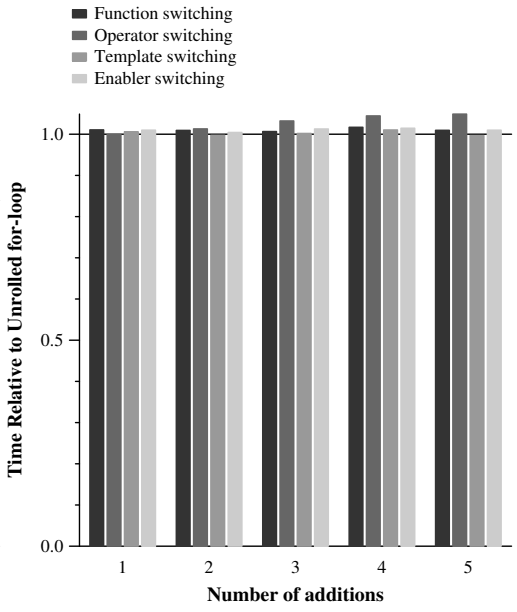


(a) 1 scalar per vvm::vector (GCC 3.1)

(b) 1 vp::vector per vvm::vector (GCC 3.1)

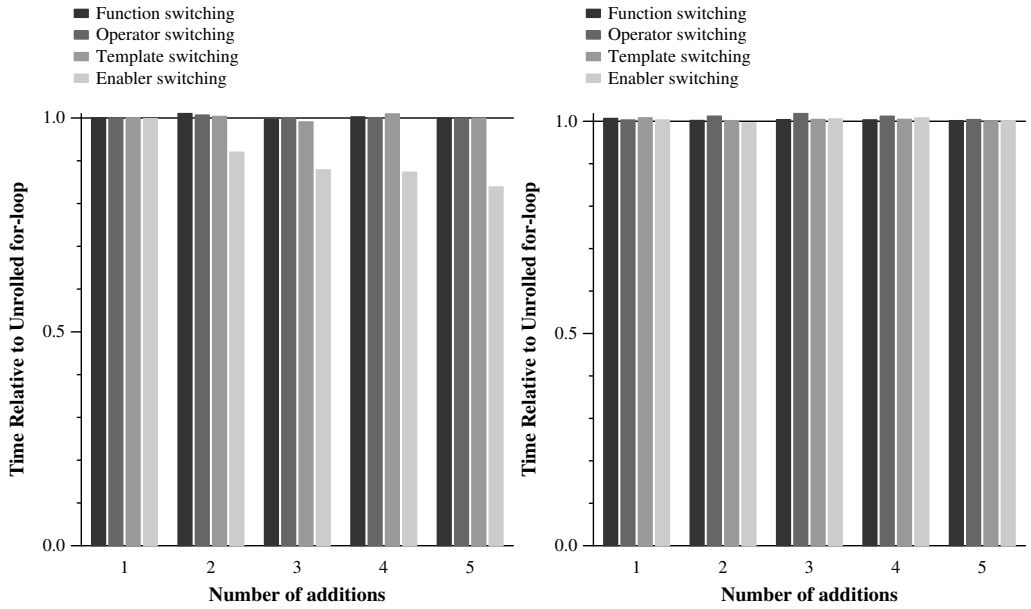


(c) 1 scalar per vvm::vector (GCC 3.3)



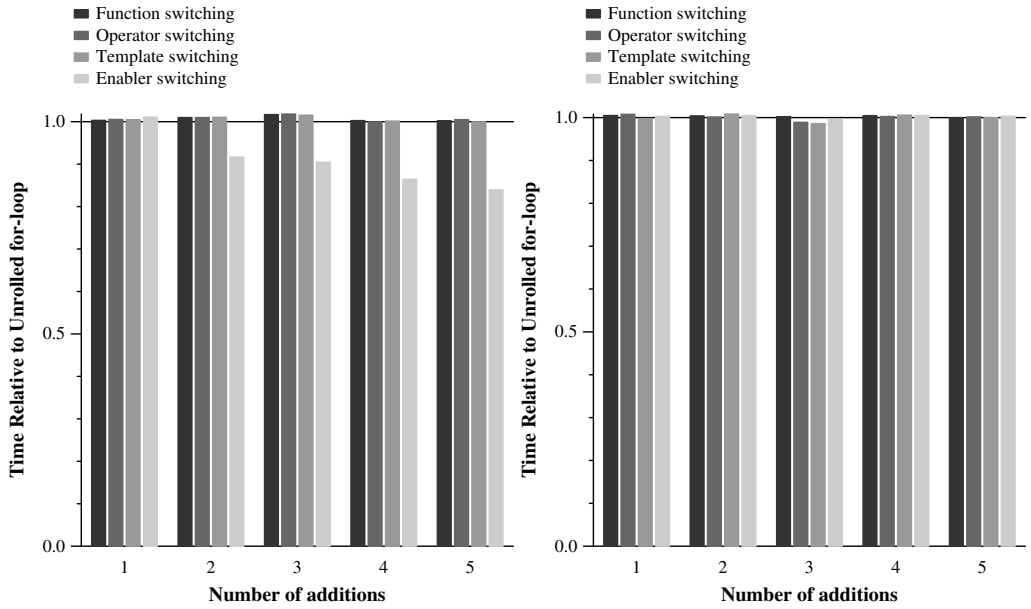
(d) 1 vp::vector per vvm::vector (GCC 3.3)

Figure 7.7 Cost of switching between the emulation layer and the active vector-processor implementation when operating on `vvm::vector<int>` with a constant scalar count of 16



(a) 16 scalars per vvm::vector<int> (GCC 3.1)

(b) 4 vp::vectors per vvm:vector<int> (GCC 3.1)



(c) 16 scalars per vvm::vector<int> (GCC 3.3)

(d) 4 vp::vectors per vvm:vector<int> (GCC 3.3)

7.5 Type conversions

From Chapter 6, type conversions involve performing a `static_cast` on each of the scalars in the `vvm::vector`. Thus, a scalar version of the `vector_cast` function can be implemented as follows:

```
template<typename toVectorT, typename fromVectorT>
toVectorT vector_cast(const fromVectorT& from) {
  typedef vector_traits<toVectorT>::scalar_type to_scalar_t;
  toVectorT ret;
  for(int i = 0;
      i < vector_traits<fromVectorT>::scalar_count;
      ++i) {
    ret.scalar(i) = static_cast<to_scalar_t>(from.scalar(i));
  }
  return ret;
}
```

Unlike operations like `operator+`, which were the focus of Section 7.3 and apply the appropriate operation to VPU vectors, `vector_cast` should be applied to `vvm::vectors`, because different VPU vectors can contain different numbers of scalars.

It is easy to provide an appropriate specialisation using enabler switching from Section 7.4.4. The following example shows how to use AltiVec functions to convert from a `__vector float` to a `__vector unsigned int`.

```
namespace priv {
  template<typename toScalarT, typename fromScalarT>
  struct vector_cast<vector<toScalarT>, vector<fromScalarT>,
    typename ct::enable_if<
      boost::is_same<
        typename scalar_traits<fromScalarT>::vpvector_type,
        __vector float>::value &&
      boost::is_same<
        typename scalar_traits<toScalarT>::vpvector_type,
        __vector unsigned int>::value
    >::type
  > {
    inline static vector<toScalarT>
    exec(const vector<fromScalarT>& a) {
      vector<toScalarT> ret;
      for(int i = 0;
```

```
          i < vector_traits<vector<toScalarT> >::vpvector_count;
          ++i) {
        ret.vpvector(i) = vec_ctu(a.vpvector(i), 0);
      }
      return ret;
    }
  };
}
template<typename toVectorT, typename fromVectorT>
toVectorT vector_cast(const fromVectorT& from) {
  return priv::vector_cast<toVectorT, fromVectorT>::exec(from);
}
```

The `priv::vector_cast` template is only enabled if the VPU vector type for the source `vvm::vector` is `__vector float` and the VPU vector type for the destination `vvm::vector` is `__vector unsigned int`. The `exec` function is the one that actually performs the conversion.

For clarity, `for` loops were used. As usual, the `for` loops shown in this section can be replaced with `meta::EFOR` which will unroll the loop for the compiler. In addition, it is good to provide a specialised version for cases where `fromVectorT` and `toVectorT` are equivalent, to reduce the costs of casting a type to itself.

7.6 Functors

The functors required by VVM are easy to implement once traits are implemented. The `equal_to` function object, for example, can be implemented in the following manner.

```
template<typename scalarT> struct equal_to
: public binary_function<
  vector<scalarT>, vector<scalarT>,
  vector<scalar_traits<scalarT>::bool_type>
> {
  vector<scalar_traits<scalarT>::bool_type>
  operator()(const vector<scalarT>& lhs,
             const vector<scalarT>& rhs) const {
    return lhs == rhs;
  }
};
```

The other comparison functors can be implemented in the same manner as `equal_to`.

Table 7.1 Contents of a `vvm::vector` in AltiVec mode and in Scalar Mode

	AltiVec Mode (Constant scalar count is 16)	Scalar Mode (Constant scalar count is 1)
	Contents of `vvm::vector`	Contents of `vvm::vector`
`char`	1 `__vector unsigned char`	1 `signed char`
	or 1 `__vector signed char`	or 1 `unsigned char`
`signed char`	1 `__vector signed char`	1 `signed char`
`unsigned char`	1 `__vector unsigned char`	1 `unsigned char`
`short int`	2 `__vector signed shorts`	1 `short int`
`unsigned short int`	2 `__vector unsigned shorts`	1 `unsigned short int`
`int`	4 `__vector signed ints`	1 `int`
`unsigned int`	4 `__vector unsigned ints`	1 `unsigned int`
`long int`	4 `__vector signed ints`	1 `long int`
`unsigned long int`	4 `__vector unsigned ints`	1 `unsigned long int`
`float`	4 `__vector floats`	1 `float`
`double`	16 `doubles` (No appropriate VPU vector)	1 `double`

7.7 Performance measurement tool results

In order for an abstract VPU implementation to have no overheads, both the actual execution of the operation, and the switching between the emulation layer and the active vector-processor implementation should have no overheads. Sections 7.3 and 7.4 evaluated these overheads separately. In this section, performance measurement tool results from two implementations of VVM are presented. These results combine both the execution and switching costs.

The performance tool evaluates the cost of adding two to six `vvm::vectors`. The addition expression is repeated a fixed number of times, and the total time taken is recorded. Twenty total times were recorded with the same input, and the lowest time was taken as the representative.

Tables 7.2 and 7.3 show the results of the performance tool on two different VVM implementations: function overloading with enabler switching, and expression templates based on Veldhuizen (1995*b*)'s paper with function overloading. The tables are divided into two sections, "AltiVec Mode" and "Scalar Mode", which refers to the mode that VVM is running in. For each mode, there are five columns, labelled 1 to 5, which refer to the number of addition operations in a single expression. Columns A and B contain the number of VPU vectors and scalars in a `vvm::vector` respectively. Table 7.1 shows the number of elements and their types in each `vvm::vector` in AltiVec and Scalar mode. For example, from Table 7.1, `vvm::vector<int>` consists of 4 `__vector signed ints` or 1 `int` in AltiVec and scalar modes respectively. Since there is no AltiVec addition function for `doubles`, the table does not show the percentage difference for `doubles` in AltiVec mode. When asked to add vvm::vectors of `doubles`, which has 16 scalars in AltiVec mode, VVM uses the scalar processor.

Table 7.2 Percentage difference between a VVM implementation, which used function overloading (Unrolled for-loop) and enabler switching, and a hand-coded program for expressions involving one to five additions. Values less than 0 indicate that the VVM implementation was faster. Refer to Table 7.1 for the number of elements in each `vvm::vector`.

	AltiVec Mode						Scalar Mode					
	A	1	2	3	4	5	B	1	2	3	4	5
`char`	1	-17.0	-16.2	-15.3	-16.7	-15.8	1	-16.1	-13.8	-16.0	-14.8	-12.9
`signed char`	1	0.0	-2.3	-0.6	-2.5	-0.5	1	-0.7	-2.3	-1.0	-2.3	-2.4
`unsigned char`	1	-0.9	-1.8	-0.9	-1.0	-0.7	1	-1.2	-1.2	-0.7	-0.9	-0.6
`short int`	2	**10.8**	12.3	14.1	14.4	17.1	1	-12.2	-9.2	-10.6	-9.4	-9.3
`unsigned short int`	2	16.2	15.2	17.2	17.7	**19.7**	1	0.3	-0.7	-0.8	-1.1	-1.2
`int`	4	**13.7**	13.9	15.4	16.9	16.9	1	-8.7	-8.9	-7.9	-8.2	-7.0
`unsigned int`	4	16.3	16.6	19.7	18.4	23.0	1	-0.2	-0.5	0.4	-1.9	-0.2
`long int`	4	18.5	17.7	18.6	19.5	**23.6**	1	-1.3	-0.4	-1.5	-0.3	-2.1
`unsigned long int`	4	16.9	18.8	19.3	19.1	20.9	1	-0.6	-0.4	0.1	-1.2	-1.2
`float`	4	18.7	20.1	19.4	15.5	13.8	1	-1.4	**0.9**	-0.9	-0.4	-0.1
`double`	16	N/A	N/A	N/A	N/A	N/A	1	-4.1	-4.7	-4.5	-3.1	-3.4

Table 7.3 Percentage difference between a VVM implementation, which used expression templates (Based on Veldhuizen's paper) and function switching, and a hand-coded program for expressions involving one to five additions. Values less than 0 indicate that the VVM implementation was faster. Refer to Table 7.1 for the number of elements in each `vvm::vector`.

	AltiVec Mode						Scalar Mode					
	A	1	2	3	4	5	B	1	2	3	4	5
`char`	1	-2.5	9.1	26.0	45.7	71.3	1	128.1	315.2	613.9	793.3	958.1
`signed char`	1	13.0	27.1	46.4	70.0	98.7	1	167.1	358.5	744.2	920.6	**1101.8**
`unsigned char`	1	12.0	26.7	46.1	68.2	98.7	1	154.7	335.2	700.7	874.9	1046.0
`short int`	2	**0.9**	6.7	15.6	27.3	38.8	1	78.5	200.2	414.0	544.0	685.0
`unsigned short int`	2	4.1	10.7	19.2	30.2	**43.2**	1	97.9	238.4	459.9	603.1	762.5
`int`	4	**-1.7**	1.7	5.3	8.6	17.2	1	44.8	128.3	232.2	338.5	438.9
`unsigned int`	4	1.8	4.9	9.5	13.8	20.6	1	56.4	134.7	257.6	361.8	463.1
`long int`	4	1.4	4.4	8.8	13.4	19.2	1	54.9	147.2	268.9	373.3	475.6
`unsigned long int`	4	1.5	3.6	8.5	15.4	19.5	1	58.0	133.5	259.5	365.0	468.3
`float`	4	1.9	4.7	11.5	14.2	**21.1**	1	66.3	123.0	193.6	241.6	291.9
`double`	16	N/A	N/A	N/A	N/A	N/A	1	30.1	61.6	102.4	137.9	177.4

From Sections 7.3.4 and 7.4, a VVM implementation that uses function overloading and enabler switching is expected to have no significant overheads when the number of elements in a `vvm::vector` is one. From Table 7.1, all `char vvm::vectors` types in AltiVec mode, and all `vvm::vectors` in scalar mode have one element. Table 7.2 shows that when the number of elements in a `vvm::vector` was one (indicated by a 1 in columns A and B in Table 7.2), the VVM implementation was within 1% (in bold in Table 7.2) slower than a hand-coded program, when operating on expressions with one to five additions. When the number of elements in a `vvm::vector` increased to two and four (indicated by 2 and 4 in column A in Table 7.2 respectively), the VVM implementation was within 11% to 20% (in bold in Table 7.2) and within 14% to 24% (in bold in Table 7.2) respectively, when operating on expressions involving one to five additions.

Sections 7.3.5 and 7.4 suggest that a VVM implementation based on expression templates would have poor performance for `vvm::vectors` with only one element, near optimal performance for `vvm::vectors` with four or sixteen elements for expressions with a single addition, and increasing overheads as the number of additions in the expression increases. These behaviours are evident in Table 7.3. For `vvm::vectors` with a single element, the VVM implementation was more than 11 times slower than a hand-coded program. For `vvm::vectors` with two elements, the overhead increased from within 1% to within 44% when the number of additions in the expression increased from one to five. For `vvm::vectors` with four elements, the overhead was within 0% to 22%.

The results obtained suggest that the main overheads are related to the technique used to perform the operation. The cost of switching between the emulation layer and the active vector-processor implementation was generally optimised away by the compiler. Expression templates appear to have poor performance for short vectors. For `vvm::vectors` with a single element, the overheads of performing the operation was slightly faster than a hand-coded program. Expression templates were faster than function overloading for `vvm::vectors` with two elements when the number of additions was less than three. For `vvm::vector` with four elements, expression templates were faster for expressions with less than five additions. Expression templates appear to be affected by the number of additions in the expression more than function overloading.

7.8 Porting to other architectures

While this chapter used only AltiVec, on desktop computers today, there are actually two major vector-processing technologies: AltiVec and MMX derivatives. VVM can also be implemented using MMX derivatives since MMX derivatives have the same constraints as VVM: short vectors, fixed-size vectors and fast access to aligned memory only. This section focuses on how VVM could be implemented using MMX derivatives. The characteristics of the different MMX derivatives are discussed first, followed by brief discus-

sions on how using MMX derivatives would change the implementation details discussed in this chapter.

MMX derivatives include SSE, SSE2, SSE3 and 3DNow!. All MMX derivatives use the floating-point unit as the vector-processing unit, unlike AltiVec which has a separate vector-processing unit. In addition MMX derivatives are little-endian processors, unlike AltiVec which is found on big-endian PowerPC processors. This difference in endianness is not expected to be a problem when implementing VVM because there is no VVM operation that depends on the exact order of bytes within a `vvm::vector`. However, similar to how scalar code can depend on endianness, the user can also write abstract VPU code that depends on endianness. Since generic programs should not depend on endianness, generic vector programs need to avoid such constructs.

The different MMX derivatives are summarised below:

MMX: MMX provides support for 64-bit vectors for bytes, 16-bit integers and 32-bit integers (Int 2005).

Like AltiVec, MMX also requires data to be aligned. Unlike AltiVec, which requires data to be aligned at 128-bit boundaries, MMX requires data to be aligned at 64-bit boundaries.

SSE: SSE adds a single 128-bit vector support for 32-bit `floats` to MMX. Ideally for a SSE vector processor implementation, the preferred scalar count would be 16. However, this means that only `float vvm::vectors` will consist of only one VPU vector. `char vvm::vectors` would consist of two VPU vectors. Because this thesis is focused on supporting a machine-vision library, and `vvm::vectors` with more than one VPU vector incurs additional overheads, a SSE implementation for this thesis would ignore SSE's support for `floats`.

SSE2: SSE2 adds 128-bit vector support for 8-bit, 16-bit, 32-bit and 64-bit integers, and 64-bit floats to SSE. This means that SSE2 actually provides more vector support for more types than AltiVec. Namely, AltiVec lacks vector support for 64-bit floats. The preferred scalar count for SSE2 would be 16. The 64-bit instructions inherited from MMX would not be used at all.

SSE3: SSE3 does not add any new data type support to SSE2. Instead, SSE3 provides 13 additional instructions.

How implementations of VVM for MMX derivatives is likely to differ from the implementation for AltiVec discussed previously is summarised below:

Fundamental scalar types: The fundamental scalar types is specified by VVM, and are independent of the VPU. Thus using a MMX derivative is not expected to have any effect on fundamental scalar types.

Traits: The template metaprograms described for mapping between scalars and VPU vectors are also applicable to MMX derivatives. The only difference from an implementation for AltiVec would be a different typelist of VPU vector types. Since MMX derivatives provide different support for different types, the scalar-to-VPU vector mappings would be different. In fact, the mappings are expected to be different between MMX derivatives. Table 7.4 summarises the expected support for fundamental scalar types. The table is not definitive, because the size of fundamental scalar types is not fixed. SSE2 and SSE3 actually provide more `vvm::vector` support than AltiVec. Namely, they provide support for `doubles`.

The preferred scalar count for MMX would be 8, because MMX vectors are 64 bits long. For SSE, because `float` vectors are 128 bits long, a SSE vector-processor implementation would ideally have a preferred scalar count of 16. However, this means that only `float vvm::vectors` will consist of only one VPU vector; `char vvm::vectors` would consist of two VPU vectors, since all other SSE vectors are 64 bits long. Because this thesis is focused on supporting a machine-vision library, and `vvm::vectors` with more than one VPU vector incurs additional overheads, a SSE implementation for this thesis would forego support for `floats`. SSE2 and SSE3 provide 128-bit vectors for all the types that it supports. Thus the preferred scalar count for SSE2 and SSE3 is 16.

Performing an operation: The best method for performing an operation depends largely on the compiler used. Since different processors are typically supported by different compilers, it would be best to re-time the different methods discussed. If the VVM implementation is to support multiple vector-processing technologies, it would be a good idea to choose a method that all compilers being used will generate efficient code for.

Switching: The suggestions for performing an operation also apply to switching.

Type conversions: All MMX derivatives have type conversion functions. VVM implementations for MMX derivatives would be similar to the AltiVec implementation except different instructions would be used.

Functors: Functors are implemented using VVM operations. Thus the VPU has no effect on the implementation of functors.

Instructions: Instruction-wise, MMX derivatives also have the arithmetic instructions, though like AltiVec, they also lack full multiplication and division. Like AltiVec, MMX only provides instructions to access aligned memory locations; it uses aligned access and shift operations to access unaligned addresses (Mittal et al. 1997). Starting with SSE however, there are actually instructions for accessing unaligned mem-

Table 7.4 Expected `vvm::vector` support for MMX derivatives with Apple GCC 3.1 20021003 and Apple GCC 3.3 20030304.

	Size in bits	MMX	SSE	SSE2	SSE3	AltiVec
`char`	8	√	√	√	√	√
`signed char`	8	√	√	√	√	√
`unsigned char`	8	√	√	√	√	√
`short int`	16	√	√	√	√	√
`unsigned short int`	16	√	√	√	√	√
`int`	32	√	√	√	√	√
`unsigned int`	32	√	√	√	√	√
`long int`	32	√	√	√	√	√
`unsigned long int`	32	√	√	√	√	√
`float`	32		√	√	√	√
`double`	64			√	√	
`long double`	64			√	√	

ory. These additional load functions would be used by VVM's unaligned load and store functions.

7.9 Applications of VVM

Ideally, since VVM is an abstract VPU, any library or application that uses real VPUs can potentially use VVM as its VPU instead of the real VPU. However, due to current implementation shortcomings, practically, any application or library that operates on non-`char` types would actually experience performance drops when switching from a real VPU to the abstract VPU. For applications suitable for vector processing that do not operate primarily on `char` types, the use of VVM would be to build for the future. It is hoped that in the future, a VVM implementation that has comparable performance to hand-coded programs for all types would be possible. Since the use of VVM does not preclude the use of real VPU instructions, real VPU instructions could be used for portions that require higher performance.

VVM is designed to allow generic algorithms to use the VPU. While VVM's implementation is not zero-cost for all types, because it requires constant scalar count across types, as Appendix C shows, an algorithm cannot be made generic without this restriction. Thus any generic library that wishes to use the VPU should use VVM.

Applications areas that use the VPU are digital signal processing, linear algebra and 3-D geometry processing. Each application area is introduced briefly, followed by examples of some libraries that provide applicable routines. How VVM can be used in these libraries is then discussed.

Digital signal processing: Digital signal processing (DSP) refers to the processing of signals digitally. Signals are generally processed to reduce "noise". By removing or at least reducing unwanted signals, the quality of the signal increases. DSP is common in many devices such as mobile phones, video recorders and even CD players. Since DSPs typically involve applying the same operation to a large amount of data, VPUs are frequently used to increase the throughput of DSP operations.

vDSP (App 2005) is a vectorised DSP library for AltiVec. vDSP provides Fast Fourier Transforms (FFTs), convolutions, and squares. vDSP functions typically operate on floating-point types: `float` and `double`. Since vDSP is targeted at AltiVec and AltiVec does not have `double` support, vDSP functions for `doubles` are not accelerated. According to Section 7.7, a VVM implementation for `float` could add overheads of as little at 2%, for expressions with a single operation, if it was implemented using expression templates. The overheads rise to 22% when the number of operations in a single expression increases to five. For `doubles`, VVM will also use the scalar processor in AltiVec. If it is compiled without support for AltiVec, it could provide vector support with minimal overheads. However, since vector and scalar programs are different, using VVM without the scalar processor would still add overheads. Unfortunately, if VVM was compiled for AltiVec the overheads would be much larger for `doubles`.

Thus vDSP might be able to be implemented using VVM. While using VVM would allow the same program to be used for `floats` and `doubles`, vDSP's existing scalar functions are likely to be faster for `doubles` than a VVM vector program. Because vDSP is not generic, and does not perform operations on different types simultaneously, VVM's design is not actually a particularly good fit. An abstract VPU designed for vDSP could have `vvm::vectors` which all consists of a single VPU vectors. Such an abstract VPU would be less likely to introduce unexpected overheads.

Linear algebra: Linear algebra is a branch of mathematics that deals with vectors, vector spaces, linear transformations and systems of linear equations. On the computer, linear algebra is usually provided by Basic Linear Algebra Subprograms (BLAS). BLAS is used as the building blocks for other linear algebra libraries such as LINPACK and LAPACK.

VecLib also provides BLAS support for AltiVec. BLAS functions also typically operate with floating-point types. Thus the same comments made on using VVM for DSP is also applicable.

3-D geometry processing: With the advent of 3-D graphics, 3-D geometry processing has become more important. VPUs improve the performance of 3-D geometry pro-

cessing by allowing more vertices to be processed simultaneously. Ma & Yang (2002) were able to achieve close to four times the throughput using SSE, by processing four vertices at a time. 3-D geometry processing involves the use of `float` vectors. Thus many of the same comments made on using VVM for DSP is applicable. The effect of using VVM in Ma & Yang (2002)'s paper would be an additional overhead of at least 2%, for a VVM implementation based on expression templates. For the VVM implementation used in this thesis, which is based on function overloading, the overheads would be at least 19%.

VVM is designed to enable the creation of generic, vectorised libraries. As such, even though it could be used in other existing libraries that are not generic, it is not a particularly good fit. VVM would provide constant scalar count, which is not required, at the cost of performance. An abstract VPU for such libraries could have `vvm::vectors` that always consists of a single VPU vector. Since there are no vectorised, generic libraries available currently, there is no existing library for which VVM would be required.

Function overloading provides near zero-cost for `vvm::vectors` with only one VPU vector or scalar. So far, this thesis has suggested constant scalar counts that would allow `char vvm::vectors` to consist of only one VPU vector. However, this is only because this thesis is targetted at machine-vision, and machine-vision operations typically use `char` types. For DSP, where the most common type used is `float` and `double`, the constant scalar count could be set so that `float vvm::vectors` have only one VPU vector. For example, the constant scalar count would be four for AltiVec. This however precludes the use of the VPU for `vvm::vectors` where the number of scalars in a VPU vector is greater than four. For example, it would preclude `char` and `short vvm::vectors` in AltiVec. These types would processed using the scalar processor. Thus it is possible to select other types apart from `char` to have the best performance, using the techniques discussed so far.

For situations where performance is required for all types and constant scalar count is required, alternative VVM implementation methods that are discussed in Chapter 12 might be a better choice.

7.10 Conclusion

Details of how VVM can be implemented were discussed in this chapter. The implementation of traits, performing an operation efficiently, switching between the emulation layer and the active vector-processor implementation efficiently, type conversions and the results from a performance tool were covered. In addition, porting VVM to other desktop vector technologies and how VVM might be used for applications other than machine vision were discussed. Through these discussions, this chapter provides new insights into

the implementability and usefulness of an abstract VPU. The VVM specification (and thus abstract VPUs) can be implemented using only Standard C++; and conditional zero-cost is possible with Apple GCC 3.1 20031003. Using only Standard C++ makes the implementation more portable across different compilers and makes it easier for users to use, since the programmer is not required to perform any additional steps to compile his program.

VVM can be implemented using only Standard C++. Through the use of template metafunctions, the compiler can deduce the correct relationships between scalar and VPU vectors, even though the size of types in C++ is not fixed. Having an expression that uses both the vector and scalar processor is possible. The fall-back nature of the abstract VPU to the emulation layer is implementable without requiring any special extensions or extended compilation processes. The implementation of VVM shows that the abstract VPU is implementable using Standard C++.

While VVM can be implemented using only Standard C++, there are some performance issues. The cost of the abstract VPU depends on how it is implemented, and the compiler used. For an abstract VPU's operation to be zero-cost, there should be no additional overheads when executing the appropriate instruction, and when switching between the emulation layer and the active vector-processor implementation. With Apple's GCC 3.1 20021003, a VVM implementation based on function overloading and template switching was within 1% slower than a hand-coded program when the number of elements in a `vvm::vector` was one. For `vvm::vectors` with two and four elements, this implementation was within 11% to 20% and within 14% to 24% slower than a hand-coded program respectively. A VVM implementation that used expression templates based on Veldhuizen (1995*b*)'s paper and function switching was faster than function overloading when the number of elements in a `vvm::vector` was two for expressions with less than three additions and when the number of elements in a `vvm::vector` was four for expressions with less than four additions. When there is only a single element in a `vvm::vector`, expression templates were at worst about 11 times slower. For `vvm::vectors` with two VPU vectors, the cost increased from within 1% to within 44% when the number of additions in the expression increased from one to five. For `vvm::vectors` with four VPU vectors, the cost was within 0% to 22%. Switching between the emulation layer and the active vector-processor implementation did not add any additional overheads if implemented correctly.

As discussed previously, there are many other applications for VPUs that do not operate primarily on `char` types, but operate primarily on a single type. For such situations, the VVM implementation can still use function overloading to be near zero-cost for the most frequently used type. The constant scalar count would be adjusted so that vvm::vectors of the most frequently used type would consists of only one VPU vector. While this would boost performance for the most frequently used type, note that types whose corresponding

VPU vectors consist of more scalars than the constant scalar count would be relegated to the scalar processor.

Since most image-processing operations operate on `char` types, a VVM implementation based on function overloading would be within 1% slower than a hand-coded program for most image processing operations. It will be within 1% slower than a hand-coded program for all types in scalar mode and for `char` types in AltiVec mode.

Chapter 8

Categorisation of Operations based on Input-to-Output Correlation

Every field has a vast number of different operations. To help manage this complexity, it is usual to categorise these operations into smaller groups. A common method of grouping is by functionality. Categorisation by functionality provides hints on what the operation does and how it works. It is evident in National Instruments' IMAQ library (Nat 1999), which has categories like lookup transformations, operators, spatial filter, and frequency filters. Intel's Performance Primitives library (Int 2000-2001) also uses this categorisation and has groups like colour conversions, and arithmetic, logical and morphological operations. Thacker et al. (1994) categorised image-processing operations by the demands placed upon hardware to ease the implementation of image-processing operations in hardware. For example, in this categorisation scheme, thresholds load "an image block into a register of the processor", generate a "block of output image data at a time" and are classified as pixel operations. Convolutions load "a row of image data into one register of the processor", generate "a row of output image data at a time" and perform "multiple 2D image operations per block". Both categorisation by functionality and by hardware demands map poorly to generic programming.

One important characteristic of a generic library is that algorithms are separated from the data they that operate on and the operations that transform the data. Algorithms are the most important component of a generic library (Musser & Stepanov 1994). Current generic programming libraries have a list of algorithms that apply the functor to the data in different ways. For example, in the Standard Template Library (STL), `std::for_each` applies the functor to each element in a data set, discarding any results; `std::transform` applies the functor to each element in the data set and writes the result to another data set. VIGRA (Köthe 2000*c*) also follows this style with algorithms such as `vigra::transformImage` and `vigra::transformImageIf`.

Having more algorithms than necessary is not an issue for Standard C++ and VIGRA because generally the algorithms are easy to write. However, as Lai et al. (2002) showed,

a corresponding vectorised algorithm is much more complex and has to handle additional issues such as memory misalignment, misalignment between the input and output sets, edge handling and prefetching. It is therefore desirable to reduce the number of algorithms to a manageable level. Is it really necessary to have a `vigra::transformImageIf` when a `vigra::transformImage` with an IF functor would be sufficient?

A division based instead on the input-to-output correlation of machine-vision algorithms is therefore proposed. By grouping algorithms using characteristics of their input-to-output correlation, it is possible to reduce the number of algorithms that are required. For example, with this categorisation, the `std::accumulate` function is not longer necessary. This method of categorisation encourages more work to be done by the functors. The algorithm is only responsible for loading, and, in some cases, writing data.

This chapter starts with a description of how the input-to-output correlation categorisation works, and discusses some characteristics that are useful for the categorisation. Advantages and disadvantages of using this input-to-output correlation categorisation and how it can be applied to generic programming are discussed next. This is followed by a description of the categories that are used in the generic, vectorised, machine-vision library, detailed in the next two chapters.

8.1 Categorisation based on input-to-output correlation

Operations are categorised by the input required to produce the output, the output produced from the input and how the output is produced from the input. In this thesis, the characteristics that are used to categorise image-processing operations are as follows:

Number of input elements per input set: This refers to the number of input elements per input set that is used to produce the output elements. This number is assumed to be the same for all input sets required to produce the output. When selecting the correct number of input elements per input set for an operation, it is useful to visualise how the operation would be implemented using generic programming. The term "element" is used instead of "pixel" because separating the type from the number of input and output elements reduces the number of distinct algorithms. For example, both threshold and rotation are operations that require one input element to produce one output element, even though the element types are different. For thresholds, the element type is pixels while for rotations, the element type is coordinates.

Number of output elements per output set: This refers to the number of output elements per output set that is produced. This number is assumed to be the same for all output sets produced. It is also useful to visualise how an operation would be

implemented using generic programming when deciding what the number of output elements per output set is.

Number of input sets: This refers to the number of input sets used. In image processing, input sets are typically images.

Number of output sets: This refers to the number of output sets produced. In image processing, output sets can be images, histograms or statistics.

Input element type: This refers to the type of the input element. All input elements are assumed to be of this type. Most image-processing operations have input types of pixels. For geometric transformations, the input elements would be coordinates.

Output element type: This refers to the type of the output element. All output elements are assumed to be of this type. Possible output element types in image processing are pixels, histograms, and statistics. For geometric transformations, the output elements would be coordinates.

To show how an operation would be categorised using these characteristics, three operations are discussed:

Threshold: Using these characteristics, a threshold operation produces one output set from one input set; both input and output sets are images. One output element is produced from one input element; both input and output elements are pixels.

Histogram: A histogram operation produces one output set (a histogram) from one input set (an image). Since it examines one input pixel at a time and uses this one pixel to later produce the histogram, a histogram would be classified as using one input element per input set to produce zero or more output elements per output set. In this case, the input elements are pixels, while the output elements are statistics.

Binary arithmetic operations: Binary arithmetic operations, such as addition and subtraction, produce one output set from two input sets; all three sets are images. From each input set, one element is used to produce one output element. Both input and output elements are pixels.

Table 8.1 illustrates how image-processing operations can be categorised using the six characteristics discussed. M and N refer to the total number of input and output elements respectively. Rectangular refers to a rectangle of input, eg. a 3×3 pixel window. Spatial filters like Sobel filters, for example, typically produce a single pixel from a square of pixels centred around a pixel; they have rectangular input elements. A rectangle was used instead of a square to make the group more general. Since a single pixel is also a rectangle, operations accepting one input element per input set also fall under rectangular.

Table 8.1 Some image processing algorithms categorised using input-to-output correlation

Number of Elements		Number of Sets		Type of Elements		
Input	Output	Input	Output	Input	Output	Examples
1	1	1	1	Pixels	Pixels	Lookup transformations. Colour conversions. Eg. threshold, equalise, reverse and invert.
1	1	2	1	Pixels	Pixels	Arithmetic and logical operations. Eg. addition and subtraction.
Rectangular (eg. 3×3 pixels windows)	1	1	1	Pixels	Pixels	Spatial filters. Eg. convolution filters, edge extraction and edge thickness. Also includes some morphological analysis. Eg. erosion and dilation.
1	0 or more	1	1	Pixels	Number	Quantitative analysis. Eg. perimeter and area.
1	1	1	1	Coordinates	Coordinates	Geometric operations.
M	N	1	1	Coordinates	Coordinates	Scale operations.

While this thesis uses these six characteristics, there are other characteristics that can be used to perform categorisation. For example, a characteristic that specifies whether all input elements were always processed would be necessary if an operation like "find first" was to be implemented. This characteristic was not considered, because the target application, image processing, always processes all input elements. Thus all the groups shown in Table 8.1 assume all input elements will be processed.

8.1.1 Applied to generic programming

Different characteristics are handled by different concepts in a generic programming library. In the Standard Template Library (STL), the number of input elements, number of output elements, number of input sets and number of output sets are handled by the algorithm. The input element type and the output element type are handled by the iterator. For example, in STL, rotations can be expressed as a `std::transform` call that takes one input set and one output set, operating on iterators that return coordinates. In VIGRA, the number of input elements, number of output elements, number of input sets and number of output sets are also handled by the algorithm. The input element type and output element type can however be handled either by the iterator, or by the accessor.

Some characteristics have more impact on implementations than others. Since an algorithm that accepts two input sets is almost identical to an algorithm that accepts only one input set, the number of sets has less impact on the algorithm than the number of elements. For example, a `std::transform` implementation that accepts two input sets is almost identical to a `std::transform` implementation that accepts only one input set. In addition, if output is not handled by algorithms, fewer algorithms would be required. The number of output sets and number of output elements can instead be handled by the functor or the accessor. Under this scheme, the algorithm is only responsible for orchestrating the loading of data, while the functor/accessor is responsible for orchestrating the output. Examples of functors that write output are `vigra::FindAverage`, `vigra::FindBoundingRectangle` and `vigra::FindMinMax`. Since element types are handled by the iterator, number of input sets have little impact on algorithm implementations, and number of output elements and output sets can be handled by the functor or accessor, the most important characteristic for categorising operations so that fewer algorithms are required is the number of input elements.

When applied to generic programming, each category defined would have one main algorithm, with variations for each combination of input and output sets. Once an operation satisfies the criteria for a category, the category's algorithm can be used, even though in some cases it might not be the most efficient. For example, find first can be implemented using the same algorithm as threshold's, since they have the same six characteristics. However, find first would be faster if it gave up searching after finding the

answer. This example shows that more characteristics can be added to further narrow the groups. However, this was not done, because the objective, a generic, vectorised, machine-vision library, did not require any other characteristics.

An operation under the input-to-output correlation categorisation proposed can belong to many groups simultaneously, because groups overlap. For example, a convolution algorithm can also be used to perform thresholding because one input element is also rectangular input. In fact, only one main algorithm would be needed, one whose input element to output element correlation would be M to N. However, such a general algorithm would be highly inefficient, especially for operations that have a one input to one output element relationship. Generally, if an operation cannot be placed into a group smaller than M to N, then it is prudent to implement an algorithm for that operation explicitly.

8.1.2 Inferences

Some characteristics have implications for how an operation can be implemented. Characteristics that affect how the operation would be implemented generically, and how the operation would be implemented using the vector processor in this thesis are discussed below:

One input element per input set: This implies that only a 1-D iteration is needed. Since only a 1-D iteration is needed, the algorithm can be written to support regions of any shape.

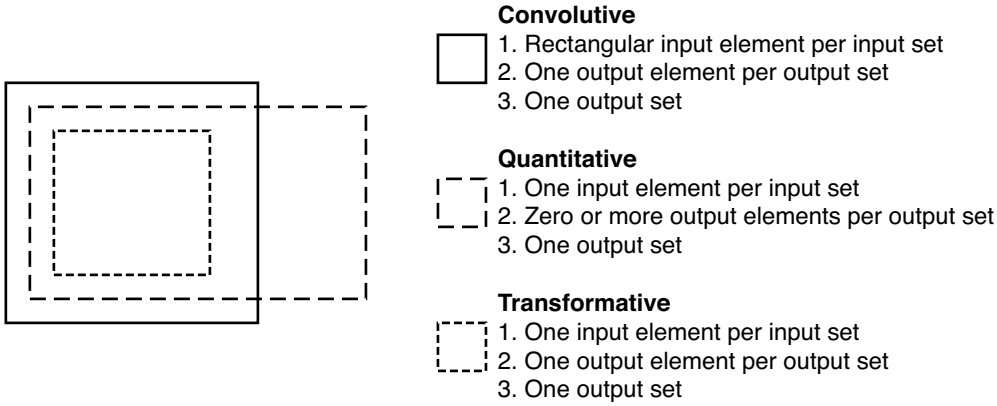
One output element: Having one output element implies that the algorithm can perform more work than necessary, as long as it ensures that the extra results are discarded. Since extra results can be discarded, edges can be handled using the vector processor (see Section 3.5.4 for more information about edges).

One input element per input set produces one output element per output set: This characteristic suggests that elements can be computed efficiently out of turn using a vector processor. The output order obviously still has to match the input order.

Rectangular input elements per input set: This characteristic indicates that a spatial iterator is required. For the vector processor, it is faster to compute this data from left to right, top to bottom, since data loaded from the last iteration can be used in the next.

Zero or more output elements: Zero or more output elements suggest that the algorithm cannot handle the output because the output is unknown. Functors cannot simply return a list of output elements because the answers might be unknown until all elements are processed, and functors generally do not know which element is the last element. While it is possible for the algorithm to inform the functor

Figure 8.1 Categorisation for the generic, vectorised, machine-vision library developed in this thesis



when the last element is reached, it is easier to let the functor handle the output. In addition, zero or more output elements suggest that performing more work than necessary can lead to wrong outputs, since the algorithm cannot discard unwanted outputs. The algorithm cannot discard unwanted outputs because it cannot handle the output.

8.2 Categories for the generic, vectorised, machine-vision library

The primary aim of using this categorisation scheme in this thesis was to reduce the number of algorithms required, while retaining efficiency. Because of this, the requirements of the algorithm form the basis for the categories. In the previous section, we observed that the most important characteristic for categorising operation so that fewer algorithms are required is the number of input elements. Using only this characteristic results in two categories. This indicates that only two main algorithms are required to handle all the image-processing operations considered in Table 8.1. However, since output characteristics were not utilised, the functor would always be responsible for the output. Because functors that handle output are more difficult to implement, consistency with other libraries makes algorithms more readily understandable, and most image-processing operations produce a single image as output, the output characteristics were also considered in deriving the categories for the generic, vectorised, machine-vision library. Since functors are the most commonly user-defined concepts, making functors more difficult to implement also makes the library more difficult to use

Both input and output characteristics were used to divide the image-processing operations considered in Table 8.1 into three categories: quantitative, transformative and convolutive. Figure 8.1 shows how the three categories are related to each other.

Quantitative operations: Quantitative operations have one input element per input set, zero or more output elements per output set and a single output set. Because they have zero or more output elements, the output is handled by the functor.

An example of a quantitative operation is the histogram.

Transformative operations: Transformative operations accept one input element per input set. In image processing, transformative algorithms require one or two input sets and produce either one output image set, or one numerical output set. Since transformative operations required either one or two input sets, two algorithms will be provided.

Since they produce a single output element, calculating more than is necessary is a not a problem. In addition, because they require only one input element, they are also easy to parallelise and can be computed out of order without problems.

Examples of transformative operations are thresholding, addition and subtraction.

Convolutive operations: Convolutive operations accept a rectangle of elements per input set, and produces a single output element for one output set. Convolutive algorithms are named after the name of operations that mostly fall under it — convolutions.

Because they have a single output element, calculating more than is necessary is not a problem. Because they have rectangular input, convolutive algorithms require spatial iterators. While convolutions can be computed out of order, there is little incentive to do so when using the VPU, because most of the input already loaded for the current output is required when computing the next output.

Examples of convolutive operations are linear and non-linear filters like Sobel and Gaussian filters.

8.3 Conclusion

In this chapter, a new categorisation scheme based on input-to-output correlations, and a description of the categories that will be used in the generic, vectorised, machine-vision library were discussed. The categorisation scheme was proposed by the author as a classification method that maps more directly to generic programming. By using this categorisation scheme, it is easier to decide how many algorithms are required, and the requirements of the iterator, the accessor and the functor. This categorisation scheme was used to derive three categories for use with the generic, vectorised, machine-vision library. The categorisation's main disadvantage is that a single operation belongs simultaneously to multiple categories.

A categorisation scheme based on input-to-output correlation is ideally suited to generic programming, identifying what algorithm an operation should use, and even providing implementation inferences. For example, if only one input element per input set is required, only a one-dimensional iteration is needed. It helps to reduce the number of algorithms required, and clearly defines the algorithm's main role as coordinating the loading, and, in many cases, the writing of data. Fewer categories would have been required if the algorithm was not responsible for coordinating the output.

Applying this categorisation scheme led to three categories for the generic, vectorised, machine-vision library: quantitative, transformative and convolutive. Since each category requires only one main algorithm, the number of algorithms required is small. A category might have more than one algorithm to handle different combinations of input and output sets. These algorithms are expected to be extremely similar. The machine-vision library only needs to provide four algorithms to support all the image-processing operations considered in this thesis. Having fewer algorithms is good for a generic, vectorised, machine-vision library, because vectorised algorithms are not particularly easy to write.

Chapter 9

A Generic, Vectorised, Machine-Vision Library

A generic, vectorised library is based on generic programming paradigms (Musser & Stepanov 1989, 1994) and uses the VPU. Genericity produces libraries that are more adaptable to the user's needs without sacrificing much performance. Examples of generic libraries include the Standard Template Library (STL) (Inf 1998), Matrix Template Library (Siek 1999) and Vision With Generic Algorithms (VIGRA) (Köthe 2000*b*, 2001). None of these generic libraries use the VPU.

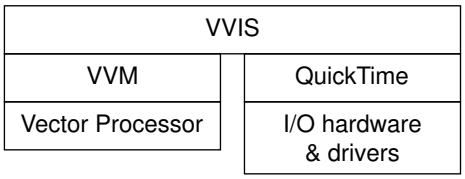
Being generic allows a library to better adapt to the user's environment, while using the VPU gives a library better performance. Generic image-processing libraries already exist. Image-processing libraries that use the VPU also exist. However, there is currently no image-processing library that is generic and uses the VPU.

This chapter discusses the creation of a generic, vectorised, machine-vision library called Vectorised Vision (VVIS). In particular, it covers issues applicable to the design of generic, vectorised libraries. An overview of VVIS is presented first, followed by why VIGRA and other generic libraries are unsuitable for vectorisation, and the division of duty selected for VVIS. Each concept and its responsibilities are detailed in ensuing sections. Import/export issues are covered last.

9.1 VVIS overview

Vectorised Vision (VVIS) aims to be a generic, vectorised, machine-vision library. Since machine vision involves image acquisition, processing, analysis and output, VVIS provides functions to facilitate all these processes. Images can be either loaded from disk or streamed from a sequence grabber. Image processing and analysis is where VVIS uses the VPU. For output, VVIS is able to write to files and to convert images to QuickTime GWorlds for display.

Figure 9.1 VVIS's architecture



VVIS uses VVM to access the VPU and QuickTime to import and export images. QuickTime is used to access sequence grabbers, to load and write images from and to files. VVIS's architecture is summarised in Figure 9.1.

Using an abstract VPU, like VVM, allows VVIS to be cross-platform, while preserving the ability to express its algorithms in a vectorised manner. Furthermore, it allows VVIS to support more vector-processor technologies more easily. A port requires only an additional vector-processor implementation for the abstract VPU. VVM provides additional benefits like an easy-to-use interface, template support and operators. The easy-to-use interface not only makes coding VVIS easier, but also user-defined functors and code. Template support is necessary for VVIS since all concepts are implemented as templates. Operators allow for a cleaner syntax, and the same templated code to be used for both scalar and `vvm::vector` implementations that do not use logical and comparison operations. The same templated code cannot always be used for both scalar and `vvm::vector` implementation because logical and comparison operators in VVM return a `vvm::vector` of booleans instead of a single `bool`.

Using VVM instead of programming for a VPU directly however introduces the possibility of speed degradation. However, since zero-cost is an important criterion for an abstract VPU, hopefully the overheads would be minimal. Section 7.3 shows that it is possible to attain zero-cost under certain circumstances when using an appropriate compiler and compiler settings. The VVM implementation used by VVIS in this thesis is within 1% slower than a hand-coded program for `char` types when AltiVec is enabled, and when there is no VPU. This implementation was chosen because in machine-vision, most images are either 8-bit or use 8-bit channels.

QuickTime is used to read images from sequence grabbers, and read and write from and to files. Using QuickTime allows VVIS to easily acquire images from sequence grabbers, like cameras, and import and export image files. VVIS leverages QuickTime's ability to read and write a wide range of image file formats. QuickTime was chosen because the target platform is a Macintosh. Since QuickTime is also available on Windows, it should be possible for the input/output interface to operate on both platforms.

9.2 Division of duty

Every library has different concepts. For generic libraries, concepts, including their interactions and responsibilities, are particularly important, because users are allowed to replace components, and because implementation-wise, they is little type safety. Being able to replace parts of the library with components that are more suitable is one of the main advantages of using a generic library. Since the rest of the system assumes that users' components will handle all the duties of the original component, it is important to establish exactly the responsibilities of each concept and its interfaces.

This section starts with a brief description of the division of duty used by other generic libraries, in particular VIGRA. After discussing existing division of duties, reasons why they are unsuitable for a generic, vectorised, machine-vision library are discussed. Viable alternatives are discussed and evaluated, to allow an informed decision to be made.

9.2.1 Existing division of duties

VIGRA is a generic image-processing library, created as a result of a Ph.D. thesis (Köthe 2000*b*, 2001, 1998, 2000*a*,*c*, 1999). VIGRA was chosen as a starting point because VIGRA, like VVIS, aims to be a generic library and provides many image-processing routines. Differences between VIGRA and VVIS include the use of the VPU, VVIS's support for sequence grabbers, and routines provided for image processing and machine vision.

VIGRA uses the following division of duty:

Images: Images in VIGRA store and provide read/write access to the pixels in an image. VIGRA's own images keep their pixels in a contiguous chunky format (see Section 9.2.4 for more information on contiguous chunky formats).

Generally users replace images completely. There is no need for the user's images to have the same interface as VIGRA's images because algorithms do not access images directly.

Accessors: Accessors add an extra level of indirection between the data and the iterator. They read and write data from and to iterators. This extra level of indirection makes it easier to preprocess the data. For example, an accessor can be used to easily restrict an algorithm to only the red channel of a RGB image and to process multi-channel planar images in a generic image-processing library (Köthe 1998). Without the accessor, additional iterators and proxy objects will be required (Kühl & Weihe 1997, Köthe 1998).

User-defined accessors are useful for accessing portions of the data.

Iterators: Iterators decouple the algorithm from the image. They represent a location and are able to traverse the data that they refer to. In VIGRA, iterators are two-

dimensional. However, only algorithms that require spatial knowledge require two-dimensional iterators (Ablavsky et al. 2003). From the categorisation presented in Chapter 8, only convolutive algorithms will require two-dimensional iterators.

Since iterators are coupled to images, user-defined images will almost certainly require a user-defined iterator.

Functors: Functors are responsible for actually utilising the data provided. Some functors return values, while other keep their results within themselves. Functors are the most common concept that are added by the user, especially if the library does not support creating functors from expressions.

Algorithms: Algorithms decide how to retrieve data from iterators and accessors, and obtain answers from functors, and write the answers back using iterators and accessors. In VIGRA, algorithms are passed the upper left and bottom right corners to determine a rectangular region of interest.

Users are not expected to produce many algorithms, though they are certainly welcome to.

STL's division of duty is similar to the VIGRA's except it does not use accessors. The duties of the accessor are handled by the iterator. Ablavsky et al. (2003) also used the same division of duty as STL; the paper discusses the creation of iterators that allow image processing to be handled using STL algorithms. Matrix Template Library (MTL) has a division of duty similar to VIGRA's, except there is no accessor, and different types of matrices in the place of images.

9.2.2 Problems with existing division of duties

VIGRA has two problems: its use of upper left and bottom right corners to specify regions of interest makes specifying non-rectangular regions difficult, and vectorisation is difficult because algorithms do not know how data are kept in memory. The first problem is VIGRA-specific, while the second applies to other generic libraries, such as STL and MTL.

VIGRA specifies regions of interest by passing two iterators, which mark the upper left and bottom right corners of a rectangular region of interest, to the algorithm. This rectangular constraint is unnecessary, since operations that do not require spatial iterators (quantitative and transformative operations from Section 8.2) can use algorithms that operate on one-dimensional iterators and can process regions of any shape. Not being able to specify non-rectangular regions of interests is not serious though because it is possible to use a mask to constrain results to a non-rectangular region of interest.

Existing generic libraries, like STL, MTL and VIGRA, are difficult to vectorise because iterators hide from algorithms how the pixels are arranged in memory. While such decoupling increases flexibility, without details of how pixels are arranged, algorithms do not know whether the pixels are arranged in a manner conducive for vector programming, and thus whether the pixels should be processed with or without the VPU. A vector program that operates on scattered data can be slower than a scalar program. Section 3.7 showed how different versions of the same algorithm have different execution times depending on the alignment of the data.

For efficient vectorisation, libraries need to consider the following:

1. How will algorithm get vectors efficiently? How would algorithms handle situations where it is impossible to load vectors efficiently?

   Aligned, contiguous data are the easiest for the vector processor to load efficiently. Contiguous data also enables easier and more efficient prefetching which leads to faster loads and stores.

   When obtaining vectors efficiently is impossible, the easiest and safest course of action is to use the scalar processor to process the data.

2. How is unaligned data handled? In addition how will two images that have different alignments be handled?

   VPUs can typically only load data efficiently from aligned locations. Algorithms not only have to handle unaligned data, they also have to handle different source alignments when an operation involves two or more different sources.

3. How are edges handled?

   Edge handling was discussed in Section 3.5.4. Edges can be handled using the scalar or the vector processor. When processing edges using the VPU, programs might trigger an exception by accessing memory that does not belong to the current process if the edge is loaded directly. In addition, using the VPU to process edges only produces correct output if the current operation does not mind more work being done, namely transformative and convolutive operations (from Section 8.2). Quantitative operations should not have their edges processed using the VPU.

   The scalar processor can always be used to process edges. In addition, as mentioned in Section 3.5.4, since AltiVec can execute scalar instructions at the same time as vector instructions, using the scalar processor to handle edges might actually be faster than using the VPU (App 2002*e*).

4. Who handles prefetching?

   Prefetching was discussed in Section 3.4.3. Prefetching refers to moving data to the caches before it is actually used. Section 3.7 showed that prefetching increased

the speedup factor of a vector program over a scalar program significantly on a PowerPC G4.

As discussed previously in Section 3.4.3, not all VPUs handle prefetching in the same manner. While the PowerPC G5s also support AltiVec, they require less manual prefetching control because they have automatic prefetching and a larger bus. In fact, prefetching instructions might be detrimental to speed on the G5 since it might cause large bubbles in program execution time (App 2003*c*). Such differences in prefetching behaviour should be handled by VVM and not VVIS, since they are VPU specific.

To allow non-rectangular regions of interest to be specified easily, VVIS introduces the region and shape concepts. The region represents a region of interest in an image. Regions, like VSIPL's views, are responsible for representing regions of interest and to transform the data into a form more suitable for vectorisation where necessary. Shapes provide algorithms information about the shape of the region of interest. They allow algorithms to specify the shape of regions of interests that are accepted where necessary. For operations that require spatial iterators, namely convolutive operations, the algorithm must be coded specifically to support non-rectangular regions of interests.

Storages were introduced to address the vectorisation problem. Storages specify constraints on how their data are arranged, allowing the algorithm to make informed decisions regarding the use of the VPU. Regions and images either provide storages or are storages themselves. Storages are passed to the algorithm instead of iterators. Since some algorithms require specific shapes, it is important to associate a shape with a storage. Since iterators can move and thereby break the constraints guaranteed by a storage, passing iterators to the algorithm makes it difficult to decide whether to use the VPU at compile-time. Passing iterators to the algorithm was not considered in this thesis.

9.2.3 Viable alternative divisions of duties

Two divisions of duties were considered: Algorithm Only and Vector Head. The differences between these divisions of duties are summarised in Table 9.1. In both divisions of duties, algorithms determine when to use the VPU, because the use of VPUs affects the implementation of algorithms. Furthermore, both divisions of duties have edges, because quantitative algorithms (see Section 8.2) cannot process more data than necessary. Having no edges is not possible in VVIS, because VVIS uses VVM and VVM has constant scalar count. Having no edges would be possible if VVM had variable scalar count, or an additional scalar count of 1. For reasons why VVM has a constant scalar count, see Section 6.1. The main difference between Algorithm Only and Vector Head is that in Vector Head, algorithms do not need to handle misalignment between images, because storages

Table 9.1 Summary of responsibilities of viable divisions of duties

	Algorithm Only	Vector Head
Vector/scalar mode	Algorithm	Algorithm
Unaligned loads	Algorithm	Storage
Edges seen by algorithm	Left, Right	Right
Misalignment between images	Algorithm	N/A
Admit/release	N/A	Maybe

always return aligned data. This allows algorithms to assume that storages are not only aligned to memory but also to each other.

Both divisions of duties use the same concepts: storages, images, regions, accessors, iterators, functors and algorithms.

Storages: Storages specify how pixels are arranged in memory. Since storages are passed to algorithms instead of iterators, algorithms can decide when to use the VPU based on the storage type at compile-time, or information provided by the storage at run-time.

Images: These represent images. Images are storages, or provide storages. They provide read/write access to pixels.

Regions: Regions represent regions of interest in images. Regions are also storages, or provide storages.

Accessors: Accessors are responsible for loading and writing data from and to iterators.

Iterators: Iterators represent positions in a storage and are responsible for traversing storages. Different storage types can have different iterator types.

Functors: Functors perform the desired operation on the pixel(s).

Algorithms: Algorithms are responsible for using iterators to traverse through storages, using accessors to read and write data from and to iterators, and using functors to obtain the results. Depending on the storage type passed to an algorithm, or information provided by the storage at run-time, algorithms decide when to use the VPU.

Algorithm Only

In Algorithm Only, the algorithm handles all the VPU issues raised in Section 9.2.2; it is responsible for deciding when to use the VPU, loading, storing and processing both the left and right edges. A single algorithm is expected to have different implementations

to support different storage formats. A single algorithm, for example, could have an implementation that uses the VPU and one that does not.

Storages indicate whether data are arranged in a VPU friendly format or not. In addition, they provide iterators that indicate where data start and end. When the data is not arranged in a VPU friendly format, the user can pass to the algorithm either a storage that indicates this deficiency, or a region that first rearranges a copy of the data. While both the start and end iterators can be unaligned, both the left and right edges should be loadable using the VPU. To load the left edge using the VPU, the storage must ensure that the data are either aligned or all data from the first aligned position before the data belongs to the current process. Without this condition algorithms cannot load the unaligned left edges safely without skipping a `vvm::vector`. To ensure the right edge can be loaded using the VPU, an extra `vvm::vector` can be allocated at the end. Ensuring the right edge is loadable using the VPU allows the right edge to be processed using the VPU and facilitates unaligned loading.

Regions and iterators in Algorithm Only should be easy to implement because iterators can be pointers, and regions generally does not need to make copies and can refer to positions within the images directly. Algorithms are more difficult to implement in Algorithm Only than Vector Head because they have to handle misalignment on two edges, and between storages. Since Algorithm Only's algorithms are able to operate on any location in memory, less preprocessing is needed.

Vector Head

Like Algorithm Only, algorithms accept different storage types and decide which implementation to use depending on the storage types passed. In addition, storages specify how data are arranged. However, in Vector Head, storages ensure that the start iterator is aligned. While the end iterator can be unaligned, storages must ensure that the right edge can be loaded using the VPU.

Storages can fulfil this requirement by making a copy of the data, or by performing unaligned loads and stores. Making a copy of the data requires an admit/release phase, while performing unaligned loads and stores do not. Without the left edge, algorithms are easier to implement, and are faster because algorithms do not handle misalignment between different sources. Regions, storages and iterators are more difficult to implement because of the increased responsibilities of the storage.

9.2.4 Performance comparison

This section starts with an investigation into the performance of storage formats suitable for vector programs. This is followed by a description of four storage formats that were investigated. This section concludes with an investigation into the performance of the two

viable divisions of duties, Algorithm Only and Vector Head.

Storage formats

Four different storage formats were investigated: contiguous planar, contiguous chunky, Illife planar and Illife chunky. Contiguous storages refer to storage formats where the structure of a 2-D image is accessed via a polynomial into a "flat" storage. Illife storages access the same structure via a table of row pointers. While planar formats store each channel in a different array, chunky formats interleave all channels in a single array. Since VVIS is aimed at algorithms that work efficiently on a VPU, it interleaves a `vvm::vector` instead of a scalar at a time. For example, if there are 4 elements in a `vvm::vector`, then a chunky RGB image would be RRRRGGGGBBBBRRRRGGGGBBBB, instead of RGBRGBRGBRGB. When the context is unclear, the former is referred to as chunky vector, while the latter is chunky scalar. Figure 9.2 provides a visual representation of the different storage formats. Note that for single-channel images, chunky and planar formats are equivalent.

Figure 9.3 shows the time taken for different hand-coded programs to process four different storage formats on an Apple PowerMac with two 450 MHz PowerPC G4 processors with 32K L1 cache and 1Mb L2 cache each. For single-channel images, the functor accepted a single value and returned the result. For RGB images, the functors accepted six parameters, three for the input and three for the output. All versions were compiled using Apple GCC 3.1 20021003 with the `-Os` (optimise for size) switch, and timed twenty times. The times shown are the lowest times and include the cost of constructing functors. The versions are described in more detail below:

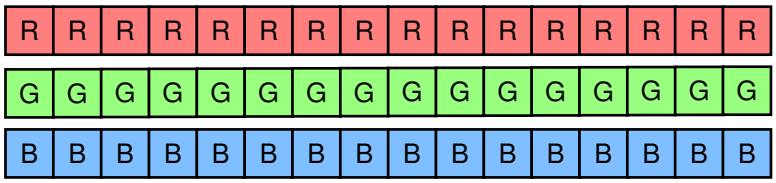
1-channel, Contiguous Planar: This version processed single-channel, `unsigned char` images, where the entire image is stored in a single contiguous block (a contiguous planar storage).

1-channel, Illife Planar: This version processed single-channel, `unsigned char` images, where all the rows were stored in separate contiguous blocks (an Illife planar storage). Illife planar images have h contiguous blocks, where h is the height of the image.

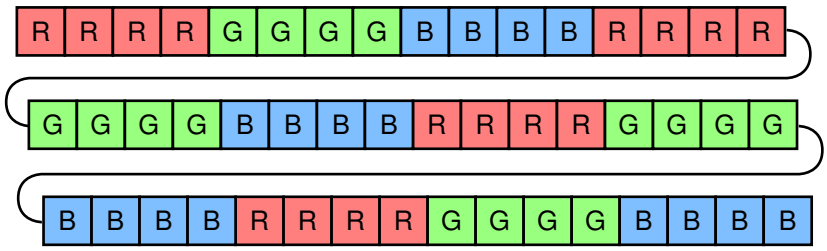
RGB, Contiguous Planar: This version processed RGB, contiguous planar images. Since each `unsigned char` channel is stored in separate contiguous blocks, these images have three contiguous blocks.

RGB, Contiguous Chunky: This version processed RGB, contiguous chunky images. In scalar mode, the pixels are interleaved one scalar at a time. In AltiVec mode, the pixels are interleaved sixteen at a time, because there are sixteen `unsigned chars`

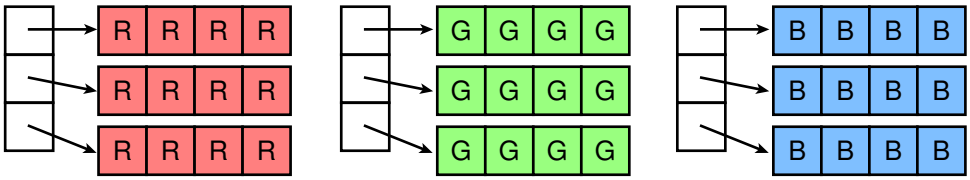
Figure 9.2 A 4x4 RGB image stored using different storage formats.



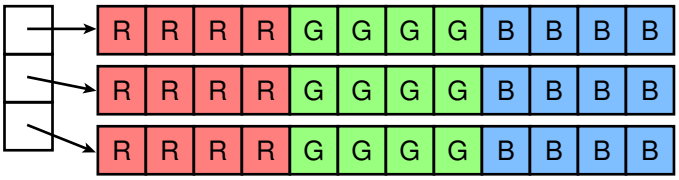
(a) Contiguous planar storage



(b) Contiguous chunky storage

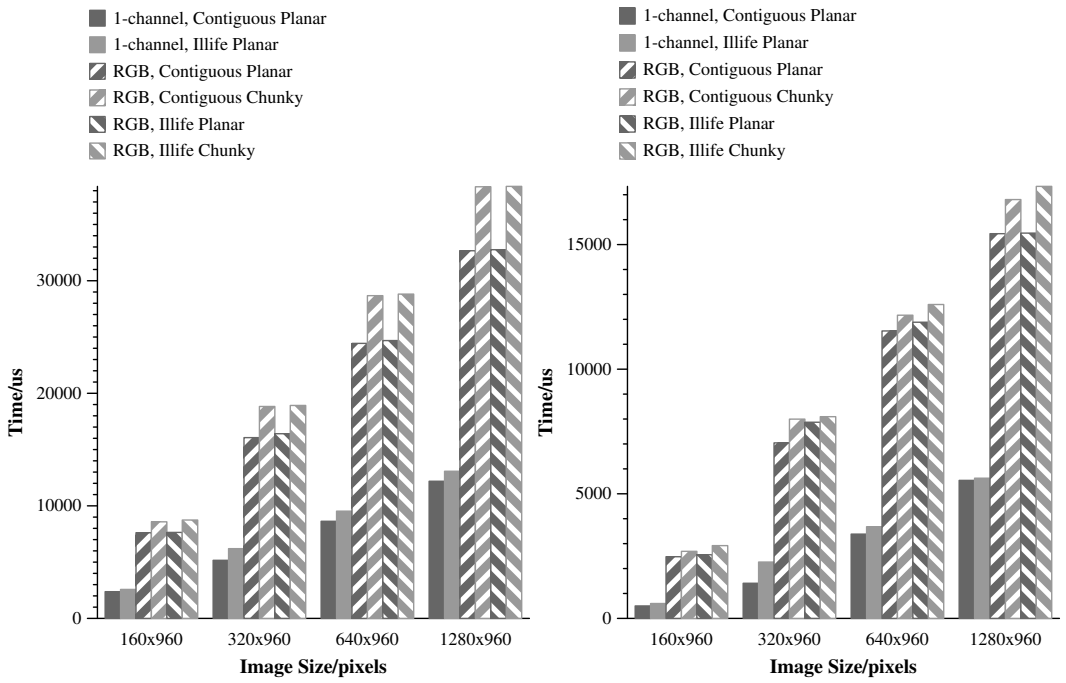


(c) Illife planar storage



(d) Illife chunky storage

Figure 9.3 Effect of storage formats on a simple transformative operation, where each pixel is added to itself, when the input and output are equivalent



(a) Scalar Mode

(b) AltiVec Mode

in a `__vector unsigned char`. The entire image is stored in a single contiguous block

RGB, Illife Planar: This version processed RGB, Illife planar images. Each row of each `unsigned char` channel is stored in a single contiguous block. RGB, Illife planar images have $3h$ contiguous blocks, where h is the height of the image.

RGB, Illife Chunky: This version processed RGB, Illife chunky images. Each row is stored in a single contiguous block, which has all three `unsigned char` channels interleaved within it. In scalar mode, the pixels are interleaved one scalar at a time. In AltiVec mode, the pixels are interleaved sixteen at a time, because there are sixteen `unsigned chars` in a `__vector unsigned char`. RGB, Illife chunky images have h contiguous blocks, where h is the height of the image.

Figure 9.3 shows the effect of storage formats on a transformative operation which adds each pixel to itself, when source and destination images are the same image. Figures 9.3(a) and 9.3(b) show the effect of storage formats in scalar and AltiVec mode respectively.

Figure 9.3 shows that in both scalar and AltiVec modes, it was slightly faster to process both single-channel and RGB images as contiguous storages than as Illife storages. For

RGB images, planar storages were noticeably faster than chunky storages in both scalar and AltiVec modes. The fastest RGB storage in both scalar and AltiVec modes was the contiguous planar storage. The Illife planar storage was not significantly slower though. Contiguous storages were faster than Illife storages probably because the pixels are closer to each other in memory. Since all channels were being processed, it was thought that chunky storages would be faster, since the chunky storage would place the pixels of the three channels that are being processed closer to each other. The results however show that planar storages were noticeably faster than chunky storages. This might be because the processor loaded the separate channels in the planar storages into the caches simultaneously. This would mean that more of the image was loaded into the caches for planar storages, which leads to less time spent waiting for the processor, than for chunky storages. Thus prefetching is likely to have more impact on chunky storages than on planar storages.

Different divisions of duties

In this section the performance of the two viable divisions of duties, Algorithm Only and Vector Head, are evaluated. Algorithms were developed for both divisions of duties and were compared to hand-coded scalar, hand-coded AltiVec and VIGRA programs in scalar and AltiVec mode. In scalar mode, the algorithms were evaluated against hand-coded scalar and VIGRA programs, while in AltiVec mode, they were compared to aligned only AltiVec, unaligned AltiVec and VIGRA programs. The scalar and AltiVec hand-coded programs were derived from the functions presented in Section 3.7. Algorithms that have similar performance to corresponding scalar or AltiVec programs are good implementations. While algorithms used VVM as its VPU, specialised versions for different channel counts were implemented; the algorithms evaluated are unsuitable for a generic library. Only contiguous planar storages were investigated because contiguous planar storages are the most common storage format used in vector programs. The performance of the algorithms on other storages are expected to follow trends shown in Section 9.2.4.

Figures 9.4 and 9.5 show the time taken for implementation to process contiguous planar storages (except VIGRA, which uses contiguous chunky scalar storages), when the source and destination storages are the same and different respectively. The subfigures (a) through (d) show the time taken for implementation to process single-channel images in scalar mode, single-channel images in AltiVec mode, RGB images in scalar mode, and RGB images in AltiVec mode respectively. The operation performed added each pixel to itself. All implementations, except VIGRA, did not use accessors. All implementations were compiled using Apple GCC 3.1 20021003, with the `-Os` (optimise for size) switch. From Chapter 7, this is the compilation environment under which the VVM implementation used with this thesis is zero-cost for `vvm::vectors` with a single element. A compilation environment that generates a non-optimal VVM implementation

will adversely affect the performance of the algorithms.

VIGRA: This version used VIGRA. VIGRA uses a contiguous chunky scalar storage format. From Figure 9.3, we expect VIGRA in scalar mode to be faster when processing RGB images and equivalent when processing single-channel images.

Scalar: This version refers to hand-coded scalar implementations. The functor was a function that accepted each channel as a separate argument.

When processing RGB images, this version accepts three pointers pointing to the start of each of the channels, a pointer pointing to the first after the end element of the first channel, three pointers pointing to the start of each channel of the output image and a functor. No tuples or arrays were used to represent RGB pixels.

AltiVec (aligned): This version is basically Aligned Load and Aligned Store from Section 3.7. This version only correctly handles storages whose data start and end on aligned memory addresses.

When processing RGB images, it accepts three pointers pointing to the start of each of the three channels, one pointer pointing to the first after the end element of the first channel, three pointers pointing to the start of each channel of the output image and a functor. The functor accepts six parameters, three for the input pixel's channels and three for the output pixel. No tuples or arrays were used to represent RGB pixels.

AltiVec (unaligned): This version is basically Unaligned Load and Aligned Store from Section 3.7. It can correctly handle storages whose data do not start or end on aligned memory addresses.

When processing RGB images, this version accepts seven pointers, one functor for VPU vector operations and one for scalar operations. Once again, no tuples or arrays were used to represent RGB pixels.

Algorithm Only (aligned): This refers to an algorithm that uses the Algorithm Only division of duty, but is only able to handle aligned data. It is basically a VVM version of the Aligned Load and Aligned Store from Section 3.7.

Algorithm Only (unaligned): This refers to an algorithm that uses the Algorithm Only division of duty and is able to handle unaligned data. It is basically a VVM version of the Unaligned Load and Aligned Store from Section 3.7.

Vector Head (without admit/release): This refers to an algorithm that uses the Vector Head division of duty. In this version, the cost of admitting and releasing is not included. Because no admit/release was included, this version is only representative when the the storage is already in the required format.

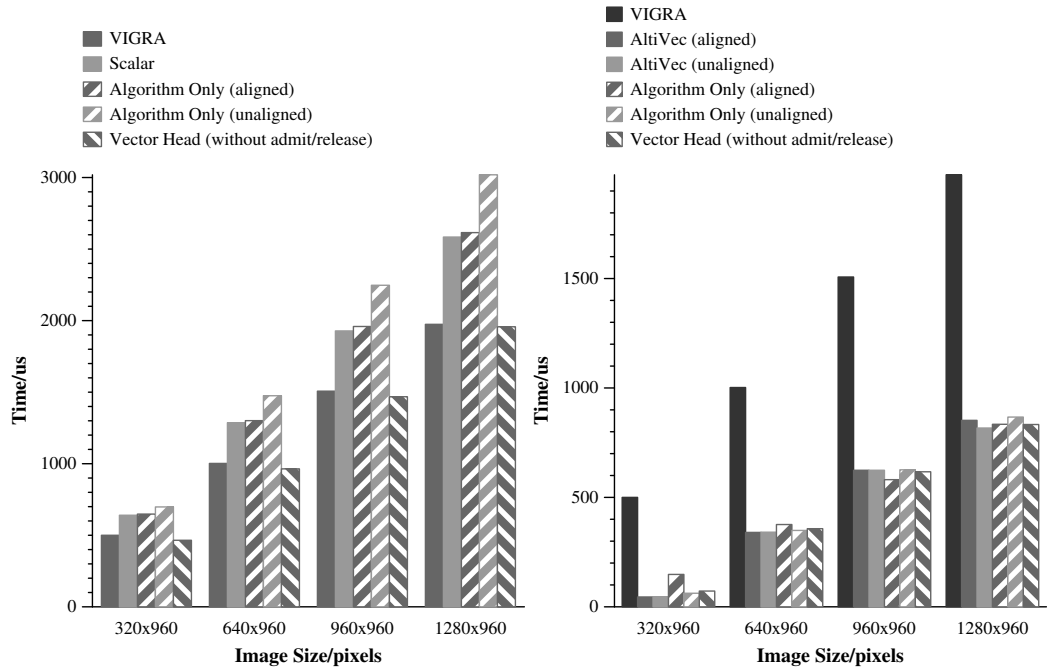
Vector Head (without admit/release) will only handle storages that start on an aligned location. This will cover images of any size as long as the entire image is used. While it handles more situations than Algorithm Only (aligned), it handles fewer than Algorithm Only (unaligned). In order for it to handle all cases, either it needs to convert a storage so that it is aligned at the beginning, or the iterator needs to perform unaligned loading for the algorithm. For example, if the user only wants to process half of the image, the user needs to use a region, admit the region of interest to the region, process the region, and then release the region.

Figure 9.4 shows how the different implementations fare when the source image is also the destination image. As mentioned previously, all implementations used contiguous planar storages except VIGRA, which used contiguous chunky scalar storages. In scalar mode when processing single-channel images, Vector Head (without admit/release)'s performance was comparable to VIGRA's and was faster than Scalar, Algorithm Only (aligned) and Algorithm Only (unaligned). This verifies that VVM introduced no significant overheads in scalar mode as expected from Section 7.7. However in scalar mode when processing RGB images, Vector Head (without admit/release) was slower than VIGRA, but comparable to Algorithm Only (aligned). In scalar mode, both Vector Head (without admit/release) and Algorithm Only (aligned) were slower than Scalar when processing 320×960 RGB images, but faster for larger images. As expected, Algorithm Only (unaligned) was the slowest when processing single-channel and RGB images in scalar mode.

In AltiVec mode when processing single-channel images, Algorithm Only (aligned) and Vector Head (without admit/release) had similar performance to AltiVec (aligned). This shows that as expected from Section 7.7, VVM in AltiVec mode had no significant overheads when processing `unsigned char vvm::vectors`. Algorithm Only (unaligned) was slower than AltiVec (unaligned) because VVM's unaligned load function performs a `switch` before a `vec_sld` (see Section B.2.1, `load(A, B, o)`). When processing RGB images, Algorithm Only (aligned) and Vector Head (without admit/release) had comparable performance. While these two implementations were slower than AltiVec (unaligned), these two implementations were surprisingly faster than AltiVec (aligned) for images 640×960 and larger. However, these two implementations were slower than AltiVec (aligned) for 320×960 images.

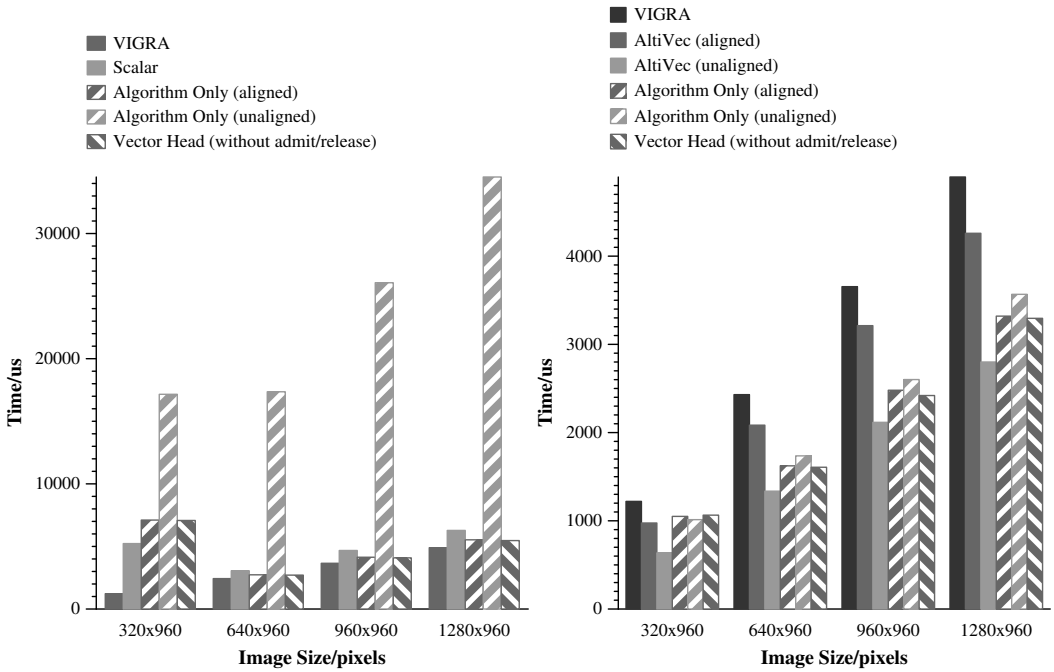
Figure 9.5 shows the performance of the different implementations when the source image is not the destination image. Both source and destination images were contiguous planar storages. Algorithm Only (unaligned) was significantly slower than Scalar in scalar mode when operating on both single-channel and RGB images. Algorithm Only (aligned) and Vector Head (without admit/release) was near optimal, in both scalar and AltiVec mode when operating on single channel images. Like Figure 9.4(a), Vector Head (without admit/release) was faster than Scalar, Algorithm Only (aligned), and Algorithm (un-

Figure 9.4 Performance of different divisions of duties when processing `unsigned char` contiguous planar storages. The source and destination images were the same image.



(a) Scalar Mode, 1-channel Images

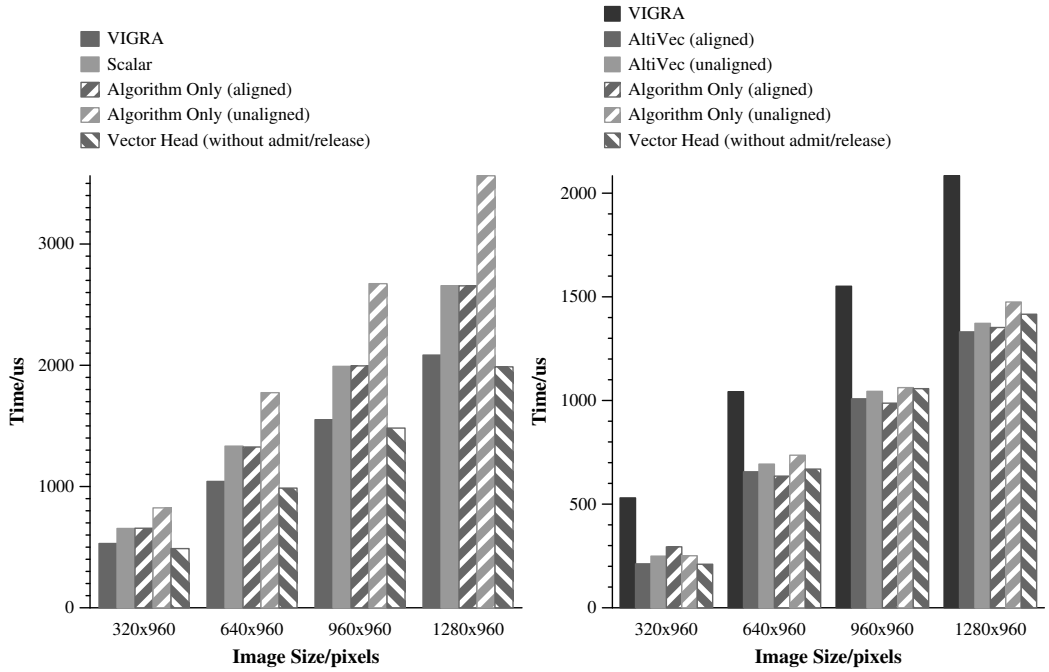
(b) AltiVec Mode, 1-channel Image



(c) Scalar Mode, RGB images

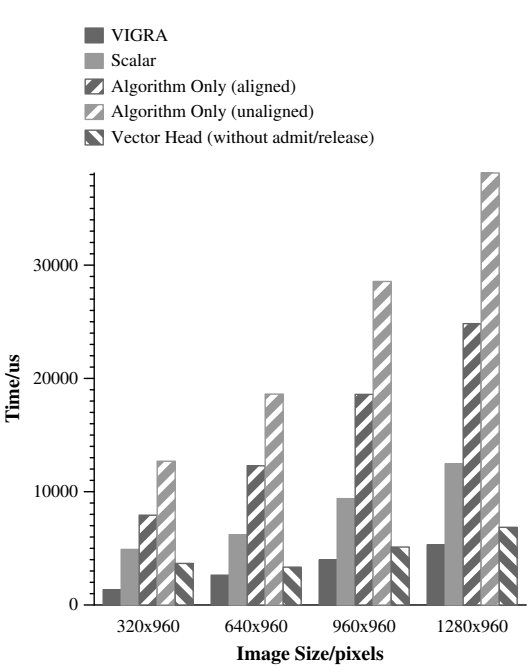
(d) AltiVec Mode, RGB images

Figure 9.5 Performance of different divisions of duties when processing `unsigned char` contiguous planar storages. The source and destination images were different images.

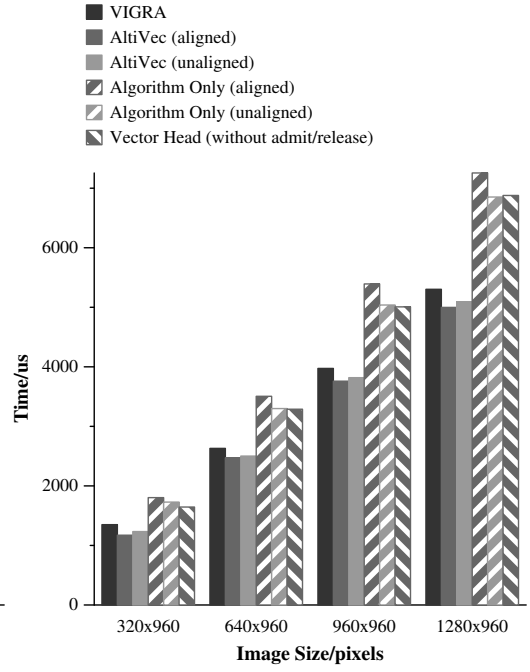


(a) Scalar Mode, 1-channel Images

(b) AltiVec Mode, 1-channel Image



(c) Scalar Mode, RGB images



(d) AltiVec Mode, RGB images

Table 9.2 Number of operations required for Vector Head to match Algorithm Only when operating on unaligned, contiguous planar storages. Source and destination storages were the same

	320×960	640×960	960×960	1280×960
Single-Channel Image, Scalar	20	18	17	17
Single-Channel Image, AltiVec	N/A	N/A	1588	560
RGB Image, Scalar	1	1	1	1
RGB Image, AltiVec	N/A	182	196	174

Table 9.3 Number of operations required for Vector Head to match Algorithm Only when operating on unaligned, contiguous planar storage. Source and destination storages were different

	320×960	640×960	960×960	1280×960
Single-Channel Image, Scalar	14	12	12	12
Single-Channel Image, AltiVec	122	143	2873	324
RGB Image, Scalar	1	1	1	1
RGB Image, AltiVec	195	2234	1199	N/A

aligned) in scalar mode when processing both single-channel and RGB images. However, in AltiVec mode when processing RGB images, Algorithm Only (aligned), Algorithm Only (unaligned), and Vector Head (without admit/release) were significantly slower.

Tables 9.2 and 9.3 show the number of addition operations that would be required to be performed, where the source and destination images are equal and not equal respectively, before the cost of admitting and releasing an entire image would be covered. In both cases, all storages used were contiguous planar storages. The admittance process copies all the data to a different location in memory, while the release process copies the data back. Under some cases, Algorithm Only (unaligned) was faster than Vector Head (without admit/release). This is because prefetching was off, and is consistent with results obtained earlier in Section 3.7. In such cases, it is impossible for Vector Head (without admit/release) to catch up; the table shows `N/A` in such situations.

The admittance and release processes that were evaluated copied each pixel one at a time. While this process is robust, its performance is not optimal. It should be possible to perform memory to memory copy operations between certain combinations of storages. Therefore the difference would be less for such cases. The program evaluated actually copies the entire image, which is something that is rarely needed, since regions of interests usually refer to a portion of the image. This version is likely to be the worst admittance method performance-wise.

Tables 9.2 and 9.3 show that the cost of copying is really quite large compared to the savings obtained through the use of Vector Head (without admit/release) instead of Algorithm Only (unaligned). Generally when AltiVec is enabled, the number of operations

Table 9.4 Template metafunctions for discovering the shape

Expression	Return Type	Notes
`is_rectangular<X>::value`	`bool`	Returns `true` if `X` is rectangular

required to match Algorithm Only is either very large, or is impossible. This suggests that Vector Head should use regions that can load data unaligned instead, especially if the work to be done is small. From Section 3.7, this version should run like Unaligned Load and Unaligned Store from Figure 3.6.

9.2.5 Division of duty chosen

Vector Head was chosen because it always performs well without admit/release. In addition, it simplifies the creation of algorithms greatly, since it removes the need to be concerned about misalignment, not only with memory, but also between storages.

Vector Head does not require an admit and release when the image is aligned, or if the region performs unaligned loads and stores automatically. Therefore it is just as fast as Algorithm Only for aligned images. The main advantage Vector Head has over Algorithm Only is that algorithms are much easier to code.

9.3 VVIS concepts

VVIS uses the Vector Head division of duty discussed in Section 9.2.3. This division of duty uses shapes, storages, iterators, images, regions, accessors, algorithms and functors. Each of these concepts is discussed in more detail in this section. The responsibilities, design issues and implementation issues of each concept are covered.

9.3.1 Shapes

Shapes refer to the shape of a storage. While VVIS only provides a single shape, the rectangular shape, shapes are included nonetheless because some algorithms require specific shapes. VVIS's convolutive algorithm requires the input and output storages to be rectangular. VVIS's transformative and quantitative algorithms accept storages of any shape, because transformative and quantitative operations do not require spatial iterators.

The template metafunctions in Table 9.4 indicate the shapes of storages. Since they are template metafunctions, algorithms can use them to decide how to handle storages at compile-time. Algorithms that can operate on any shape do not need to check the shape of a storage before using it.

Figure 9.6 Row-major and Column-major two-dimensional arrays

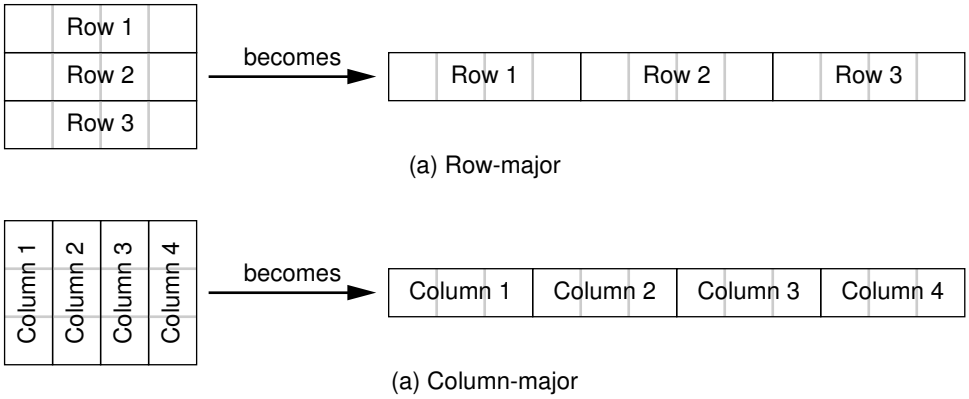


Table 9.5 Contiguous storage's required interface

Expression	Return Type	Notes
`X::component_tl`	`ct::typelist` of `T1`, `T2`, ...	
`X::iterator`	iterator type pointing to `T`	Convertible to `X::const_iterator`
`X::const_iterator`	iterator type pointing to `const T`	
`a.begin()`	`iterator`; `const_iterator` for constant `a`	
`a.end()`	`iterator`; `const_iterator` for constant a	

9.3.2 Storages

Storages refer to how pixels are arranged, relative to each other. Pixel arrangement is important to algorithms because it helps determine which method of processing the storage is best. Three storages are specified: contiguous, unknown and Illife. Contiguous and unknown storages are one-dimensional storages while Illife storages are *n*-dimensional storages. Only contiguous storages will be handled using the VPU.

All storages in VVIS are expected to be row-major; pixels adjacent horizontally are adjacent in memory, while pixels adjacent vertically are not. For a graphical representation of row-major and column-major refer to Figure 9.6. Row-major was chosen because this is the usual order used in the implementation language C++.

Contiguous storages

Contiguous storages are 1-D storages where components of pixels are assumed to be adjacent to each other in memory. Originally, components were required to be adjacent to each other from the beginning to the end, but the restriction was later eased to support chunky storages. Components of pixels now just have to be adjacent for a single `vvm::vector`.

The `begin` function returns an aligned iterator that points to the first element. The

Figure 9.7 Expected allocation requirements of contiguous storage

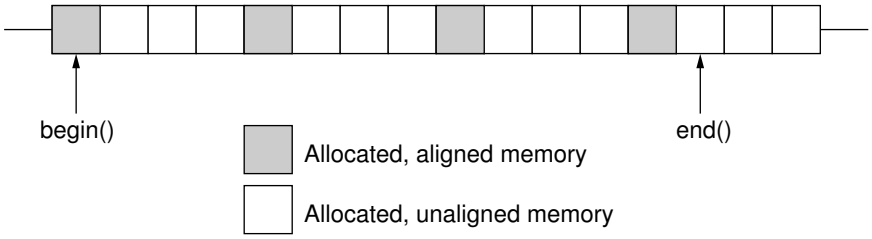
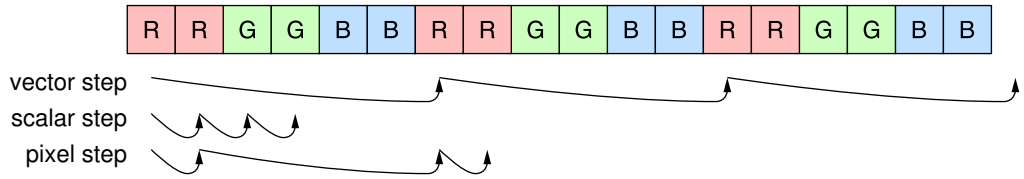


Figure 9.8 The difference between vector, scalar and pixel steps



`end` function returns an iterator, which may be unaligned, that points to the first past-the-end scalar. Note that even if the `end` function returns an unaligned iterator, it is expected that performing a `vvm::vector` load from the last aligned position is valid. This means that the contiguous storage is expected to pad the number of bytes allocated so that the allocation ends on an aligned position. Figure 9.7 illustrates this requirement. The figure assumes that there are four scalars in a `vvm::vector`. In this figure, three extra scalars were allocated at the end to allow the last aligned position to be loaded as a `vvm::vector`.

The iterator of a contiguous storage is expected to be able to traverse forwards and backwards by taking `vvm::vector`, scalar or pixel steps. Backward iteration is only used by convolutive algorithms. Scalar and `vvm::vector` steps refer to advancing the position so that the iterator now points to a position that is one scalar or one `vvm::vector` length away. Walking by scalar steps is only valid if the step does not leave the current `vvm::vector`. Pixel steps refer to steps where the unit is a scalar, but can cross `vvm::vector` boundaries. Pixel steps are used by convolutive algorithms to adjust the end iterator to cater for kernel sizes. Scalar and pixel steps both exist even though they both walk by the same unit because a pixel step may require more processing and an algorithm does not usually need to walk across `vvm::vector` boundaries. For planar images, the scalar and pixel step implementations are expected to be the same. For chunky storages however, the scalar implementation is much easier than pixel step's. Refer to Figure 9.8 for an illustration depicting the difference between `vvm::vector`, scalar and pixel steps in a chunky storage. While it is possible to use only pixel steps, advancing by `vvm::vectors` would be slower than necessary.

Originally, the iterator advanced by `vvm::vector` and scalar steps by adding different numbers. A `vvm::vector` step would be `i += vector_step`, and a scalar step would be `i += scalar_step`, where `i` is an iterator, `vector_step` is equal to VVM's constant

scalar count and `scalar_step` is one. Since this scheme allows iterators to be pointers, iterators are easier to implement, and produces code that is likely to be easier for the compiler to optimise. While this iterator style can iterate through planar storages easily, it cannot iterate through chunky storages, because unlike planar storages, `scalar_step` and `vector_step` would be constant only if their units are different; the units should be number of scalars and bytes respectively. For planar storages, `scalar_step` and `vector_step` both refer to the number of scalars to move by. If `scalar_step` and `vector_step` both referred to the number of scalars to move by, then they would only support chunky storages if all channels in the chunky storage were the same type. If we use different object types to represent `vvm::vector`, scalar and pixel steps, we can overcome this problem. Weihe (2002) provides some ideas on how such a syntax could be implemented correctly. It was not investigated further because it seemed to be a lot of work for small gains.

Using different iterator types for each type of walk was also considered and discarded because the resultant interface would be more confusing. It was considered because it was thought that it would be possible to use simple pointers as iterators in this scheme with both planar and chunky storages. Unfortunately, this is not possible, because there would have been only two pointer types (scalar and `vvm::vector`) and three different walks. In addition, the different iterators need to be convertible between one another, because we would want to be able to advance by `vvm::vector` steps when we were using the VPU and switch to scalar steps when we came to the edges. The only way to convert pointers legally is to use `reinterpret_cast`. This is not suitable if the iterators were not pointers. It is not possible to perform automatic conversions between pointer types.

The iterator originally kept a single location, and provided methods to move the position by different steps. Keeping a single location forces the algorithm to calculate where the position of the right edge, and might increase `!=` operator's requirements (`!=` operator might have to calculate the real position first). If instead, the iterator kept two orthogonal components, a scalar and a `vvm::vector` component, like a complex number, then `!=` operator can be applied to each component separately, and there would be no need to calculate the position of the right edge since once the `vvm::vector` steps were completed, only the right edge will remain. Keeping two orthogonal components however increases the cost of constructing the iterator, since the `vvm::vector` and scalar components have to be calculated. Keeping two orthogonal components was chosen because this method simplifies the algorithm's implementation, and reduces processing requirements during execution.

Table 9.5 shows the required interface of a contiguous storage. The expected interface of a contiguous storage's iterator is shown in Table 9.6. Note that there is no operation that reads data from or writes data to the current location. The required interface for that is determined by what accessors the iterator should be compatible with. See Section 9.3.5

Table 9.6 Contiguous storage's required iterator interface

Operation	Result	Notes
`difference_type`	Type	Returns type of `i - j`
`i = j`	`iterator&`	After operation, `i != j` is false
`i != j`	`bool`	After operation, `i == j` is false
`i - j`	`difference_type`	Returns number of pixels between `i` and `j`
`i.vector != j.vector`	`bool`	
`++i.vector; i.vector++`	`iterator&`	Move forward by 1 vector
`i.vector += j`	`iterator&`	Move forward by `j` vectors
`i.scalar != j.scalar`	`bool`	
`++i.scalar; i.scalar++`	`iterator&`	Move forward by 1 scalar
`i.scalar += j`	`iterator&`	Move forward by `j` scalars
`++i; i++`	`iterator&`	Move forward by 1 pixel
`i += j`	`iterator`	Returns an iterator that has moved forward by `j` pixels
`--i.vector; i.vector--`	`iterator&`	Move backward by 1 vector
`i.vector -= j`	`iterator&`	Move backward by `j` vectors
`--i.scalar; i.scalar--`	`iterator&`	Move backward by 1 scalar
`i.scalar -= j`	`iterator&`	Move backward by `j` scalars
`--i; i--`	`iterator&`	Move backward by 1 pixel
`i -= j`	`iterator&`	Returns an iterator that has moved backward by `j` pixels

for more details on accessors.

Unknown storages

Unknown storages refer to a 1-D stream of pixels where the relationship between the location of pixels in memory is unknown. Unlike contiguous storages, there are no constraints on how the pixels are stored relative to each other. While unknown storages provide the same functions as contiguous storages, they have different iterators. In addition, the `begin` and `end` functions can both return unaligned iterators. The `begin` function returns an iterator that points to the first element. The `end` function returns an iterator that points to the past-the-end element.

The iterator of a unknown storage is expected to be able to traverse forwards and backwards by taking pixel steps. There is no need for `vvm::vector` steps because unknown storages are expected to be processed by the scalar processor. The unknown storage's and its iterator's interfaces are shown in Tables 9.7 and 9.8 respectively.

Note that the unknown storage's iterator's interface is the same as the contiguous storage's iterator's interface except it does not have `.scalar` and `.vector` operations. This means that a contiguous storage also conforms to the unknown storage's interface.

Table 9.7 Unknown storage's required interface

Expression	Return Type	Notes
`X::component_tl`	`ct::typelist` of `T1`, `T2`, ...	
`X::iterator`	iterator type pointing to `T`	Convertible to `X::const_iterator`
`X::const_iterator`	iterator type pointing to `const T`	
`a.begin()`	`iterator`; `const_iterator` for constant `a`	
`a.end()`	`iterator`; `const_iterator` for constant a	

Table 9.8 Unknown storage's required iterator interface

Operation	Result	Semantics
`i = j`	`iterator&`	After operation, i != j is false
`i != j`	`bool`	Checks if `i` and `j` are pointing to the same location
`++i; i++`	`iterator&`	Move forward by 1 pixel
`i += j`	`iterator&`	Move forward by `j` pixels
`--i; i--`	`iterator&`	Move backward by 1 pixel
`i -= j`	`iterator&`	Move backward by `j` pixels

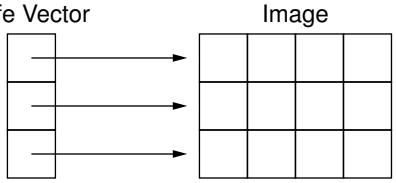
This is important because when processing storages, if a single storage is unknown, then all of the storages would be processed as unknown storages.

Illife storages

Another way of storing pixels in memory is to use an Illife vector (Group 2000). The Illife vector is a vector which stores pointers to rows in an image. An illustration of an Illife vector is shown in Figure 9.9.

In VVIS, Illife storage refers to storages where the rows are stored independently from each other. How the rows are stored is not the concern of the Illife storage. Unlike contiguous or unknown storage, the Illife storage actually represents a *n*-dimensional storage. It is *n*-dimensional because there is no constraint on the storage type of the rows; the storage type of a row can also be an Illife storage. Some algorithms, like convolutions, will require the Illife storage to be 2-D though.

Figure 9.9 An Illife vector



The Illife storage originally started with an interface similar to contiguous storage's, except that the `begin` and `end` functions accepted an `int` which represented a row. However, implementation-wise, the Illife storage algorithm is usually just the contiguous or unknown algorithm applied a row at a time instead of an image at a time. This `begin(int)`, and `end(int)` prototype meant that it was more difficult to call the contiguous algorithm for a row directly, because algorithms accept storages and not iterators. The main advantage of this interface is the the same class can implement both the Illife and a 1-D storage interface.

To counter this, the `begin` and `end` functions return iterators which on dereferencing will return a storage for the current row. This interface is actually the same as the container interface from STL. Using such an interface, the `for_each` algorithm could be implemented in the following manner:

```
template<
  typename regionInT,
  typename accessorT,
  typename functorT
> void for_each(const regionInT& in, accessorT a, functorT f) {
  for(typename regionInT::iterator i = in.begin();
      i != in.end();
      ++i) {
    // Call for_each for each row
    for_each(*i, a, f);
  }
}
```

The Illife storage interface uses some of the same functions as the contiguous storage interface. However, these functions do different things. For example, in contiguous and unknown storages, `begin()` returns an iterator that points to the beginning of image data, while in Illife storages, `begin()` returns an iterator pointing to the first row. Because of this, a storage that supports the Illife storage interface cannot support the contiguous storage interface.

A contiguous storage can not always be an Illife storage with contiguous storages, because the beginning of a row would not necessarily be aligned correctly. A contiguous storage can only be an Illife storage with contiguous storages when the number of pixels in a row is exactly divisible by the number of scalars in a `vvm::vector`. A contiguous storage however can always be an Illife storage with unknown storages. In addition, an unknown storage can also be an Illife storage with unknown storages and vice versa. An Illife storage might be able to pose as a contiguous storage if it used an iterator like Ablavsky et al. (2003)'s and was able to join the right edge of one row with the left edge

Table 9.9 Illife storage's required interface

Expression	Return Type	Notes
`X::value_type`	`T`	`T` is assignable
`X::iterator`	iterator type pointing to `T`	Convertible to `X::const_iterator`
`X::const_iterator`	iterator type pointing to `const T`	
`a.begin()`	`iterator`; `const_iterator` for constant `a`	
`a.end()`	`iterator`; `const_iterator` for constant a	

Table 9.10 Illife storages' required iterator interface

Expression	Return Type	Notes
`i = j`	`iterator&`	After operation, i != j is false
`i != j`	`bool`	
`++i; i++`	`iterator&`	Moves down by 1 row
`i += j`	`iterator&`	Moves down by j rows
`i + j`	`iterator`	Returns an iterator j rows down
`--i; i--`	`iterator&`	Moves up by 1 row
`i -= j`	`iterator&`	Moves up by j rows
`i - j`	`iterator`	Returns an iterator j rows up
`*i`	`value_type`	Returns the current row
`i[n]`	`value_type`	Returns the row `n` rows down

of the next. However after performing this join, the next row would have to be loaded unaligned.

It should however be possible to create an Illife storage of unknown storages wrapper around a contiguous storage, if required. In addition, regions can be used by the user to convert the storage types if required.

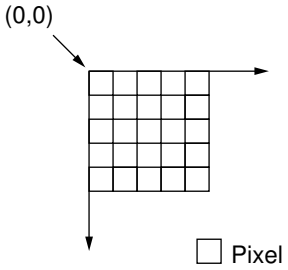
The interface for Illife storages is shown in Table 9.9. Since the Illife storage interface is quite similar to STL containers, it should be possible to use STL containers to house each storage of a row easily. The `begin` function returns an iterator referring to the first row's storage. The `end` function returns an iterator that refers to the first past-the-end row. The result of the expression `begin() == end()` is `true` when there are no rows in the Illife storage.

The iterator's requirements is shown in Table 9.10. The iterator supports the +, - and `[]` operators. This is required for easy access to other rows positioned relatively to the current row. An algorithm that requires such functionality is the convolutive algorithm.

Table 9.11 Template metafunctions for discovering the storage type

Expression	Return Type	Notes
`uses_contiguous_storage<T>::value`	`bool`	Returns true if `T` is a contiguous storage
`uses_illife_storage<T>::value`	`bool`	Returns true if `T` is an Illife storage
`uses_unknown_storage<T>::value`	`bool`	Returns true if `T` is neither contiguous nor Illife

Figure 9.10 Pixels are labelled with (0, 0) in the top-left corner. The value of *x* increases from left to right, while *y* increases from top to bottom.



Discovering storage types

Table 9.11 shows VVIS template metafunctions that determine the storage's type. Because they are template metafunctions, they can be used at compile-time. This allows algorithms to make their decision on how to process storages at compile-time. A storage does not have to specify that it supports an unknown storage interface. Any storage that is not contiguous or Illife is treated as unknown. Since VVIS currently cannot detect storage types at run-time, writing storages whose type changes at run-time is impossible.

9.3.3 Images

Images are two-dimensional representations of a scene composed of pixels. Pixels are often square and have values which represent the light intensity at a point. They can be referenced individually via an *x* and *y* index. The origin of images in VVIS is located in the top-left corner of an image. The values of *x* and *y* increase from left to right and top to bottom respectively. Many image-processing libraries such as VIGRA (Köthe 2001), IMAQ (Nat 1999) and the Microsoft Vision SDK (Group 2000) all label pixels in this manner. However, some libraries, like the Microsoft Vision SDK (Group 2000) allow the origin to be moved from its default top-left corner. For a visual representation of this pixel labelling scheme, refer to Figure 9.10. All images in VVIS are rectangular.

Figure 9.11 VVIS image hierarchy

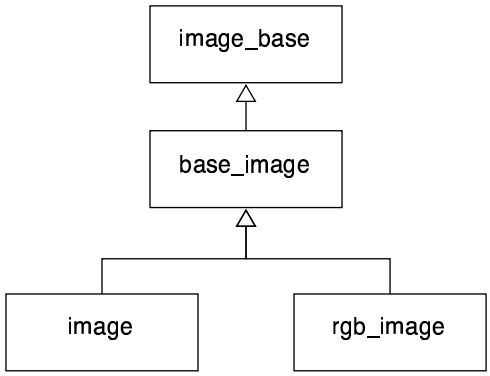


Table 9.12 VVIS storage policies

Storage Policy	Description
`contiguous_planar_storage`	Keeps data in a contiguous planar format
`contiguous_chunky_storage`	Keeps data in a contiguous chunky format
`illife_planar_storage`	Keeps data in an Illife storage of contiguous planar storages
`illife_chunky_storage`	Keeps data in an Illife storage of contiguous chunky storages

The VVIS image hierarchy is shown in Figure 9.11. The `image_base` type contains definitions that all images must have. It does not have any implementation. The `base_image` type provides the implementation of a general multi-channel image. The `image` type represents single-channel images, while `rgb_image` represents RGB images.

Since VVIS images implement the storage interface, images can be passed directly to algorithms. The exact storage type of an image in VVIS is dictated by a storage policy. VVIS provides four different storage policies: `contiguous_planar_storage`, `contiguous_chunky_storage`, `illife_planar_storage`, and `illife_chunky_storage`. The default storage policy is `illife_chunky_storage`. See Table 9.12 for a brief description of each storage policy.

The `base_image` type's declaration is as follows:

```
template<
  typename typelistT,
  typename storageP = illife_chunky_storage
> class base_image<typelistT, storageP>;
```

The template parameter `typelistT` is a typelist containing the types for the different channels. If `typelistT` contains only one type, then the pixel type is that type. Otherwise, the pixel type is a `ct::tuple<typelistT>`. This tuple is how VVIS represents multi-channel pixels. Using a typelist to specify channel types allows the same declaration to

Table 9.13 The interface provided by `base_image`

Expression	Return Type	Notes
`A::component_tl`	Type	Component Typelist
`A::pixel_type`	Type	Pixel Type
`a(w, h)`		Creates an image of size `w`×`h`
`a.component<i>(x, y)`	`i`th type in `component_tl`	Returns the `i`th component of pixel at (`x`, `y`)
`a.pixel(x, y)`	`pixel_type`	Returns pixel at (`x`, `y`)

support both single-channel and multi-channel images. In addition, it allows different channels to have different types. Coupled with storage policies, the same declaration also supports planar and chunky images. For example, a RGBA image with `unsigned char` for RGB channels, but `bool` for the alpha channel can be declared as:

```
typedef base_image<CT_TYPELIST4(unsigned char,
                                unsigned char,
                                unsigned char,
                                bool)> rgba_image_t;
```

The interface provided by `base_image` is summarised in Table 9.13.

The types `image` and `rgb_image` are children of `base_image` that allow the user to create single-channel and RGB images without having to use a typelist to specify each channel's type. In addition, they provide additional functions to access channels in a more intuitive manner. For example, `rgb_image` adds `red`, `green`, and `blue` functions. If `base_image` was used directly, the user would have to remember that channels 0, 1 and 2 are red, green and blue respectively.

9.3.4 Regions

Regions are storages that represent regions of interest in an image. Since they are storages, they have to ensure that data are kept in a particular format. To support this, regions can either have an admit/release phase, or perform unaligned loads and stores automatically. Like VSIPL, after admittance, the portion of the image referred to by the region should not be used by the user until it has been released. An image can have more than one portion admitted to different regions at one time, as long as the portions do not overlap.

Like images, regions were implemented using storage policies. Since regions are storages, regions can be passed to algorithms. A region can have a different storage policy from the image (or region) that it is admitting. Because of this, you can use the region to convert between storage types using admittance if required. Storage formats can also be converted by copying pixels from one storage to another, using a transformative algorithm with a functor that did not change the source value.

9.3.5 Accessors

In the STL, data elements are accessed via references to the original data returned by `operator*` and `operator[]`. The problem with this approach is that for multi-channel planar images, there is no efficient way of returning a reference to a pixel with all the channels. Returning pixel references efficiently from chunky images or from single-channel planar images is possible. To access a pixel that contains all the channels of a multi-channel planar image, a proxy object is required. This proxy object reduces the efficiency of accessing the data elements (Kühl & Weihe 1997).

Accessors, which were introduced by Kühl & Weihe (1997), solve this problem by providing an extra level of indirection. Unlike accessors in VIGRA or those described in Kühl & Weihe (1997) that provide access to a single type, accessors in VVIS provide access to two types — a scalar and a vector type. In addition, apart from being responsible for retrieving and writing data, and performing any necessary type conversions, accessors in VVIS are also responsible for prefetching. Read and write accessor requirements for contiguous storage are shown in Tables 9.14 and 9.15, respectively. For unknown storages, `A::prefetch_channel_count`, `A::vector_type`, `a.prefetch_read<ch>(i)`, `a.prefetch_write<ch>(i)`, `a.get_vector(i)`, `a.get_vector(i, o)`, and `a.set_vector(v, i)` do not need to be implemented. VVIS comes with two accessors — `pixel_accessor` and `component_accessor`. The accessor `pixel_accessor` reads and writes complete pixels while `component_accessor` reads and writes a single component in a pixel.

VVIS accessors provide get and set functions for a scalar and a vector type. Apart from providing functions that load a scalar and a vector from an iterator, or an offset from an iterator, VVIS accessors also provide a function that loads a vector from two vectors originally loaded by the accessor. This extra vector load function is used for unaligned loading.

For scalar types, type conversions are usually automatically performed by the compiler. For `vvm::vector` types however, type conversions must be specified explicitly. The accessors provided by VVIS all perform explicit `vvm::vector` type conversions automatically on write. To prevent the compiler from having to perform explicit type conversions when none is necessary, the accessors provided by VVIS overload `a.set_vector(v, i)` with a template function.

Prefetching is handled by accessors because only accessors know exactly what data are being loaded and stored. For example, `component_accessor` would only need to prefetch a single channel instead of all the channels. Prefetching functions are required by both read and write accessors.

The types `scalar_type` and `vector_type` return the scalar and `vvm::vector` type used by the accessor. This is useful if the algorithm wishes to keep a cache of the values returned from the accessor. The number of prefetch channels required when prefetching

Table 9.14 Read accessor's required interface

Expression	Return Type	Notes
`A::prefetch_channel_count`	`int`	Returns number of prefetch channels required
`A::scalar_type`	Type	Scalar type
`A::vector_type`	Type	`vvm::vector` type
`a.prefetch_read<ch>(i)`	`void`	Prefetch from iterator `i` for read using channel `ch`
`a.get_scalar(i)`	scalar	Returns the scalar at iterator `i`
`a.get_scalar(i, o)`	scalar	Returns the scalar at iterator `i+o` scalar steps
`a.get_vector(i)`	vector	Returns the vector at iterator `i`
`a.get_vector(i, o)`	vector	Returns the vector at iterator `i+o` vector steps
`a.get_vector(a, b, o)`	vector	Returns the vector obtained by ....

For unknown storages, `A::prefetch_channel_count`, `A::vector_type`, `a.prefetch_read<ch>(i)`, `a.get_vector(i)`, and `a.get_vector(i, o)` do not need to be implemented.

Table 9.15 Write accessor's required interface

Expression	Return Type	Notes
`A::prefetch_channel_count`	`int`	Returns number of prefetch channels required
`A::scalar_type`	Type	Scalar type
`A::vector_type`	Type	`vvm::vector` type
`a.prefetch_write<ch>(i)`	`void`	Prefetch from iterator `i` for writing using channel `ch`
`a.set_scalar(s, i)`	`void`	Writes scalar `s` to iterator `i`
`a.set_vector(v, i)`	`void`	Writes vector `v` to iterator `i`

For unknown storages, `A::prefetch_channel_count`, `A::vector_type`, `a.prefetch_write<ch>(i)`, and `a.set_vector(v, i)` do not need to be implemented.

Table 9.16 VVIS algorithms

Expression	Return Type	Notes
`vvis::convolute(in, ia, out, oa, f)`	`void`	Convolutive algorithm
`vvis::for_each(in, ia, f)`	`void`	Quantitative algorithm
`vvis::transform(in, ia, out, a, f)`	`void`	Transformative algorithm
`vvis::transform(in1, ia1, in2, ia2, out, oa, f)`	`void`	Transformative algorithm

is stored in `prefetch_channel_count`. If you want to prefetch from multiple storages, then be sure to pass the `prefetch_channel_count` of the previously prefetched storage to the prefetch functions, so that they do not reuse the same prefetch channels. The following example illustrates how to prefetch more than one storage.

```
// Assume there are three accessor types, A1, A2, A3
// Assume a1, a2, a3 are of types A1, A2, A3
// Assume there are three iterators i1, i2, i3
// Assume i1 and i2 are for reading, i3 is for writing
a1.prefetch_read<0>(i1);
a2.prefetch_read<A1::prefetch_channel_count>(i2);
a3.prefetch_write<A1::prefetch_channel_count +
                  A2::prefetch_channel_count>(i3);
```

9.3.6 Algorithms

Algorithms are responsible for coordinating how the rest of the concepts are used to solve a problem. In Section 8.2, three categories, convolutive, quantitative and transformative, were identified for use with VVIS. The `convolute`, `for_each` and `transform` algorithms are for convolutive, quantitative and transformative categories respectively. Table 9.16 shows the algorithms that VVIS supports. There are two versions of the `transform` algorithm to handle one and two input sets. Like VIGRA, algorithms accept an accessor for each input or output set. Without an accessor for each input or output set, it is more difficult to perform operations on images that involve different channels. The algorithms will be discussed in more detail later, in Chapter 10. Details will be given on how it is implemented, how they compare to hand-coded scalar and AltiVec versions and the functor requirements. Implementation issues common to all algorithms are discussed in this section.

All algorithms are expected to process data differently depending on the storage type. An algorithm in VVIS does not have to provide an implementation for all three storage

types. For transformative and quantitative operations, algorithms must provide an implementation for the Illife (with row storages of any type) and unknown storages. For convolutive operations, algorithms must provide only an Illife (with unknown row storages) implementation. Implementations for contiguous storages are not required because contiguous storages also implement the unknown storage interface.

Selecting algorithm implementation at compile-time

Different algorithm implementations can be selected at compile-time by using tag-dispatching (see Section 4.5), or function enablers (see Section 4.6). The appropriate tag or function enabler can be selected at compile-time by using the template metafunctions for storage type discovery discussed in Section 9.3.2. Selecting the correct implementation to use at compile-time should lead to faster programs. While different algorithms can select their implementations differently, usually, we would select contiguous and Illife before unknown.

An example of a template metafunction that selects an implementation is shown below:

```
namespace priv {
  struct using_contiguous_storage {};
  struct using_illife_storage {};
  struct using_unknown_storage {};

  template<typename typelistT> struct fastest_storage {
    typedef typename boost::mpl::if_c<
      ct::all<typelistT, uses_contiguous_storage>::value,
      using_contiguous_storage,
      typename boost::mpl::if_c<
        ct::all<typelistT, uses_illife_storage>::value,
        using_illife_storage,
        using_unknown_storage
        >::type
      >::type type;
  };
} // End of priv namespace
```

The metafunction `fastest_storage` accepts a typelist of storages. If all the storages are contiguous or Illife storages, then it returns `using_contiguous_storage`, or `using_illife_storage` respectively. Otherwise, `fastest_storage` will return `using_unknown_storage`. Since Illife storages are not unknown or contiguous storages, if Illife storages are mixed with contiguous or unknown storages, the algorithm will not compile.

Once we have a template metafunction that returns a tag representing the version we should be using, we can implement the algorithm using enablers as shown below:

```
namespace priv_unknown_storage {
  template<
    typename storageInT,
    typename accessorInT,
    typename functorT
  > typename ct::enable_if<
    boost::is_same<
      typename priv::fastest_storage<
        CT_TYPELIST1(storageInT)>::type,
      priv::using_unknown_storage
    >::value
  >::type
  for_each(const storageInT& in, accessorInT ia, functorT f) {
    // Unknown for_each implementation goes here
  }
} // End of priv_unknown_storage namespace

namespace priv_illife_storage {
  template<
    typename storageInT,
    typename accessorInT,
    typename functorT
  > typename ct::enable_if<
    boost::is_same<
      typename priv::fastest_storage<
        CT_TYPELIST1(storageInT)>::type,
      priv::using_illife_storage
    >::value
  >::type
  for_each(const storageInT& in, accessorInT ia, functorT f) {
    // Illife for_each implementation goes here
  }
} // End of priv_illife_storage namespace

namespace priv_contiguous_storage {
  template<
    typename storageInT,
```

```
    typename accessorInT,
    typename functorT
  > typename ct::enable_if<
    boost::is_same<
      typename priv::fastest_storage<
        CT_TYPELIST1(storageInT)>::type,
      priv::using_contiguous_storage
    >::value
  >::type
  for_each(const storageInT& in, accessorInT ia, functorT f) {
    // Contiguous for_each implementation goes here
  }
} // End of priv_contiguous_storage namespace

using priv_unknown_storage::for_each;
using priv_contiguous_storage::for_each;
using priv_illife_storage::for_each;
```

Each `for_each` version is in a different namespace because Apple GCC 3.1 20021003 fails to compile the program if all the functions are in the same namespace. This workaround technique was discussed in Section 4.6.

Illife algorithms

The Illife algorithm's implementation is usually quite trivial, since it will usually use the algorithms for the other storage types. As an example, the `for_each` Illife algorithm is presented in Algorithm 9.1. The Illife algorithms for `transform` and `convolute` are similar, expect they take different arguments and call different algorithms.

Calculating the start of the right edge

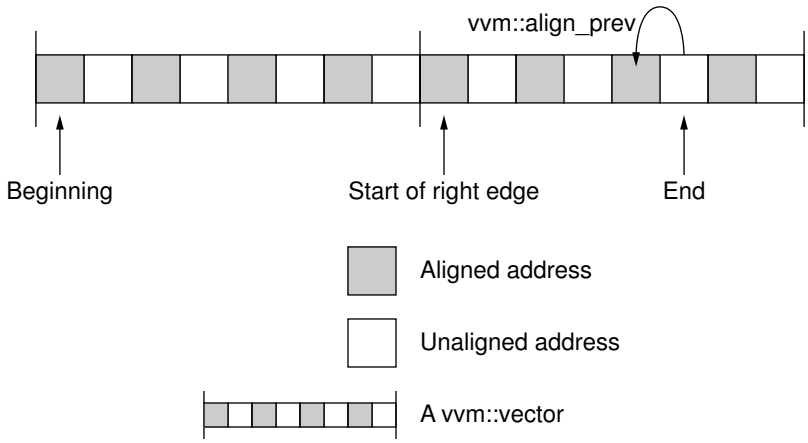
Calculating the right edge is only applicable to contiguous storages. In VVIS, all edges were computed using the scalar processor. Knowing where the right edge is allows us to deduce which portion of the storage should be processed using the VPU and which portion uses the scalar processor.

Originally, the author intended to use `vvm::align_prev` to calculate the right edge's start location from `end`. While this method would be correct if the VPU were used directly, as Figure 9.12 shows, this would be erroneous for VVIS, because a `vvm::vector` spans more than one aligned memory address. The `vvm::align_prev` function can be used to calculate the start of the right edge when using AltiVec `char` types and with a scalar count

Algorithm 9.1 `for_each` Illife implementation

```
template<
  typename storageInT,
  typename accessorInT,
  typename functorT
> typename ct::enable_if<
  boost::is_same<
    typename priv::fastest_storage<
      CT_TYPELIST1(storageInT)>::type,
    priv::using_illife_storage
  >::value
>::type
for_each(const storageInT& in, accessorInT ia, functorT f) {
  for(int y = 0; y < in.height(); ++y) {
    vvis::for_each(in.row(y), ia, f);
  }
}
```

Figure 9.12 Comparison of VVIS and VVM `align_prev` behaviour



of sixteen. Instead, the right edge's location should be calculated using the following formula:

```
begin + ((begin - end) / VVM_SCALAR_COUNT)
```

The introduction of orthogonal `vvm::vector` and scalar components to the iterator, removes the need to calculate the start of the right edge. When the algorithm iterates through all the `vvm::vector` components, the remaining portion is the right edge, as illustrated by the following code:

```
for(; begin.vector != end.vector; ++begin.vector) {
  // vvm::vector portion
}
```

```
for(; begin.scalar != end.scalar; ++begin.scalar) {
  // scalar portion
}
```

9.3.7 Functors

Different categories have different functor requirements. Quantitative and transformative functors both accept input via a call to `operator()`. Transformative functors however are expected to return their answers, while quantitative functors keep their answers. Convolutive functors have a more complex interface because they accumulate input one pixel at a time and then return the answer for that set of input. Functor requirements are discussed in more detail in Chapter 10. In this section, issues common to all functors are discussed. The issues discussed are the need for both a scalar and a `vvm::vector` implementation, determining types automatically and binders.

The need for a scalar and a vvm::vector implementation

VVIS requires functors to provide two implementations for the same operation, one for the scalar processor and one for the VPU. The scalar version is used to handle edges and in situations where the storage format is not suitable for vector processing (when processing unknown storages).

Because VVM provides many of the same operations for `vvm::vectors` as scalars, in many cases it is possible implement a single templated version for both versions. In some cases however, the scalar and `vvm::vector` implementations are not identical. Functors using comparison and logical operators and `vvm::vector` specific functions will usually require different scalar and `vvm::vector` implementations because the scalar and `vvm::vectors` functions are not identical. An example of a non-identical functor would be the `less` functor which is shown below:

```
template<typename T = void> struct less
: public std::binary_function<
  T, T, typename vvm::scalar_traits<T>::bool_type
> {
  typename vvm::scalar_traits<T>::bool_type
  operator()(const T& a, const T& b) const {
    return a < b ? VVM_TRUE : VVM_FALSE;
  }
  vvm::vector<typename vvm::scalar_traits<T>::bool_type>
  operator()(const vvm::vector<T>& a,
             const vvm::vector<T>& b) const {
```

```
    return a < b;
  }
};
```

The `less` functor is non-identical because `vvm::vector`'s `operator<` returns `~0` for `true` while the scalar `operator<` returns 1 for `true`.

Having two implementations for each functor, while required, makes it impossible to use STL binders with VVIS functors. Because STL binders cannot be used, VVIS provides its own binders. Having two implementations also prohibits the use of Boost's Lambda library to generate functors from expressions automatically. While VVIS currently does not provide a substitute, it should be possible to create a replacement using expression templates.

Functors determine types automatically

Since VVM provides many of the same operations for `vvm::vectors` as for scalars, in many cases it is possible implement a single templated version for both versions. For example, the `plus` functor can be implemented as follows:

```
struct plus {
  template<typename T>
    const T operator(const T& a, const T& b) const {
      return a + b;
    }
};
```

Such an implementation does not require the user to specify a type for `plus`, and makes it impossible for the user to specify the wrong type. These make `plus` easier to use.

This `plus` functor is not adaptable, and therefore cannot be used with adaptors like binders, because it does not contain information about its arguments and return type (Meyers 2002). Binders require argument and return types when a functor is constructed. This information is only available when the `plus` functor is used and not when it is constructed. While the binder can decide the argument types when it is used, determining the return type is not possible. To solve this problem, we can create functors whose argument and return types can both be determined automatically or specified explicitly by the user, using techniques described in (Meyers 2002). Shown below is the `plus` functor reimplemented to support both automatic type determination and explicit type specification.

```
template<typename T = void> struct plus
: public std::binary_function<T, T, T> {
  T operator()(const T& a, const T& b) const {
```

```
    return a + b;
  }
  vvm::vector<T> operator()(const vvm::vector<T>& a,
  const vvm::vector<T>& b) const {
    return a + b;
  }
};
template<> struct plus<void> {
  template<typename T>
  T operator()(const T& a, const T& b) const {
    return a + b;
  }
};
```

VVIS functors that can determine the required types automatically are:

```
plus              minus             multiplies
divides           modulus           negate
equal_to          not_equal_to      greater
less              greater_equal     less_equal
logical_and       logical_or        logical_not
plus_saturated    minus_saturated   bitwise_and
bitwise_not       bitwise_or        constant
```

VVIS functors that cannot determine the required types automatically are:

```
equalize          linear_filter     max_filter
```

The functors cannot determine their type automatically because they keep additional information. For example, `equalize` requires a range. If a type is not specified for the range, then `equalize` would introduce conversion costs during execution if the type chosen for the range is not the type that was required.

Binders

Binders require the functor to be instantiated with a type. Functors in VVIS are parameterised using their scalar type. The vector type is deduced by applying the template metafunction `vvm::add_vector` to the scalar type. Using `vvm::add_vector` is preferable to simply using `vvm::vector` because the scalar type `T` might be `const` or a `ct::tuple`. When `vvm::add_vector` encounters a `const T`, it removes the `const`. When `T` is a `ct::tuple`, `vvm::add_vector` returns a `ct::tuple` of `vvm::vectors`. Note that the

VVM implementation used cannot handle `const vvm::vectors` because it did not initialise the new `vvm::vector` using the constructor.

Because there are two implementations for each functor, STL C++ binders cannot be used with VVIS functors. Their reimplementation is straightforward though, once we establish that functors are parameterised using their scalar type.

```
template<typename opT>
class binder1st
: public std::unary_function<
  typename opT::second_argument_type,
  typename opT::result_type
> {
public:
  binder1st(const opT& op,
            const typename opT::first_argument_type& value)
  : _op(op), _value(value) {
  }
  typename opT::result_type
  operator()(const typename opT::second_argument_type& a) const {
    return _op(_value, a);
  }
  typename vvm::add_vector<typename opT::result_type>::type
  operator()(const typename vvm::add_vector<
  typename opT::second_argument_type>::type& a) const {
    return _op(_value, a);
  }
protected:
  opT _op;
  typename opT::first_argument_type _value;
};
template<typename opT, typename T>
binder1st<opT> bind1st(const opT& op, const T& x) {
  return binder1st<opT>(op, x);
}
```

For single-channel images, the pixel value is passed to the functor. For multi-channel images however, the components are packaged together in at tuple (`ct::tuple` to be exact).

The binders provided by VVIS are as follows:

```
binder1st          bind1st
```

```
binder2nd          bind2nd
```

Like STL, `binder1st` and `binder2nd` are binder types, while `bind1st` and `bind2nd` are functions that facilitate the creation of `binder1st` and `binder2nd` respectively.

9.4 Import/Export

VVIS's import/export interface consists of three main classes — `fimporter`, `fexporter` and `sgimporter`. The `fimporter` and `fexporter` classes read and write from and to image files respectively. The `sgimporter` class grabs images from a sequence grabber that is connected to the computer.

VVIS imports, exports and captures images using QuickTime. However, QuickTime usually provides images in a chunky scalar format. While VVIS can process QuickTime images directly as unknown storages, the VPU is not used with unknown storages. To process the image using the VPU, VVIS has to convert the chunky scalar image to an appropriate storage format. Four different conversion methods, using `GetCPixel`, using `GetPixBaseAddr`, using `GetPixBaseAddr` with AltiVec, and using `GetPixBaseAddr` with VVM, were implemented and evaluated. AltiVec and VVM dechunkify and chunkify functions also need to handle unalignment.

While QuickTime supports planar images, the author was unable to capture directly to planar images. In addition, since QuickTime does not support chunky `vvm::vector` images, conversion routines might still be required. Moreover since video cards usually use a chunky scalar format, QuickTime might still need to perform conversions internally even if it captured directly to planar images.

9.4.1 GetCPixel

The easiest and most portable method of getting pixels from a GWorld is to use the `GetCPixel` function (App 2002*d*). This function returns a `RGBColor` structure (App 2002*d*) which contains values for red, green and blue as 16-bit integers. Since they are 16-bit integers, they need to be scaled if the target destination does not use 16-bit integers to represent the red, green and blue components. `GetCPixel` is portable because the same code not only works on any GWorld regardless of the internal representation of the pixels, it also works with on-screen GWorlds. None of the other methods discussed work with on-screen GWorlds.

Converting a GWorld to a VVIS image using `GetCPixel` is shown below:

```
// Assume width & height refer to the width & height of the image
const float uc_max =
```

```
  (float)std::numeric_limits<unsigned char>::max();
const float us_max =
  (float)std::numeric_limits<unsigned short>::max();
// Keep original port and device
CGrafPtr orig_port;
GDHandle orig_device;
GetGWorld(&orig_port, &orig_device);
// Convert
SetGWorld(gworld, NULL);
for(int y = 0; y < height; ++y) {
  for(int x = 0; x < width; ++x) {
    RGBColor c;
    GetCPixel(x, y, &c);
    red[y][x] = (unsigned char)((c.red / us_max) * uc_max);
    blue[y][x] = (unsigned char)((c.blue / us_max) * uc_max);
    green[y][x] = (unsigned char)((c.green / us_max) * uc_max);
  }
}
// Restore original port and device
SetGWorld(orig_port, orig_device);
```

Converting VVIS images back to a GWorld is shown below:

```
// Assume width & height refer to the width & height of the image
const float uc_max =
  (float)std::numeric_limits<unsigned char>::max();
const float us_max =
  (float)std::numeric_limits<unsigned short>::max();
// Keep original port and device
CGrafPtr orig_port;
GDHandle orig_device;
GetGWorld(&orig_port, &orig_device);
// Convert
SetGWorld(gworld, NULL);
for(int y = 0; y < height; ++y) {
  for(int x = 0; x < width; ++x) {
    RGBColor c;
    c.red = (unsigned short)((red[y][x] / uc_max) * us_max);
    c.blue = (unsigned short)((blue[y][x] / uc_max) * us_max);
```

```
      c.green = (unsigned short)((green[y][x] / uc_max) * us_max);
      SetCPixel(x, y, &c);
    }
  }
  // Restore original port and device
  SetGWorld(orig_port, orig_device);
```

While this code is easy to write and highly portable, it is also rather slow. A quick look ahead to Table 9.17 shows that this method is abysmally slow, being unable to keep up with the QuickTime capture process, even when capturing 320×240 images.

9.4.2 GetPixBaseAddr

Othmer & Reed (1992) discussed methods for accessing data in off-screen GWorlds quickly. Instead of using `GetCPixel`, the pixmap structure can be accessed directly. When using this method, there is no need to call `GetCPixel`, nor is there any need to use `RGBColor`. QuickTime supports a number of different formats (see App (2002*f*), IV-2862 for a list of formats), and selecting one that closely matches the target image reduces the time required to massage the data to the required format. For example, appropriate pixel formats for converting to `unsigned char` RGB images using the method discussed in this section are either 24-bit RGB or 32-bit RGBA.

To access the pixels directly, we have to obtain the `PixMap` from the GWorld using the `GetGWorldPixMap` (App 2002*d*) function. After obtaining a handle to the `PixMap`, `LockPixels` (App 2002*d*) should be called. `LockPixels` prevents QuickTime from moving the GWorld's data while the program is accessing the pixels directly. Since programs typically decompress to the same image, the `LockPixels` function might only need to be issued once after the GWorld is created. `GetPixBaseAddr` (App 2002*d*) provides an address to the raw image data in memory. `GetPixRowBytes` (App 2002*d*) returns the number of bytes per row. In Windows, since `GetPixRowBytes` is not available, `(**pixmap).rowBytes & 0x3fff` can be used instead. After obtaining the address of the first pixel, and the number of bytes per row, we iterate through memory and copy the components to separate channels. The meaning of the pixels depends on the pixel format of the GWorld.

The code below dechunkifies a `k32ARGBPixelFormat` GWorld. While a 24-bit RGB pixel format could have been chosen, `k32ARGBPixelFormat` was used to enable all methods discussed to use the same pixel format. Using `GetPixBaseAddr` with AltiVec and VVM requires 32-bit images.

```
// Assume width & height refer to the width & height of the image
```

```
// Convert from k32ARGBPixelFormat
PixMapHandle p = GetGWorldPixMap(gworld);
LockPixels(p);
unsigned char* addr = (unsigned char*)GetPixBaseAddr(p);
// row_bytes is the number of bytes in a row of the GWorld image
unsigned short row_bytes = GetPixRowBytes(p);
for(int y = 0; y < height; ++y) {
  for(int x = 0, i = 0; x < width; ++x, i += 4) {
    // If we had wanted alpha
    // alpha[y][x] = addr[i];
    red[y][x] = addr[i+1];
    green[y][x] = addr[i+2];
    blue[y][x] = addr[i+3];
  }
  addr += row_bytes;
}
UnlockPixels(p);
```

Converting the data back to the GWorld involves writing directly to the memory location returned by the `GetPixBaseAddr`. The alpha channel should be set to `0xff` to prevent the image from being transparent.

```
// Assume width & height refer to the width & height of the image
// Remember to call LockPixels on gworld
// Convert to k32ARGBPixelFormat
PixMapHandle p = GetGWorldPixMap(gworld);
LockPixels(p);
unsigned char* addr = (unsigned char*)GetPixBaseAddr(p);
// row_bytes is the number of bytes in a row of the GWorld image
unsigned short row_bytes = GetPixRowBytes(p);
for(int y = 0; y < height; ++y) {
  for(int x = 0, i = 0; x < width; ++x, i += 4) {
    addr[i] = 0xff; // Opaque
    addr[i+1] = red[y][x];
    addr[i+2] = green[y][x];
    addr[i+3] = blue[y][x];
  }
  addr += row_bytes;
}
```

```
UnlockPixels(p);
```

9.4.3 GetPixBaseAddr with AltiVec

Computers with AltiVec (Mot 1999, App 2002*c*, AltiVec.org 2002, Lai & McKerrow 2001) can use AltiVec functions to convert chunky scalar data to planar data more rapidly. An appropriate pixel format for converting to `unsigned char` planar RGB images is `k32ARGBPixelFormat`, a 32-bit RGB format. A 24-bit RGB format is not suitable because it is more difficult to separate 24-bit pixels using AltiVec. Any other 32-bit RGB format, such as `k32BGRAPixelFormat`, would also have be appropriate. Like the version presented in the previous section, the AltiVec version also accesses the image data directly using `GetGWorldPixMap` and `GetPixBaseAddr`. It therefore also requires a call to `LockPixels`. The AltiVec version differs in the manner in the way in which it dechunkifies and chunkifies images.

Since the pixel format chosen was `k32ARGBPixelFormat`, the source image is of the form ARGBARGBARGBARGB. Applying the `vec_pack` function reduces the input to GBGBGBGB. A second pass results in BBBB. To get the other components, we shift the ARGB and GB vectors by 8 and 16 bits respectively using `vec_sro`. For example, shifting the GB vector right by 8 bits produces 0GBGBGBG. Passing this through `vec_pack` gives us GGGG. A 32-bit RGB pixel format was chosen even though the alpha channel was not needed, because this technique can only dechunkify images with pixels that consist of two, four, eight, sixteen, and so on, bytes.

```
// Assume width & height refer to the width & height of the image
// Remember to call LockPixels on gworld
// Convert from k32ARGBPixelFormat
vector unsigned char c16 = (vector unsigned char)(16);
vector unsigned char c8 = (vector unsigned char)(8);
vector unsigned int argb1, argb2;
vector unsigned int argb3, argb4;
vector unsigned short gb1, gb2;
vector unsigned short ar1, ar2;
vector unsigned char r, g, b;
PixMapHandle p = GetGWorldPixMap(gworld);
LockPixels(p);
char* addr = GetPixBaseAddr(p);
// row_bytes is the number of bytes in a row of the GWorld image
unsigned short row_bytes = GetPixRowBytes(p);
for(int y = 0; y < height; ++y) {
```

```
  for(int x = 0, i = 0; x < width; x += 16, i += 4*16) {
    unsigned int* iaddr = (unsigned int*)(&addr[i]));
    argb1 = vec_ld( 0, iaddr);
    argb2 = vec_ld(16, iaddr);
    argb3 = vec_ld(32, iaddr);
    argb4 = vec_ld(48, iaddr);
    gb1 = vec_pack(argb1, argb2);
    gb2 = vec_pack(argb3, argb4);
    vec_st(vec_pack(gb1, gb2), 0, &blue[y][x]);
    vec_st(vec_pack(vec_sro(gb1, c8), vec_sro(gb2, c8)), 0,
           &green[y][x]);
    ar1 = vec_pack(vec_sro(argb1, c16), vec_sro(argb2, c16));
    ar2 = vec_pack(vec_sro(argb3, c16), vec_sro(argb4, c16));
    vec_st(vec_pack(ar1, ar2), 0, &red[y][x]);
    // For if we had wanted alpha
    // vec_st(vec_pack(vec_sro(ar1, c8), vec_sro(ar2, c8)), 0,
    //        &alpha(x, y));
  }
  addr += row_bytes;
}
UnlockPixels(p);
```

Chunkifying is a bit easier than dechunkifying because AltiVec provides two merge operations. The `mergel` interlaces the lower half of the vectors, while `mergeh` interlaces the other half. For example, a `mergel` and `mergeh` on 1234 and 5678 produces 3748 and 1526 respectively.

```
// Assume width & height refer to the width & height of the image
// Remember to call LockPixels on gworld
// Convert to k32ARGBPixelFormat
vector unsigned char gr1, gr2;
vector unsigned char ag1, ag2;
vector unsigned char r, g, b;
vector unsigned char a = (vector unsigned char)(0xff); // Opaque
PixMapHandle p = GetGWorldPixMap(gworld);
LockPixels(p);
char* addr = GetPixBaseAddr(p);
// row_bytes is the number of bytes in a row of the GWorld image
long row_bytes = GetPixRowBytes(p);
```

```
for(int y = 0; y < height; ++y) {
  for(int x = 0, i = 0; x < width; x += 16, i += 4*16) {
    r = vec_ld(0, &red[y][x]);
    g = vec_ld(0, &green[y][x]);
    b = vec_ld(0, &blue[y][x]);
    rb1 = vec_mergeh(r, b);
    rb2 = vec_mergel(r, b);
    ag1 = vec_mergeh(a, g);
    ag2 = vec_mergel(a, g);
    vec_st(vec_mergeh(ag1, rb1), 0, &addr[i]);
    vec_st(vec_mergel(ag1, rb1), 0, &addr[i+16]);
    vec_st(vec_mergeh(ag2, rb2), 0, &addr[i+32]);
    vec_st(vec_mergel(ag2, rb2), 0, &addr[i+48]);
  }
  addr += row_bytes;
}
UnlockPixels(p);
```

The AltiVec chunkify and dechunkify functions presented above assume that the result of `GetPixBaseAddr(p)` is aligned, and that the width of the image is divisible by 16. The address returned by `GetPixBaseAddr(p)` does not actually seem to be guaranteed to be aligned. In addition, the width of the image is definitely not always divisible to 16. The unaligned AltiVec chunkify version, shown below, addresses these problems.

```
// Assume width & height refer to the width & height of the image
PixMapHandle pixmap = GetGWorldPixMap(gworld);
unsigned char* addr = GetPixBaseAddr(pixmap);
// row_bytes is the number of bytes in a row of the GWorld image
long row_bytes = GetPixRowBytes(pixmap);
__vector unsigned char c16 = (__vector unsigned char)(16);
__vector unsigned char c8 = (__vector unsigned char)(8);
__vector unsigned int argb1, argb2, argb3, argb4, argb5;
__vector unsigned short gb1, gb2, ar1, ar2;
// Assuming ARGB
for(int y = 0; y < height; ++y) {
  // Calculate left/right edges
  const int left_edge = (((unsigned long)addr) & 0xf != 0) ?
                        VVM_SCALAR_COUNT : 0;
  const int right_edge = (width - ((width - left_edge) % 16)) +
```

```
                        left_edge;

  int i = 0, x = 0;
  for(; x < left_edge; ++x, i += 4) {
    red[y][x] = addr[i+1];
    green[y][x] = addr[i+2];
    blue[y][x] = addr[i+3];
  }
  __vector unsigned char vmask = vec_lvsl(0, addr);
  argb5 = vec_ld(0, (unsigned int*)&addr[i]);
  for(; x < right_edge; x += 16, i += 4*16) {
    unsigned int* iaddr = (unsigned int*)&addr[i];
    argb1 = argb5;
    argb2 = vec_ld(16, iaddr);
    argb3 = vec_ld(32, iaddr);
    argb4 = vec_ld(48, iaddr);
    argb5 = vec_ld(64, iaddr);
    // Do shift
    argb1 = vec_perm(argb1, argb2, vmask);
    argb2 = vec_perm(argb2, argb3, vmask);
    argb3 = vec_perm(argb3, argb4, vmask);
    argb4 = vec_perm(argb4, argb5, vmask);
    // Chunkify
    gb1 = vec_pack(argb1, argb2);
    gb2 = vec_pack(argb3, argb4);
    vec_st(vec_pack(gb1, gb2), 0, &blue[y][x]);
    vec_st(vec_pack(vec_sro(gb1, c8), vec_sro(gb2, c8)), 0,
           &green[y][x]);
    ar1 = vec_pack(vec_sro(argb1, c16), vec_sro(argb2, c16));
    ar2 = vec_pack(vec_sro(argb3, c16), vec_sro(argb4, c16));
    vec_st(vec_pack(ar1, ar2), 0, &red[y][x]);
  }
  for(; x < width; ++x, i += 4) {
    red[y][x] = addr[i+1];
    green[y][x] = addr[i+2];
    blue[y][x] = addr[i+3];
  }
  addr += row_bytes;
}
```

The value of `left_edge` is either `VVM_SCALAR_COUNT` or `0` depending on if the address is unaligned or aligned respectively. It is always `VVM_SCALAR_COUNT` when unaligned because in VVIS, contiguous storages only have contiguous pixels across a `vvm::vector`. If `left_edge` had been set to the unaligned offset, then the function would have operated correctly only with contiguous planar storages, but not contiguous chunky storages. `VVM_SCALAR_COUNT` was set to 16 to emulate VVIS behaviour in AltiVec mode.

The unaligned AltiVec dechunkify operation is shown below:

```
// Assume width & height refer to the width & height of the image
// AltiVec unaligned chunkify
PixMapHandle pixmap = GetGWorldPixMap(gworld);
unsigned char* addr = GetPixBaseAddr(pixmap);
// row_bytes is the number of bytes in a row of the GWorld image
long row_bytes = GetPixRowBytes(pixmap);
__vector unsigned char rb1, rb2, ag1, ag2;
__vector unsigned char r, g, b;
__vector unsigned char a = (__vector unsigned char)(0xff);
// Assuming ARGB
for(int y = 0; y < height; ++y) {
  // Calculate left/right edges
  const int right_edge = width - (width % 16);

  int i = 0, x = 0;
  for(; x < right_edge; x += 16, i += 4*16) {
    r = vec_ld(0, &red[y][x]);
    g = vec_ld(0, &green[y][x]);
    b = vec_ld(0, &blue[y][x]);
    rb1 = vec_mergeh(r, b);
    rb2 = vec_mergel(r, b);
    ag1 = vec_mergeh(a, g);
    ag2 = vec_mergel(a, g);
    store(vec_mergeh(ag1, rb1), (unsigned char*)&addr[i]);
    store(vec_mergel(ag1, rb1), (unsigned char*)&addr[i+16]);
    store(vec_mergeh(ag2, rb2), (unsigned char*)&addr[i+32]);
    store(vec_mergel(ag2, rb2), (unsigned char*)&addr[i+48]);
  }
  for(; x < width; ++x, i += 4) {
    addr[i] = 0xff;
    addr[i+1] = red[y][x];
```

```
    addr[i+2] = green[y][x];
    addr[i+3] = blue[y][x];
  }
  addr += row_bytes;
}
```

In this case, the source is always aligned, but the destination is not. Thus we can load aligned, but we need to write unaligned. The unaligned write function, `store`, was discussed previously, in Section 3.5.3.

9.4.4 GetPixBaseAddr with VVM

The VVM versions of the AltiVec chunkify and dechunkify operations are discussed in this section. While VVM does not have AltiVec's `vec_pack` functions, it has type conversions, which perform the same operation as `vec_pack`. See Appendix B for more information about what AltiVec functions are used when the VVM implementation converts from `vuint32` to `vuint8`.

```
// Assume width & height refer to the width & height of the image
const int step = VVM_SCALAR_COUNT;
PixMapHandle pixmap = GetGWorldPixMap(gworld);
char* addr = GetPixBaseAddr(pixmap);
// row_bytes is the number of bytes in a row of the GWorld image
long row_bytes = GetPixRowBytes(pixmap);
vvm::vuint32 argb1;
// Assuming ARGB
for(int y = 0; y < height; ++y) {
  for(int x = 0, i = 0; x < width; x += step, i += 4*step) {
    argb1 = vvm::load(reinterpret_cast<vvm::uint32*>(&addr[i]),
                      0);
    vvm::store(vvm::vector_cast<vvm::vuint8>(argb1),
               &image.blue(x, y));
    vvm::store(vvm::vector_cast<vvm::vuint8>(argb1 >> 8),
               &image.green(x, y));
    vvm::store(vvm::vector_cast<vvm::vuint8>(argb1 >> 16),
               &image.red(x, y));
    // Discard alpha channel
  }
  addr += row_bytes;
}
```

The chunkify operation is similar to the AltiVec chunkify function, because VVM also has `mergel` and `mergeh` functions.

```
// Assume width & height refer to the width & height of the image
const int step = VVM_SCALAR_COUNT;
PixMapHandle pixmap = GetGWorldPixMap(gworld);
char* addr = GetPixBaseAddr(pixmap);
// row_bytes is the number of bytes in a row of the GWorld image
long row_bytes = GetPixRowBytes(pixmap);
vvm::vuint8 rb1, rb2, ag1, ag2;
vvm::vuint8 r, g, b;
vvm::vuint8 a(0xff); // Opaque
// Assuming ARGB
for(int y = 0; y < height; ++y) {
  for(int x = 0, i = 0; x < width; x += step, i += 4*step) {
    r = vvm::load(&image.red(x, y));
    g = vvm::load(&image.green(x, y));
    b = vvm::load(&image.blue(x, y));
    rb1 = vvm::mergeh(r, b);
    rb2 = vvm::mergel(r, b);
    ag1 = vvm::mergeh(a, g);
    ag2 = vvm::mergel(a, g);
    vvm::store(vvm::mergeh(ag1, rb1), (uint8*)&addr[i]);
    vvm::store(vvm::mergel(ag1, rb1),(uint8*)&addr[i+step]);
    vvm::store(vvm::mergeh(ag2, rb2),(uint8*)&addr[i+2*step]);
    vvm::store(vvm::mergel(ag2, rb2),(uint8*)&addr[i+3*step]);
  }
  addr += row_bytes;
}
```

The unaligned VVM dechunkify operation is shown below. Like the unaligned AltiVec dechunkify function presented in the previous section, `left_edge` is either `VVM_SCALAR_COUNT` or `0` to support both contiguous planar and contiguous chunky storages. The value of `i` starts from `-offset` to ensure that `&addr[i]` is always aligned, because, as mentioned previously, loading unaligned addresses using `vvm::load` is undefined in VVM.

```
// Assume width & height refer to the width & height of the image
PixMapHandle pixmap = GetGWorldPixMap(gworld);
```

```
unsigned char* addr = GetPixBaseAddr(pixmap);
// row_bytes is the number of bytes in a row of the GWorld image
long row_bytes = GetPixRowBytes(pixmap);
vvm::vuint32 argb1, argb2, argb;
vvm::vuint32 c8(8), c16(16);
// Assuming ARGB
for(int y = 0; y < height; ++y) {
  const int offset = vvm::uoffset(addr);
  // Calculate left/right edges
  const int left_edge = (vvm::uoffset(addr) != 0) ?
                        VVM_SCALAR_COUNT : 0;
  const int right_edge = (width - ((width - left_edge)
                         % VVM_SCALAR_COUNT)) + left_edge;

  // Use -offset to ensure addr[i] is aligned
  int i = -offset, x = 0;
  for(; x < left_edge; ++x, i += 4) {
    red[y][x] = addr[i+1];
    green[y][x] = addr[i+2];
    blue[y][x] = addr[i+3];
  }
  argb2 = vvm::load(reinterpret_cast<vvm::uint32*>(&addr[i]));
  for(; x < right_edge;
      x += VVM_SCALAR_COUNT, i += 4*VVM_SCALAR_COUNT) {
    argb1 = argb2;
    argb2 = vvm::load(reinterpret_cast<vvm::uint32*>(&addr[i])
      + VVM_SCALAR_COUNT);
    argb = vvm::load(argb1, argb2, offset);
    vvm::store(vvm::vector_cast<vvm::vuint8>(argb),
               &blue[y][x]);
    vvm::store(vvm::vector_cast<vvm::vuint8>(argb >> c8),
               &green[y][x]);
    vvm::store(vvm::vector_cast<vvm::vuint8>(argb >> c16),
               &red[y][x]);
  }
  for(; x < width; ++x, i += 4) {
    red[y][x] = addr[i+1];
    green[y][x] = addr[i+2];
    blue[y][x] = addr[i+3];
```

```
  }
  addr += row_bytes;
}
```

The unaligned VVM chunkify operation is shown below.

```
// Assume width & height refer to the width & height of the image
PixMapHandle pixmap = GetGWorldPixMap(gworld);
char* addr = GetPixBaseAddr(pixmap);
// row_bytes is the number of bytes in a row of the GWorld image
long row_bytes = GetPixRowBytes(pixmap);
vvm::vuint8 rb1, rb2, ag1, ag2;
vvm::vuint8 r, g, b;
vvm::vuint8 a(0xff); // Opaque
// Assuming ARGB
for(int y = 0; y < height; ++y) {
  // Calculate left/right edges
  const int right_edge = width - (width % VVM_SCALAR_COUNT);

  int x = 0, i = 0;
  for(; x < right_edge;
      x += VVM_SCALAR_COUNT, i += 4*VVM_SCALAR_COUNT) {
    r = vvm::load(&red[y][x]);
    g = vvm::load(&green[y][x]);
    b = vvm::load(&blue[y][x]);
    rb1 = vvm::mergeh(r, b);
    rb2 = vvm::mergel(r, b);
    ag1 = vvm::mergeh(a, g);
    ag2 = vvm::mergel(a, g);
    store_unaligned(vvm::mergeh(ag1, rb1),
      reinterpret_cast<vvm::uint8*>(&addr[i]));
    store_unaligned(vvm::mergel(ag1, rb1),
      reinterpret_cast<vvm::uint8*>(&addr[i+VVM_SCALAR_COUNT]));
    store_unaligned(vvm::mergeh(ag2, rb2),
      reinterpret_cast<vvm::uint8*>(&addr[i+2*VVM_SCALAR_COUNT]));
    store_unaligned(vvm::mergel(ag2, rb2),
      reinterpret_cast<vvm::uint8*>(&addr[i+3*VVM_SCALAR_COUNT]));
  }
  for(; x < width; ++x, i += 4) {
```

```
      addr[i+1] = red[y][x];
      addr[i+2] = green[y][x];
      addr[i+3] = blue[y][x];
    }
    addr += row_bytes;
  }
```

The unaligned store function implemented using only VVM's aligned functions is shown below:

```
void store_unaligned(const vvm::vuint8& v, unsigned char *p) {
  // Load the surrounding areas
  vvm::vuint8 low = vvm::load(p, 0);
  vvm::vuint8 high = vvm::load(p, 16);
  // Prepare the constants that we need
  vvm::vuint8 pv = vvm::lvsl(p, 0);
  vvm::vuint8 offset(vvm::uoffset(p));
  // Insert our data into the low and
  // high vectors
  low = vvm::select(pv >= offset, low, v);
  high = vvm::select(pv < offset, v, high);
  // Store the two aligned result
  // vectors
  vvm::store(low, p, 0);
  vvm::store(high, p, 16);
}
```

Note that VVM's unaligned functions were added after these results were collated. As a result, the VVM dechunkify and chunkify operations discussed in this section uses only VVM's aligned functions. This is why the `store_unaligned` function was used.

9.4.5 Results

All results were collected on a PowerMac Dual 450 MHz PowerPC G4, with 384 MB of RAM and 1 MB L2 cache for each processor. The camera used was an Orange Micro iBot FireWire. The iBot is capable of delivering frame rates of up to 30 frames per second at 640×480 pixels depending on the configuration of the computer (Ora 2002). Programs were compiled using Apple GCC 3.1 20021003 with the `-Os` (optimise for size) setting.

Tables 9.17 and 9.18 show the capture and maximum conversion rates for different conversion methods at different sizes respectively. The versions tested were:

Decompress Only: This version does not perform any conversions. Since there are no conversions, maximum conversion rate is not applicable.

GetCPixel: This version was discussed in Section 9.4.1.

GetPixBaseAddr: This version was discussed in Section 9.4.2.

AltiVec (Aligned): This refers to the aligned AltiVec chunkify and dechunkify functions discussed in Section 9.4.3.

VVM (Aligned) in Scalar mode: This refers to the aligned VVM chunkify and dechunkify functions discussed in Section 9.4.4, running in scalar mode. In this mode, there is only 1 scalar in each `vvm::vector`.

VVM (Aligned) in AltiVec mode: This refers to the aligned VVM chunkify and dechunkify functions discussed in Section 9.4.4, running in AltiVec mode. In this mode, there are 16 scalars in each `vvm::vector`.

AltiVec (Unaligned): This refers to the unaligned AltiVec chunkify and dechunkify functions discussed in Section 9.4.3.

VVM (Unaligned) in Scalar mode: This refers to the unaligned VVM chunkify and dechunkify functions discussed in Section 9.4.4, running in scalar mode. In this mode, there is only 1 scalar in each `vvm::vector`.

VVM (Unaligned) in AltiVec mode: This refers to the unaligned VVM chunkify and dechunkify functions discussed in Section 9.4.4, running in AltiVec mode. In this mode, there are 16 scalars in each `vvm::vector`.

Table 9.17 shows the capture rate, in frames per second, for the different conversion methods for images of different sizes. For each method, except Decompress Only, two capture rates, read-only and read-write, are shown. Decompress Only's capture rate is equal to the maximum capture rate attainable under the configuration used. Images were captured during a 20 second period, except for GetCPixel and VVM (Unaligned) in Scalar mode, which were run for 60 seconds.

Table 9.17 shows that GetCPixel requires so much processing that not only were programs using GetCPixel unable to keep up with Decompress Only even with 320×240 images, they were more than 5 times slower. For read-only, GetPixBaseAddr, AltiVec (Aligned), VVM (Aligned) in AltiVec mode, AltiVec (Unaligned), and VVM (Unaligned) in AltiVec mode were able to keep up with Decompress Only for images of size 640×480 and smaller. For read-write, only AltiVec (Aligned), and AltiVec (Unaligned) were able to keep up with Decompress Only for images of size 640×480 and smaller.

Table 9.17 Capture rate in frames per second. Where applicable, the two frame rates represent capture rates for read-only and read-write tasks respectively.

	320×240	640×480	800×600
Decompress Only	29.9	29.9	29.9
GetCPixel	5.3, 4.5	1.5, 1.2	1.0, 0.8
GetPixBaseAddr	29.9, 29.9	29.9, 25.8	21.9, 15.6
AltiVec (Aligned)	29.9, 29.9	29.9, 29.9	27.9, 23.4
VVM (Aligned) in Scalar mode	29.9, 29.9	18.4, 8.7	11.3, 5.7
VVM (Aligned) in AltiVec mode	29.9, 29.9	29.9, 29.0	24.5, 18.1
AltiVec (Unaligned)	29.9, 29.9	29.9, 29.9	28.2, 19.1
VVM (Unaligned) in Scalar mode	29.9, 13.4	17.8, 3.5	10.9, 2.3
VVM (Unaligned) in AltiVec mode	29.9, 29.9	29.9, 19.8	18.0, 12.3
	1024×768	1280×960	1600×1200
Decompress Only	25.1	19.6	11.4
GetCPixel	0.7, 0.6	0.6, 0.5	0.8 , 0.6
GetPixBaseAddr	14.1, 9.7	8.7, 6.3	5.8, 4.3
AltiVec (Aligned)	14.1, 9.7	11.1, 9.2	7.4, 6,1
VVM (Aligned) in Scalar mode	7.1, 3.7	4.7, 2.6	3.3, 2.1
VVM (Aligned) in AltiVec mode	15.3, 10.9	10.4, 7.2	6.6, 4.9
AltiVec (Unaligned)	20.3, 11.9	11.4, 7.7	7.5, 5.1
VVM (Unaligned) in Scalar mode	6.7, 1.5	4.3, 1.0	2.8, 0.8
VVM (Unaligned) in AltiVec mode	11.3, 7.7	7.3, 5.1	4.9, 3.5

Table 9.18 Maximum conversion rate in frames per second when operating on a QuickTime image in memory. The two frame rates represent maximum conversion rates for read-only and read-write tasks respectively.

	320×240	640×480	800×600
GetCPixel	8.2, 4.4	2.0, 1.1	1.3 , 0.7
GetPixBaseAddr	387.4, 189.4	76.7, 39.2	47.8, 25.0
AltiVec (Aligned)	1200.6, 587.1	156.5, 88.5	94.5, 54.1
VVM (Aligned) in Scalar mode	93.1, 42.0	22.7, 10.4	14.7, 6.6
VVM (Aligned) in AltiVec mode	497.7, 245.3	97.9, 48.0	59.2, 31.6
AltiVec (Unaligned)	1177.8, 469.0	155.3, 57.8	89.9, 34.9
VVM (Unaligned) in Scalar mode	98.0, 14.1	22.8, 3.5	14.5, 2.2
VVM (Unaligned) in AltiVec mode	233.2, 134.8	49.7, 26.6	31.0, 16.7
	1024×768	1280×960	1600×1200
GetCPixel	0.8, 0.4	0.5 , 0.3	0.3, 0.2
GetPixBaseAddr	28.0, 14.9	17.8, 9.5	11.4, 6.0
AltiVec (Aligned)	55.6, 32.5	35.0, 20.4	22.2, 12.9
VVM (Aligned) in Scalar mode	8.9, 4.0	5.7, 2.6	3.7, 1.7
VVM (Aligned) in AltiVec mode	38.9, 19.0	25.3, 12.2	16.2, 7.6
AltiVec (Unaligned)	54.5, 20.9	34.3, 13.6	22.3, 8.5
VVM (Unaligned) in Scalar mode	8.8, 1.3	5.6, 0.9	3.6, 0.5
VVM (Unaligned) in AltiVec mode	18.8, 10.2	12.1, 6.4	7.7, 4.1

Table 9.18 shows the maximum conversion rate when operating on a QuickTime image in memory. It is calculated in order to provide some indication of the amount of processing power left after conversion. The maximum conversion rate should always be higher than the capture rate for the same method and target image size because the capture rate also involves decompressing and scaling the image obtained from the camera, which the maximum conversion rate does not take into account. In addition, unlike the maximum conversion rate, the capture rate is also influenced by the speed of the camera, and by the speed of the line that connects the camera to the computer. All versions captured 100 frames, except GetCPixel which captured only 10 frames. This was because of its extremely poor performance.

Table 9.18 shows that AltiVec (Aligned) was fastest, followed by AltiVec (Unaligned), VVM (Aligned) in AltiVec mode, GetPixBaseAddr, VVM (Unaligned) in AltiVec mode, VVM (Aligned) in Scalar mode, VVM (Unaligned) in Scalar mode and GetCPixel. As expected, the maximum conversion rate was generally larger than the capture rate.

VVM in AltiVec mode was slower than AltiVec, because the conversion from `vuint32` to `vuint8` is not zero-cost, since the number of elements in a `vuint32` is greater than one. In addition, there are also overheads involved in loading `vuint32` and performing unaligned loads on `vuint32`. Unaligned loads on `vuint32` seem to have particularly high overheads, because while VVM (Aligned) in AltiVec mode was about two times slower

than AltiVec (Aligned), VVM (Unaligned) in AltiVec mode was about five times slower than AltiVec (Aligned).

Since VVM in AltiVec mode was significantly slower than AltiVec, VVIS uses AltiVec (Unaligned) dechunkify and chunkify operations. In the absence of a VPU, VVIS will use GetPixBaseAddr since it was faster than both VVM (Aligned) and VVM (Unaligned) in both Scalar mode. Furthermore, GetPixBaseAddr was faster than VVM (Unaligned) in both AltiVec and Scalar mode.

9.5 Conclusions

In this chapter, the design and implementation of a generic, vectorised, machine-vision library, named Vectorised Vision (VVIS), was discussed. The reason why divisions of duty used in current generic libraries, such as STL and VIGRA, cannot be directly vectorised was discussed. This was followed by an evaluation of two different divisions of duties that would be viable for a generic, vectorised library. The interfaces and general implementation of the different concepts in VVIS were then discussed. This chapter concluded with an investigation into the cost of converting from the chunky scalar format provided through QuickTime to a more VPU-friendly format.

QuickTime allows images to be captured from a camera. Unfortunately, the author was unable to get QuickTime to capture to planar RGB images, which are required for efficient vector processing. Eight different methods of dechunkifying and chunkifying were discussed. AltiVec (Aligned) was fastest, followed by AltiVec (Unaligned), VVM (Aligned) in AltiVec mode, GetPixBaseAddr, VVM (Unaligned) in AltiVec mode, VVM (Aligned) in Scalar mode, VVM (Unaligned) in Scalar mode and GetCPixel. For read-only operations, GetPixBaseAddr, AltiVec (Aligned), VVM (Aligned) in AltiVec mode, AltiVec (Unaligned), and VVM (Unaligned) in AltiVec mode were able to keep up with the QuickTime capture process for images of size 640×480 and smaller. For read-write operations, only AltiVec (Aligned) and AltiVec (Unaligned) were able to keep up for images of size 640×480 and smaller. VVM in AltiVec mode was slower than AltiVec, because the conversion from `vuint32` to `vuint8` is not zero-cost, since the number of VPU vectors in `vuint32` is more than one. In addition, there are also overheads involved in loading `vuint32` and performing unaligned loads on `vuint32`. Since VVM in AltiVec mode was significantly slower than AltiVec, VVIS uses AltiVec (Unaligned) dechunkify and chunkify operations in AltiVec mode and GetPixBaseAddr in scalar mode.

In this chapter, two new different divisions of duties suitable for vectorised generic programming were discussed. Two new concepts, shape and storage, were introduced. In addition, accessors and functors in VVIS use both scalars and `vvm::vectors`. Also new in this chapter is the use of different algorithms and iterators to process for different storages, the contiguous storage iterator, the use of typelists to declare channel types in

an image, and a contiguous chunky storage format suitable for vector programs.

Two new divisions of duties that are suitable for a generic, vectorised library were investigated: Algorithm Only and Vector Head. While these two divisions of duties use the same concepts, they distribute responsibilities differently. Algorithm Only assigns all vector processing responsibilities to the algorithm, while Vector Head distributes vector processing responsibilities between the storage, algorithm and potentially the iterator and accessor. Vector Head was chosen for VVIS, because it provides good performance when whole images are processed, and it simplifies the implementation of algorithms, because storages ensure that there is no left edge.

Shapes allow algorithms to explicitly specify what shapes they can process correctly. Quantitative and transformative algorithms can process storages of any shape, because quantitative and transformative operations do not require spatial iterators (see Section 8.2). For convolutive operations, the algorithms may be valid only for certain shapes.

The storage concept is the cornerstone of a vectorised, generic library because it provides information about how the pixels are arranged relative to each other. Thus it provides the information that is required to decide how the pixels can be loaded efficiently into a VPU. Furthermore storages allow non-rectangular regions to be specified if required. Three different storages were introduced: contiguous, unknown and Illife. Contiguous storages can be processed efficiently using a VPU, while unknown storages should be processed using the scalar processor. Illife storages are *n*-dimensional; their iterators provide access to other storages.

Unlike accessors and functors from existing libraries like STL and VIGRA, accessors and functors in VVIS provide and process both scalars and `vvm::vectors`. Accessors and functors that support both scalars and `vvm::vectors` are required, because it is inefficient to use the VPU in all situations. The scalar version is used to process edges and in situations where the storage format is not suitable for vector processing, namely when processing unknown storages.

VVIS has different algorithms and iterators for different storages. Different algorithms and iterators are required for different storages, because different storages can be processed differently. Contiguous and unknown storages are processed with a VPU and a scalar processor respectively. Unknown storages provide access to only scalars, contiguous storages provide access to both scalars and `vvm::vectors`, and Illife storages provide access to other storages.

The contiguous storage iterator was introduced in this thesis to enable extraction of both `vvm::vectors` and scalars. Like VIGRA's iterator, the contiguous storage iterator has two orthogonal components. However, while VIGRA's iterators are two-dimensonal, the contiguous storage iterator is one-dimensional. It allows traversal by `vvm::vector`, scalar or pixel steps. Each component is changed independently by `vvm::vector` and scalar steps, while pixel steps can change both. Having two orthogonal components re-

moves the need to pre-calculate the position of the right edge, because once the `vvm::vector` component is exhausted, the remaining portion is the right edge. Keeping two orthogonal components however increases the cost of constructing the iterator, since the `vvm::vector` and scalar components have to be calculated during construction. If the two components were not orthogonal, the algorithm would have had to calculate the position of the right edge, and `!=` operator's requirements might be increased, because the `!=` operator might have to calculate the real position first.

A chunky storage format suitable for vector programs was also proposed. This chunky format basically interleaves a vector at a time instead of a scalar. The advantages of such a storage format is that all components of a pixel are closer to each other in memory, only a single pointer is needed when traversing through this image, and that only one prefetch channel is needed.

VVIS uses typelists to decide the types of channels in an image. This makes it easy to use different types for different channels in images. In addition, it makes it easy to create multi-channel images. Only a single real image implementation is needed.

Chapter 10

Vectorisability of Machine-Vision Algorithms

In Chapter 8, three categories of image-processing operations were identified for use with a generic, vectorised, machine-vision library: quantitative, transformative and convolutive. For each of these categories, the general algorithm is developed and evaluated against hand-coded implementations and VIGRA. Some common operations that fall in each of these categories are then discussed, and timed.

10.1 Quantitative operations

Quantitative operations (see Section 8.2) have one input element per input set, zero or more output elements per output set and a single output set. Because quantitative algorithms can produce an unknown number of output elements per input pixel, quantitative functors are responsible for their outputs.

10.1.1 Algorithms

The quantitative algorithm accepts a single storage, an accessor and a functor.

```
namespace vvm {
  template<
    typename storageInT,
    typename accessorT,
    typename functorT
  > void for_each(const storageInT& in, accessorT a,
                  functorT& f);
}
```

Algorithm 10.1 Quantitative generic scalar algorithm

```
template<typename inIteratorT, typename functorT>
void for_each(inIteratorT begin, inIteratorT end, functorT f) {
  for(; begin != end; ++begin) {
    f(*begin);
  }
}
```

The `for_each` algorithm requires the functor to be passed by reference instead of by value, because quantitative functors keep their outputs. Without the reference requirement, the program would compile and run, but would be unable to return the output, since the answer had been calculated in a copy and thrown away after the algorithm returned. VIGRA's `inspectImage` algorithms, which are equivalent to VVIS's `for_each`, also accept a reference to the functor. Unlike VIGRA, which provides `inspectImage` algorithms that accept one or two input sets, VVIS only provides a quantitative algorithm for a single input, because VVIS does not have any functors that require more than one input set. The algorithm can easily be extended to support more than one input set.

The generic scalar algorithm is discussed first, followed by a non-generic AltiVec algorithm. The generic VVIS algorithm, which is the result of both of these algorithms, is described last. After discussing how the algorithm can be implemented, execution times are presented and discussed.

Generic scalar algorithm

Generic scalar algorithms for quantitative operations already exist and are easy to implement. Examples of quantitative algorithms in VIGRA are `vigra::inspectImage`, `vigra::inspectImageIf`, `vigra::inspectTwoImages`, and `vigra::inspectTwoImagesIf` (Köthe 2001). An example of a quantitative algorithm in the STL is `std::for_each`. In all these existing generic algorithms, the functor is responsible for handling output. Algorithm 10.1 shows how a generic scalar quantitative algorithm might be implemented.

AltiVec algorithm

The quantitative AltiVec algorithm is similar to the transformative algorithm presented in Section 3.7. This is expected as transformative operations are a subset of quantitative operations. Quantitative algorithms however are easier to implement since algorithms do not have to handle output. Quantitative algorithms only coordinate the loading of input. Algorithm 10.2 shows a simple AltiVec implementation that is able to handle any contiguous block. The left and right edges are processed using the scalar processor. Since there is only one input image, there is no need to deal with misalignment between images.

The AltiVec implementation would need to handle misalignment between images if there was more than one input image.

Generic VVIS algorithm

The generic VVIS quantitative algorithms for unknown and contiguous storages are presented in Algorithms 10.3 and 10.4. There is also a generic VVIS quantitative algorithm for Illife storages, which simply calls `vvis::for_each` for each storage contained within. Algorithms for Illife storages were discussed in Section 9.3.6. As discussed in Section 9.3.6, these algorithms are selected using function enablers. The `priv::fastest_storage` template metafunction will return `priv::using_contiguous_storage` or `priv::using_illife_storage` if all the storages are contiguous or Illife storages respectively. Otherwise, `priv::fastest_storage` returns `priv::using_unknown_storage`. Since the generic VVIS quantitative algorithms should be able to process storages of any shape, the enabler logic does not check the shape of the storages.

The algorithm for handling unknown storages is essentially the same as the generic scalar algorithm presented in Algorithm 10.1. It is shown in Algorithm 10.3.

The generic VVIS algorithm for contiguous storages is simple, because storages in VVIS have no left edges. Thus there is no need to handle misalignment between storages, even if the algorithm accepted more than one storage. Algorithm 10.4 shows the generic `vvis::for_each` algorithm for handling contiguous storages. Note that the algorithm for contiguous storages performs prefetching.

Results

Figure 10.1 shows the execution times for different `for_each` implementations. Figures 10.1(a) and 10.1(b) show how these implementations fare when processing a single-channel image, and a RGB image respectively. In both cases, the channels were 8-bit unsigned integer (`unsigned char`) types. All implementations were timed with do-nothing functors. The timings include the costs of obtaining iterators to the region of interest and of creating the do-nothing functor. All implementations were compiled using Apple GCC 3.1 30031003 with the `-Os` (optimise for size) switch.

The implementations investigated were:

VIGRA: This implementation uses VIGRA's `inspectImage` algorithm. VIGRA by default uses a contiguous chunky scalar storage format. For single-channel images, contiguous chunky storages are equivalent to contiguous planar storages. When processing RGB images, VIGRA represents a pixel using the `vigra::RGBColor` struct.

Scalar: This is a hand-coded scalar loop that processes contiguous planar storages.

Algorithm 10.2 Quantitative AltiVec algorithm

```
template<typename T, typename V, typename F>
void for_each(T* start, T* end, F f) {
  typedef __vector unsigned char vec_uc;
  const int step = sizeof(V)/sizeof(T);
  V ov, iv, iv_xtr;
  vec_uc ifix;
  int count = end - start;
  T* pi = start;
#ifdef DST
  vec_dst(pi, 0x10010100, 0);
#endif
  int roffset = ((unsigned long)pi & 0xf);
  // Do Initial load
  iv_xtr = vec_ld(0, pi);
  pi += step;
  ifix = vec_lvsl(roffset, pi);
  // Do front with scalar processor
  for(int i = 0; i < roffset; ++i)
    f(start[i]);
  // Do middle with vector processor
  for(int i = roffset; i < count; i += step) {
#ifdef DST
    vec_dst(pi, 0x10010100, 0);
#endif
    // Load unaligned
    iv = iv_xtr;
    iv_xtr = vec_ld(0, pi);
    pi += step;
    iv = vec_perm(iv, iv_xtr, ifix);
    // Do operation
    f(iv);
  }
  // Do end with scalar processor
  int starti = (count / step) *  step;
  for(int i = starti; i < count; ++i)
    f(start[i]);
}
```

Algorithm 10.3 Quantitative generic VVIS algorithm for unknown storages

```
template<
  typename storageInT,
  typename accessorInT,
  typename functorT
> typename ct::enable_if<
  boost::is_same<
    typename priv::fastest_storage<
      CT_TYPELIST1(storageInT)>::type,
    priv::using_unknown_storage
  >::value
>::type for_each(storageInT in, accessorT a, functorT& f) {
  typename storageInT::const_iterator pi = in.begin();
  typename storageInT::const_iterator end = in.end();
  for(; pi != end; ++pi) {
    f(a.get_scalar(pi));
  }
}
```

Algorithm 10.4 Quantitative generic VVIS algorithm for contiguous storages

```
template<
  typename storageInT,
  typename accessorInT,
  typename functorT
> typename ct::enable_if<
  boost::is_same<
    typename priv::fastest_storage<
      CT_TYPELIST1(storageInT)>::type,
    priv::using_contiguous_storage
  >::value
>::type for_each(storageInT in, accessorT a, functorT& f) {
  // Calculate aligned end position
  typename storageInT::const_iterator pi = in.begin();
  typename storageInT::const_iterator end = in.end();
  // Process front using the vector processor
  for(; pi.vector != end.vector; ++pi.vector) {
    ia.template prefetch_read<0>(pi);
    // Perform operation
    f(ia.get_vector(pi));
  }
  // Process rest using the scalar processor
  for(; pi.scalar != end.scalar; ++pi.scalar) {
    f(ia.get_scalar(pi));
  }
}
```

When processing single-channel images, the function accepts two pointers, one pointing to the beginning and one pointing to the first past the end element, and a functor. The single-channel function is shown below:

```
typedef unsigned char T;
template<typename F>
void for_each(const T* begin, const T* end, F f) {
  for(; begin != end; ++begin) {
    f(*begin);
  }
}
```

When processing RGB images, the function accepts four pointers instead of two, and a functor. Three of these pointers point to the beginning of each channel. The last pointer points to the first past the end element of the first channel. The RGB function is shown below:

```
typedef unsigned char T;
template<typename F>
void for_each(const T* begin0, const T* begin1,
              const T* begin2, const T* end0, F f) {
  for(; begin0 != end0; ++begin0, ++begin1, ++begin2) {
    f(*begin0, *begin1, *begin2);
  }
}
```

Scalar Mode, VVIS (Specialised Iterators): This version was compiled without any VPU support. This version uses a specialised iterator for single-channel images instead of using the general iterator that also supports multi-channel images. Prefetching was not enabled in this version.

Scalar Mode, VVIS (Automatic Unknown): This version was compiled without any VPU support. In the absence of a VPU, this version treats contiguous storages as unknown storages. This optimisation has no effect when VVM runs in AltiVec mode. Prefetching was not enabled in this version.

In scalar mode, we can process the entire storage as scalars instead of as `vvm::vectors` and scalars. This is because in scalar mode, there is only one scalar in a `vvm::vector` and therefore trying to treat it as a `vvm::vector` just adds unnecessary overheads. To implement this tweak, the preprocessor could be used to remove the `vvm::vector` portion of the contiguous `for_each` algorithm. This method however fails for chunky images because the algorithm should be taking

pixel steps and not scalar steps. Instead of modifying the algorithm, we can treat contiguous storages as unknown storages by simply changing `vvis::uses_contiguous_storage` to always return `false`. Treating contiguous storages as unknown storages in scalar mode is not only easy to implement, more importantly, it does not lead to an incorrect solution. VVIS detects that there is no VPU available by querying the `VVM_VECTOR_PROCESSOR` macro defined by VVM.

Scalar Mode, VVIS (Automatic Unknown + Prefetch): This version was compiled without VPU support. This version is basically the Scalar Mode, VVIS (Automatic Unknown) version but with prefetching enabled. In scalar mode, this version is actually equivalent to Scalar Mode, VVIS (Automatic Unknown) because in scalar mode, the unknown algorithm, which never prefetchs, is used.

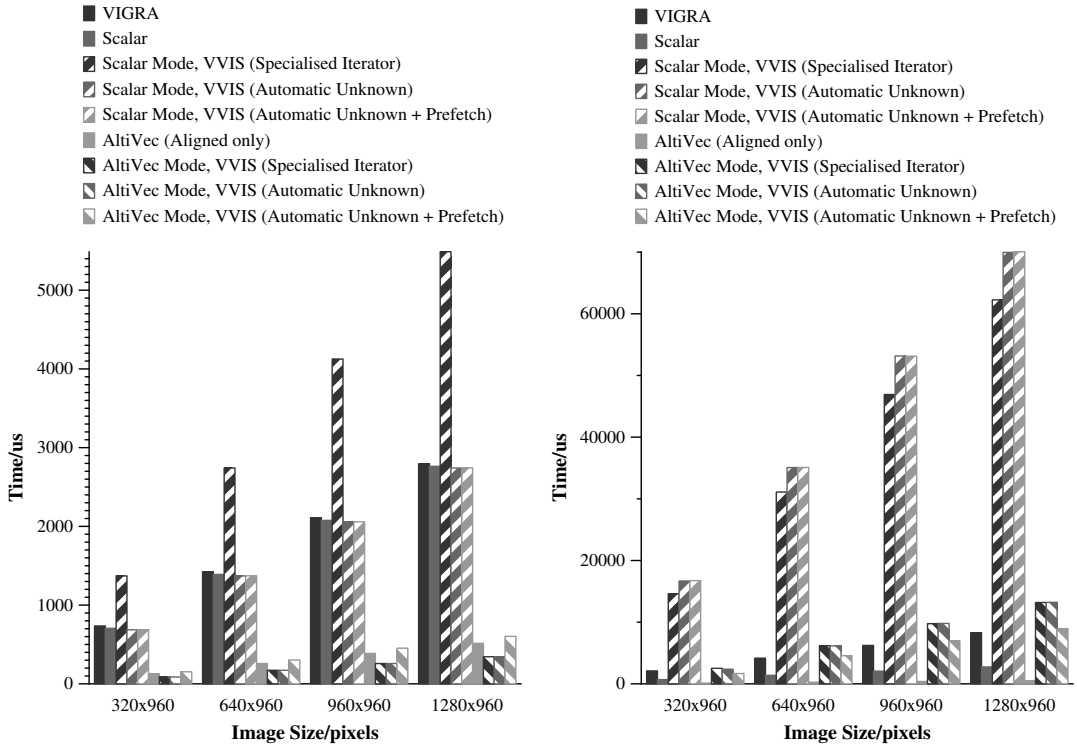
AltiVec (Aligned only): This version is an AltiVec algorithm that does not have any edge handling capabilities. This version actually solves a simpler problem than the other versions evaluated. It is only applicable in AltiVec mode. Prefetching was not enabled. The single-channel version is shown below:

```
typedef unsigned char T;
typedef __vector unsigned char V;
template<typename F>
  void for_each(const T* begin, const T* end, F f) {
    const int step = sizeof(V)/sizeof(T);
    for(; begin != end; begin += step) {
      f(vec_ld(0, begin));
    }
  }
```

When processing RGB images, the AltiVec function accepts a contiguous storage for each channel, and passes the value of each channel as a separate parameter. The RGB version is shown below:

```
typedef unsigned char T;
typedef __vector unsigned char V;
template<typename F>
  void for_each_rgb(const T* begin0, const T* begin1,
                    const T* begin2, const T* end0, F f) {
    const int step = sizeof(V)/sizeof(T);
    for(; begin0 != end0;
        begin0 += step, begin1 += step, begin2 += step) {
      f(vec_ld(0, begin0),
```

Figure 10.1 Performance different `vvis::for_each` implementations



(a) Single-Channel Image

(b) RGB Image

```
            vec_ld(0, begin1),
            vec_ld(0, begin2));
        }
    }
```

AltiVec Mode, VVIS (Specialised Iterators): This version is the same as Scalar Mode, VVIS (Specialised Iterators) but compiled with AltiVec support.

AltiVec Mode, VVIS (Automatic Unknown): This version is the same as Scalar Mode, VVIS (Automatic Unknown) but compiled with AltiVec support.

AltiVec Mode, VVIS (Automatic Unknown + Prefetch): This version is the same as "Scalar Mode, VVIS (Automatic Unknown + Prefetch)" but compiled with AltiVec support.

Figure 10.1 shows that when processing single-channel images in scalar mode, VVIS with a specialised iterator for single-channel images was significantly slower than scalar hand-coded and VIGRA versions, because the VVIS algorithm selected was vectorised, and thus performs more work than necessary. In AltiVec mode however, a specialised iterator allows the VVIS implementation to run on par with the AltiVec hand-coded version.

Table 10.1 Quantitative algorithms' required functor interface

Expression	Return Type	Notes
`f(s)`	`void`	`s` is a scalar
`f(v)`	`void`	`v` is a vector

Prefetching had no effect in scalar mode as expected because the VVIS algorithms do not prefetch when processing unknown storages. Prefetching in AltiVec mode produced slower programs when processing single-channel images, but was faster when processing RGB images. This suggests that there was so little work to be done when processing the single-channel images that the VPU outruns the prefetching process.

When processing RGB images, VVIS was always significantly slower. Specialised iterators, automatic unknown storages, and prefetching did not make any difference. This was probably because `ct::tuple` was used, and `ct::tuple` is probably more difficult for Apple GCC 3.1 20021003 to optimise.

10.1.2 Functors

The VVIS quantitative algorithm requires functors to implement the interface detailed in Table 10.1. How the output is returned to the user is not defined. The output requirements were not defined because the VVIS quantitative algorithm does not handle output and because the output is unknown.

10.2 Transformative operations

As discussed in Section 8.2, transformative operations are a subset of quantitative operations, and thus with the right functor, can also use the quantitative algorithm. However, to reduce the responsibilities of the functor, and to be consistent with other generic libraries, transformative operations have their own algorithms in VVIS. Since transformative operations are a subset of quantitative operations, as expected, the transformative algorithm is similar to the quantitative algorithm. Transformative operations produce a single output element from a single input element per input set, and have one output set. In VVIS, transformative algorithms are provided for one or two input sets.

10.2.1 Algorithms

The transformative algorithm in VVIS is the `transform` algorithm, which has the following definitions.

```
namespace vvm {
```

```
  template<
    typename storageInT,
    typename accessorInT
    typename storageOutT,
    typename accessorOutT,
    typename functorT
  > void transform(const storageInT& in, accessorInT ia,
                   storageT& out, accessorOutT oa, functorT f);
template<
    typename storageIn1T,
    typename accessorIn1T
    typename storageIn2T,
    typename accessorIn2T
    typename storageOutT,
    typename accessorOutT,
    typename functorT
  > void transform(const storageIn1T& in1, accessorIn1T ia1,
                   const storageIn2T& in2, accessorIn2T ia2,
                   storageT& out, accessorOutT oa, functorT f);
}
```

Like VIGRA, each input and output set has its own accessor. This allows for more flexibility. Unlike the quantitative algorithm, the transformative algorithm uses a copy of the functor. This allows unnamed functors and functions to be used. While `const functorT&` accepts unnamed `const` functors, it does not accept functions. For example, if `transform` accepted a reference like `for_each`, `vvis::transform(... vvis::plus<>())` would not be valid. Using a copy however makes it impossible to return values through the functor.

Only the one input set version is discussed here. The two input set version is simply longer and does not pose any new problems.

Generic scalar algorithm

The generic scalar transformative algorithm is easy to implement, with suitable generic versions already existing in VIGRA (Köthe 2001) and the Standard C++ Library (Inf 1998). VIGRA provides a `vigra::transformImage` and `vigra::transformImageIf` function which uses a 2-D iterator to walk through the image. The `std::transform` function provided in the Standard C++ Library can also be used if the image is presented as a 1-D iterator.

Algorithm 10.5 is an example of how a generic scalar transformative algorithm would

Algorithm 10.5 Transformative generic scalar algorithm

```
template<typename inT, typename outT, typename F>
void transform(inT begin, inT end, outT out, F f) {
  for(; begin != end; ++begin, ++out)
    *out = f(*begin);
}
```

be implemented. A one-dimensional loop was used because transformative operations do not require spatial iterators.

Note that `std::transform`, `vigra::transformImage`, and Algorithm 10.5 are capable of using any memory location; they are capable of handling unaligned loads and stores. In addition, all three algorithms fail if the input region overlaps with the output region, because the algorithms write the output to the correct position immediately. If this output position located in input that is yet to be processed, the output will be erroneous since the input has changed.

AltiVec algorithm

AltiVec transformative functions were discussed in Section 3.7. Section 3.7 also investigated the effects of alignment, prefetching and function complexity on the performance of AltiVec transformative functions.

Generic VVIS algorithm

The generic VVIS transformative algorithms for unknown and contiguous storages are presented in Algorithms 10.6 and 10.7. There is also a generic VVIS transformative algorithm for Illife storages, which simply calls `vvis::transform` for each storage contained within. Algorithms for Illife storages were discussed in Section 9.3.6. Like the generic VVIS quantitative algorithms presented in Section 10.1.1, transformative algorithms use the same enabler logic and can process storages of any shape.

The generic VVIS transformative algorithm for unknown storages is straightforward, and is similar to the scalar algorithm presented earlier. The unknown storage version does not use the VPU and is selected if any of the input or output storages are unknown storages. The generic VVIS transformative algorithm for unknown storages is presented in Algorithm 10.6.

In VVIS, since storages do not have left edges, all storages have the same alignment. Because of this, the VVIS generic algorithm for transformative operations on contiguous storages is straightforward. It is shown in Algorithm 10.7. This algorithm is only used when all input and output storages are contiguous storages and performs prefetching.

Algorithm 10.6 Transformative generic VVIS algorithm for unknown storages

```
template<
  typename storageInT,
  typename accessorInT,
  typename storageOutT,
  typename accessorOutT,
  typename functorT
> typename ct::enable_if<
  boost::is_same<
    typename priv::fastest_storage<
      CT_TYPELIST2(storageInT, storageOutT)>::type,
    priv::using_unknown_storage
  >::value
>::type transform(const storageInT& in, accessorInT,
storageOutT& out, accessorOutT a, functorT f) {
  typename storageInT::const_iterator pi_rend = in.end();
  typename storageInT::const_iterator pi = in.begin();
  typename storageOutT::iterator po = out.begin();

  for(; pi != pi_rend; ++pi, ++po) {
    a.set_scalar(f(a.get_scalar(pi)), po);
  }
}
```

Results

Figures 10.2, 10.3 and 10.4 show how different implementations of the transformative algorithm perform. All implementations were compiled using Apple GCC 3.1 20021003, with the `-Os` (optimise for size) switch. The functor used with the implementations, simply returns the value passed to it without any modifications.

The implementations that were evaluated are as follows:

VIGRA: This implementation uses VIGRA's `transformImage` algorithm. VIGRA uses a contiguous chunky scalar storage format. When processing RGB images, VIGRA uses `vigra::RGBColor` to represent a pixel.

Scalar: This version is a hand-coded scalar loop that processes contiguous planar storages.

Scalar Mode, VVIS (No Prefetch): This version uses VVIS's `transform` algorithm without prefetching and without a VPU. VVIS by default automatically treats contiguous storages as unknown storages when there is no VPU present. Thus, the VVIS `transform` algorithm used in this version operated on unknown storages. In addition, since the storages were unknown storages, VVIS used `T*` and `ct::tuple<-`

Algorithm 10.7 Transformative generic VVIS algorithm for contiguous storages

```
template<
  typename storageInT,
  typename accessorInT,
  typename storageOutT,
  typename accessorOutT,
  typename functorT
> typename ct::enable_if<
  boost::is_same<
    typename priv::fastest_storage<
      CT_TYPELIST2(storageInT, storageOutT)>::type,
    priv::using_contiguous_storage
  >::value
>::type transform(const storageInT& in, accessorInT,
storageOutT& out, accessorOutT a, functorT f) {
  typename storageInT::const_iterator end = in.end();
  // Process front using the vector processor
  typename storageInT::const_iterator pi = in.begin();
  typename storageOutT::iterator po = out.begin();
  for(; pi.vector != end.vector; ++pi.vector, ++po.vector) {
    ia.template prefetch_read<0>(pi);
    oa.template prefetch_write<
      accessorInT::prefetch_channel_count>(po);
    a.set_vector(f(a.get_vector(pi)), po);
  }
  // Process rest using the scalar processor
  for(; pi.scalar != end.scalar; ++pi.scalar, ++po.scalar) {
    a.set_scalar(f(a.get_scalar(pi)), po);
  }
}
```

`CT_TYPELIST3(T*, T*, T*)>` as iterators for single-channel and RGB images respectively, where `T` is `unsigned char`.

Scalar Mode, VVIS (Prefetch): This version is the same as Scalar Mode, VVIS (No Prefetch) except prefetching is enabled. Because VVIS automatically treats contiguous storages as unknown storages when there is no VPU available and because VVIS algorithms operating on unknown storages never prefetch, this version should run at the same speed as Scalar Mode, VVIS (No Prefetch).

AltiVec (Aligned only): This version uses an AltiVec transform function that only performs aligned loads and aligned stores. It is equivalent to Aligned Load and Aligned Store from Section 3.7.

The single-channel version is shown below.

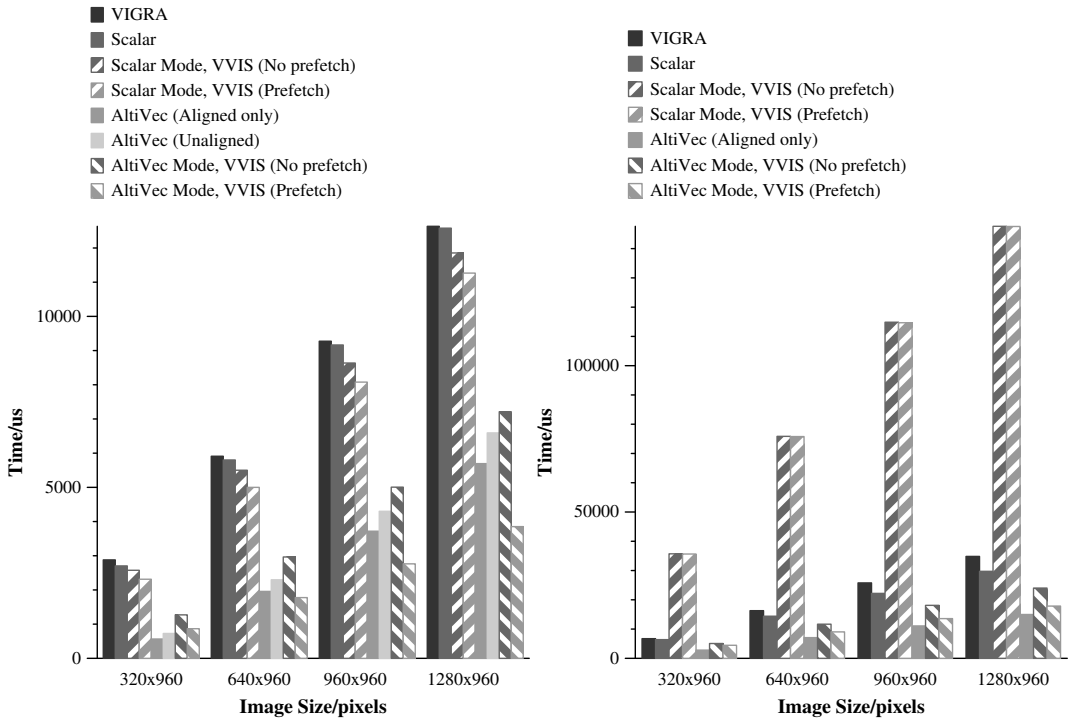
```
typename unsigned char type;
typename __vector unsigned char vector_type;
template<class F>
void transform(const type* begin, const type* end,
type* out, F f) {
  const int step = sizeof(vector_type)/sizeof(type);
  vector_type ov, iv;

  for(; begin != end; begin += step, out += step) {
    iv = vec_ld(0, begin);
    ov = f(iv);
    vec_st(ov, 0, out);
  }
}
```

AltiVec (Unaligned): This version uses an AltiVec transform function that is equivalent to Unaligned Load and Unaligned Store (Optimised version) from Figure 3.6. This version was only timed for single-channel images.

AltiVec Mode, VVIS (No Prefetch): This version uses VVIS's `transform` algorithm without prefetching but with AltiVec active. In this case, VVIS does not treat contiguous storages as unknown storages because the VPU is available. The `transform` algorithm in this version is therefore the contiguous version. When operating on single-channel images, VVIS uses a specialised iterator.

AltiVec Mode, VVIS (Prefetch): This version is the same as AltiVec Mode, VVIS (No Prefetch) except prefetching is enabled. Because the VVIS `transform` algorithm

Figure 10.2 Performance of different `vvis::transform` implementations, when the source image is the destination image, and the functor returned input values unchanged



(a) Single-Channel Image

(b) RGB Image

does not actually detect when the source and destination storages are equal, the algorithm prefetchs the same memory block twice using different channels. Since the storage is a planar storage, three prefetch channels were required for prefetching each storage. Since AltiVec only has four prefetch channels, all three channels of the sources storage would be prefetched, but only one of the destination would be. Chunky storage formats would have required only one prefetch channel per storage.

Figure 10.2 shows the performance of different transformative algorithm implementation when the source and destination storages are the same. The net effect is that the transform algorithm copies the input image to itself.

Figure 10.2(a) shows that when processing single-channel images, VVIS appears to perform adequately. In scalar mode, it runs on par with or slightly faster than the hand-coded scalar or VIGRA versions. When AltiVec is available, it is slower than AltiVec (Aligned) which is expected. It is also slower than AltiVec (Unaligned) when prefetching is off, which is consistent with results from Section 3.7, despite the VVIS algorithm only handling unalignment on one edge, while AltiVec (Unaligned) handles unalignment on two edges. From Section 3.7, AltiVec (Unaligned) will be slower than AltiVec (Aligned) when prefetching is on. VVIS was slower than AltiVec (Aligned) because of the extra

cost involved in creating the iterator. VVIS has to calculate the `vvm::vector` and scalar components of the iterator. When prefetching is on, VVIS was either close to or faster than AltiVec versions. The margin increased as the image size increased which is expected. Prefetching provides more benefits when there is more data to process. VVIS actually run faster than Scalar or VIGRA in scalar mode, which was unexpected. This was probably because of the optimiser. Scalar Mode, (No Prefetch) was slower than Scalar Mode, (Prefetch) even though no prefetching was done probably because the prefetch version was timed after the non-prefetching version in the same program and thus benefited from data already in the cache.

Figure 10.2(b) shows that when processing RGB images, VVIS's performance in AltiVec mode was comparable to AltiVec hand-coded programs, but was dismal in scalar mode. In scalar mode the iterator was a tuple of pointers while in AltiVec mode, the iterator was an object which contained a tuple of `vvm::vector` pointers and an integer. In RGB mode, prefetching produced faster results but was unable to overtake AltiVec (Aligned).

In Figure 10.3, all the implementations were evaluated using a functor that simply returned the same value that it was given. Since in Figure 10.3, the source and destination images are different images, the net effect is that the transform algorithm copies one image to the other.

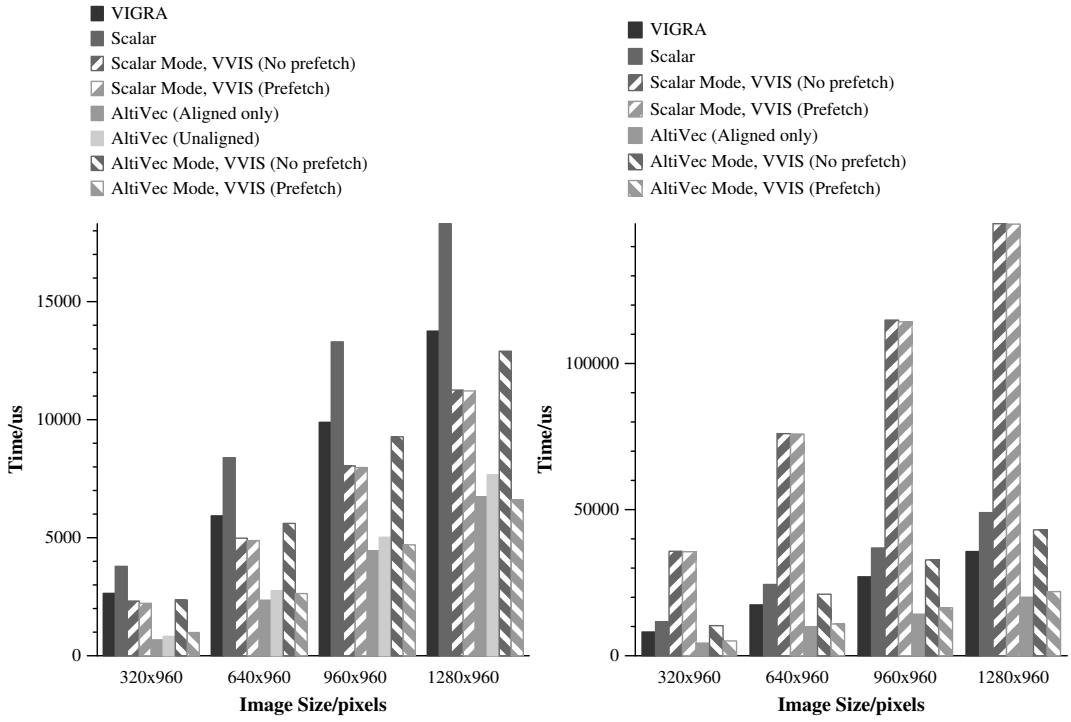
Figure 10.3 displays generally the same behaviour as Figure 10.2. The main difference is that Scalar was noticeably faster than VIGRA. In addition, Scalar Mode, VVIS (No Prefetch) run at the same speed as Scalar Mode, VVIS (Prefetch) as expected. The AltiVec Mode, VVIS (No Prefetch) was much slower in this version. It was even slower than Scalar Mode, VVIS (No Prefetch) when handling single-channel images, which is a bit disappointing. As shown in Figure 10.2(b), when processing RGB images, VVIS in scalar mode was much slower while it seems to be alright in AltiVec mode.

In Figure 10.4, all implementations were evaluated against a functor that added each value to itself. The net effect is that the algorithm doubled the value of pixels in the source image. While the value on overflow is undefined, this is not important because the purpose was to give the vector processor something extra to do. Lai et al. (2002) used the same operation to evaluate its transform algorithms.

Figure 10.4 also displays generally the same behaviour as Figure 10.2. The AltiVec (Unaligned) version was faster AltiVec (Aligned) which was unexpected. However the speedup ratio was lower when the source and destination images were different.

Drawing conclusions from Figures 10.2, 10.3 and 10.4 is a bit more difficult because of the presence of the optimiser. The optimiser was included because VVIS and VVM depend on the optimiser to remove extra functions calls. Generally though, they show that VVIS provides close to hand-coded performance in AltiVec mode when processing single-channel images. When processing single-channel images, in scalar mode, VVIS

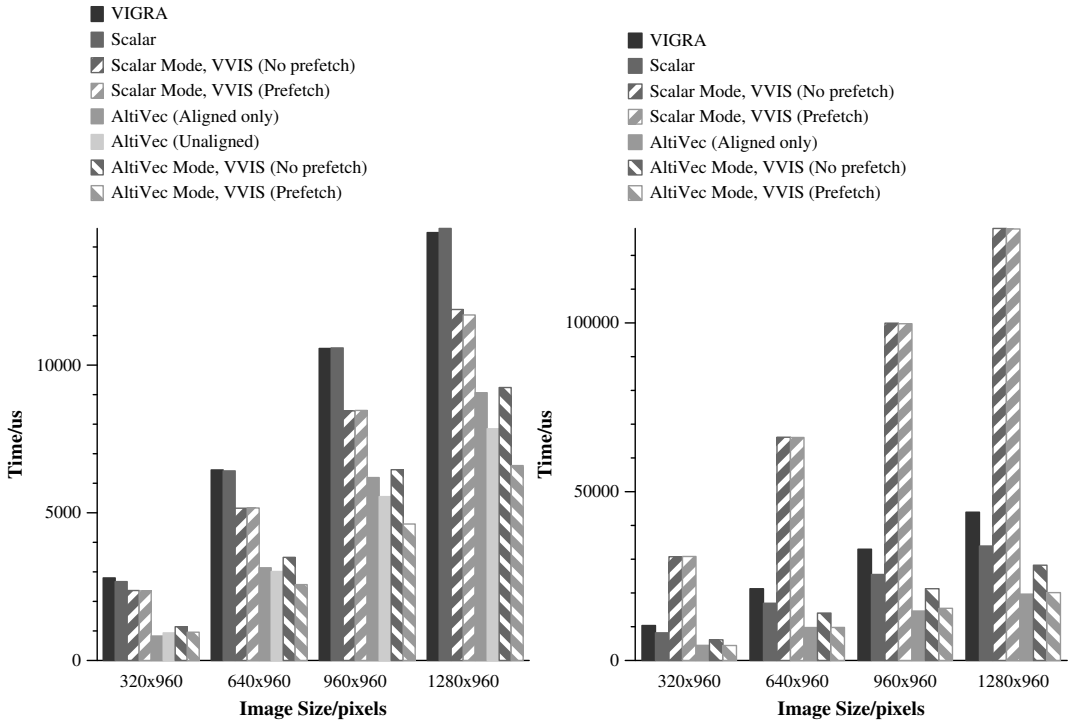
Figure 10.3 Performance of different `vvis::transform` implementations when the source image and the destination images are different, and the functor returned input values unchanged



(a) Single-Channel Image

(b) RGB Image

Figure 10.4 Performance of different `vvis::transform` implementations when the source and destination images are different, and the functor returns 2×input value



(a) Single-Channel Image

(b) RGB Image

Table 10.2 Transformative algorithms' required functor interface

Expression	Return Type	Notes
`f(s)`	Output	`s` is a scalar
`f(v)`	Output	`v` is a vector

was actually faster than the hand-coded version. When processing RGB images, it provides close to hand-coded performance only in AltiVec mode. In addition, they show that AltiVec generally provides only about a one-fold speed even with `unsigned char` images when optimisation is on for simple transformative algorithms. These figures suggest that the treatment of RGB images in scalar mode needs more adjustment.

10.2.2 Functors

Functors for transformative algorithms should implement the interface shown in Table 10.2. As usual functors should provide both a scalar and `vvm::vector` version. For transformative algorithms that have two input sets, there would be two arguments to both functions.

Arithmetic, logical, equalise and threshold operations were investigated in more detail. For each of these operations, the implementation of the functor is discussed, followed by timing results. In both scalar and AltiVec modes, the storage policy used was Illife planar and prefetching was on. In addition, in scalar mode, contiguous storages are treated as unknown storages. Timing results were collected for `unsigned char`, `unsigned short int`, `unsigned int`, `float` and `double` images. Images consisting of `long`s were not timed because in the compiler used, Apple GCC 3.1 20021003, `long` is the same size as a `int`. All programs were compiled with the `-Os` (optimise for size) switch, which is required for the VVM implementation used to run at zero-cost when operating on `char`s.

The timing results show the ratio of VVIS running in AltiVec mode compared to scalar mode. A value of one indicates that VVIS runs at the same speed in AltiVec mode as in scalar mode. A value greater than one indicates that VVIS runs faster in AltiVec mode while a value less than one indicates that it runs slower in AltiVec mode. Since VVIS is comparable to scalar hand-coded programs in scalar mode, the ratio obtained is also a rough indication of the speedup factor of VVIS in AltiVec mode over hand-coded scalar programs.

Arithmetic

The `plus` functor's implementation that is discussed in this section is representative of arithmetic operations. The `plus` functor is shown in Algorithm 10.8 and 10.9. Algorithm 10.8 works for both single and multi-channel images. This is because tuples also have

Algorithm 10.8 `plus` functor (single-channel and multi-channel)

```
template<typename T = void> struct plus
: public std::binary_function<T, T, T> {
  T operator()(const T& a, const T& b) const {
    return a + b;
  }
  typename vvm::add_vector<T>::type operator()(
  const typename vvm::add_vector<T>::type& a,
  const typename vvm::add_vector<T>::type& b) const {
    return a + b;
  }
};
```

Algorithm 10.9 `plus` functor (autodetect)

```
template<> struct plus<void> {
  template<typename T>
  T operator()(const T& a, const T& b) const {
    return a + b;
  }
};
```

the standard operators defined. `T` is the pixel type. While the implementation of `plus` is generally straightforward, note the use of `vvm::add_vector<T>::type` to deduce the appropriate `vvm::vector` type instead of `vvm::vector<T>`. As discussed in Section 9.3.7, `vvm::add_vector<T>::type` is more appropriate because the scalar type `T` might be `const` or a `ct::tuple`. VVIS also provides a `plus` functor that is able to autodetect the pixel type, in the manner presented by Meyers (2002).

You can use the `plus` functor in the following manner:

```
typedef vvis::image<vvis::uint8> image_t;
image_t source1, source2, destination;
vvis::transform(source1, vvis::pixel_accessor<image_t>(),
                source2, vvis::pixel_accessor<image_t>(),
                destination, vvis::pixel_accessor<image_t>(),
                plus<image_t::pixel_type>());
```

You can also use `plus<>()`, which is the autodetect version, instead. The non-autodetect version is required for binding. All the arithmetic functors, listed below, can be implemented in the same manner as `plus`.

```
plus              minus             multiplies
divides           modulus           negate
```

Algorithm 10.10 `plus_saturated` functor (single-channel)

```
template<typename T = void> struct plus_saturated
: public std::binary_function<T, T, T> {
  T operator()(const T& a, const T& b) const {
    typename ct::promote<T>::type r = a + b;
    return r < std::numeric_limits<T>::max() ?
      r : std::numeric_limits<T>::max();
  }
  typename vvm::add_vector<T>::type operator()(
  const typename vvm::add_vector<T>::type& a,
  const typename vvm::add_vector<T>::type& b) const {
    return vvm::adds(a, b);
  }
};
```

The `plus_saturated` and `minus_saturated` functors are a bit different because VVM provides saturated addition and subtraction functions for `vvm::vectors` but not for scalars. The scalar version is coded using additions, promotions and `if`s. The `plus_saturated` functor is shown in Algorithms 10.10, 10.11 and 10.12. Unlike `plus` which could use the same version for both single and multi-channel images, `plus_saturated` requires different versions for single and multi-channel images because `vvm::adds` is not defined for tuples. The tuple version is actually similar to the autodetect version. The tuple implementation basically applies the functor object's `operator()` to each element in the tuple. This can be performed using template metaprogramming. It was omitted from Algorithm 10.12 for brevity.

Once again, note the use of `vvm::add_vector` to get the `vvm::vector` type. The multi-channel and autodetect versions does not use `vvm::add_vector` to help the compiler select the most specialised version. In fact, if `vvm::add_vector` was used, Apple GCC 3.1 20021003 selects the general scalar version. The scalar version uses `ct::promote` to keep the result of `a + b` in a larger type. If `T` was used, then the result will never be larger than `T`'s largest value. If the result is never larger, then we will not detect that an overflow had occurred.

For arithmetic operations that have a one-to-one relationship with an AltiVec instruction, namely addition and subtraction, the potential speedup is shown in Tables 10.3. The `plus_saturated` functor is also profiled because `plus_saturated` has a VPU instruction while the scalar code is coded using additions, promotions and `if`s. As expected, `plus_saturated`'s speedup was greater than `plus`'s.

Algorithm 10.11 `plus_saturated` functor (multi-channel)

```
template<typename TL> struct plus_saturated<ct::tuple<TL> >
: public std::binary_function<ct::tuple<TL>, ct::tuple<TL>,
ct::tuple<TL> > {
  ct::tuple<TL> operator()(
  const ct::tuple<TL>& a, const ct::tuple<TL>& b) const {
    ct::tuple<TL> ret;
    meta::EFOR4<0, meta::Less, ct::length<TL>::value, +1,
      priv::function_call>::exec(*this, ret, a, b);
    return ret;
  }
  typename vvm::add_vector<ct::tuple<TL> >::type operator()(
  const typename vvm::add_vector<ct::tuple<TL> >::type& a,
  const typename vvm::add_vector<ct::tuple<TL> >::type& b) const {
    typename vvm::add_vector<ct::tuple<TL> >::type ret;
    meta::EFOR4<0, meta::Less, ct::length<TL>::value, +1,
      priv::function_call>::exec(*this, ret, a, b);
    return ret;
  }
public:
  // Should be private, but template friendship fails in GCC 3.1
  /// Scalar implementation
  template<typename T>
  T operator()(const T& a, const T& b) const {
    typedef typename ct::promote<T>::type promote_t;
    promote_t r = a + b;
    return r < std::numeric_limits<T>::max() ?
      static_cast<T>(r) : std::numeric_limits<T>::max();
  }
  /// Vector implementation
  template<typename T>
  typename vvm::vector<T> operator()(
  const vvm::vector<T>& a,
  const vvm::vector<T>& b) const {
    return vvm::adds(a, b);
  }
};
```

Algorithm 10.12 `plus_saturated` functor (autodetect)

```
template<> struct plus_saturated<void> {
  /// Tuple implementation
  template<typename TL>
  ct::tuple<TL> operator()(
  const ct::tuple<TL>& a,
  const ct::tuple<TL>& b) const {
    // Tuple implementation here
    // Apply operator() to each element of a
    // Not reproduced here for brevity
  }
  /// Scalar implementation
  template<typename T>
  T operator()(const T& a, const T& b) const {
    typedef typename ct::promote<T>::type promote_t;
    promote_t r = a + b;
    return r < std::numeric_limits<T>::max() ?
      static_cast<T>(r) : std::numeric_limits<T>::max();
  }
  /// Vector implementation
  template<typename T>
  typename vvm::vector<T> operator()(
  const vvm::vector<T>& a,
  const vvm::vector<T>& b) const {
    return vvm::adds(a, b);
  }
};
```

Table 10.3 Speedup attained by executing some VVIS's arithmetic functors in AltiVec mode over scalar mode

	Single-Channel Image				RGB Image			
	320 ×960	640 ×960	960 ×960	1280 ×960	320 ×960	640 ×960	960 ×960	1280 ×960
`unsigned char`	2.3	1.6	1.9	1.8	6.0	5.8	6.6	6.5
`unsigned short int`	1.2	1.3	1.4	1.3	2.6	2.9	2.9	2.8
`unsigned int`	1.2	1.2	1.3	1.3	1.3	1.3	1.9	1.3
`float`	1.1	1.2	1.2	1.2	1.4	1.3	1.5	1.4
`double`	0.9	1.0	1.0	1.0	1.2	1.3	1.5	1.2

(a) `plus` functor

	Single-Channel Image				RGB Image			
	320 ×960	640 ×960	960 ×960	1280 ×960	320 ×960	640 ×960	960 ×960	1280 ×960
`unsigned char`	3.7	2.5	2.8	2.7	7.2	7.2	8.1	7.8
`unsigned short int`	1.7	1.9	2.0	2.0	3.1	3.4	3.5	3.3
`unsigned int`	1.3	1.4	1.4	1.4	1.5	1.4	2.1	1.5
`float`	1.3	1.3	1.4	1.3	1.3	1.3	1.4	1.3
`double`	0.9	0.9	0.9	0.9	1.3	1.3	1.5	1.3

(b) `plus_saturated` functor

Logical

Most logical operators, like arithmetic operators, require two inputs and have AltiVec support. The `equal_to` functor's implementation is discussed in this section as representative of logical operators. The `equal_to` functor is shown in Algorithms 10.2.2, 10.2.2 and 10.2.2. The `equal_to` functor's implementation is similar to `plus_saturated`. The scalar versions return `VVM_TRUE` and `VVM_FALSE` for `true` and `false` respectively. This is required because `true` is converted to `1`, and not `~0` in scalar programs. See Section 6.2 for more information on `VVM_TRUE` and `VVM_FALSE`.

Algorithm 10.13: `equal_to` functor (single-channel)

```
template<typename T = void> struct equal_to
: public std::binary_function<
  T, T, typename vvm::scalar_traits<T>::bool_type
> {
  typename vvm::scalar_traits<T>::bool_type
  operator()(const T& a, const T& b) const {
    return a == b ? ~0 : 0;
  }
  typename vvm::add_vector<
```

```
    typename vvm::scalar_traits<T>::bool_type
  >::type operator()(
  const typename vvm::add_vector<T>::type& a,
  const typename vvm::add_vector<T>::type& b) const {
    return a == b;
  }
};
```

Algorithm 10.14: `equal_to` functor (multi-channel)

```
template<typename TL> struct equal_to<ct::tuple<TL> >
: public std::binary_function<ct::tuple<TL>, ct::tuple<TL>,
typename vvm::to_bool<ct::tuple<TL> >::type> {
  typename vvm::to_bool<ct::tuple<TL> >::type operator()(
  const ct::tuple<TL>& a, const ct::tuple<TL>& b) const {
    typename vvm::to_bool<ct::tuple<TL> >::type ret;
    meta::EFOR4<0, meta::Less, ct::length<TL>::value, +1,
      priv::function_call>::exec(*this, ret, a, b);
    return ret;
  }
  typename vvm::add_vector<
    typename vvm::to_bool<ct::tuple<TL> >::type
  >::type operator()(
  const typename vvm::add_vector<ct::tuple<TL> >::type& a,
  const typename vvm::add_vector<ct::tuple<TL> >::type& b) const {
    typename vvm::add_vector<
      typename vvm::to_bool<ct::tuple<TL> >::type>::type ret;
    meta::EFOR4<0, meta::Less, ct::length<TL>::value, +1,
      priv::function_call>::exec(*this, ret, a, b);
    return ret;
  }
public:
  // Should be private, but template friendship fails in GCC 3.1
  template<typename T>
  typename vvm::scalar_traits<T>::bool_type
  operator()(const T& a, const T& b) const {
    return a == b ? ~0 : 0;
  }
  template<typename T>
```

```
  vvm::vector<typename vvm::scalar_traits<T>::bool_type>
  operator()(
  const vvm::vector<T>& a, const vvm::vector<T>& b) const {
    return a == b;
  }
};
```

Algorithm 10.15: `equal_to` functor (autodetect)

```
namespace priv {
  /** Function Call
   * Implements tuple code by calling self.operator()
   */
  struct function_call {
    template<int i> struct Code {
      template<
        typename selfT,
        typename returnT,
        typename T
     > static void exec(const selfT& self, returnT& ret,
                        const T& a) {
        ret.template get<i>() =
          self.operator()(a.template get<i>());
      }
      template<typename selfT, typename returnT, typename T>
      static void exec(const selfT& self, returnT& ret,
                       const T& a, const T& b) {
        ret.template get<i>() =
          self.operator()(a.template get<i>(),
                          b.template get<i>());
      }
    };
  };
} // End of priv namespace
template<> struct equal_to<void> {
  template<typename TL>
    ct::tuple<typename ct::apply<TL, vvm::to_bool>::type>
    operator()(const ct::tuple<TL>& a,
               const ct::tuple<TL>& b) const {
```

```
        ct::tuple<typename ct::apply<TL, vvm::to_bool>::type> ret;
        meta::EFOR4<0, meta::Less, ct::length<TL>::value, +1,
          priv::function_call>::exec(*this, ret, a, b);
        return ret;
    }
    template<typename T>
    typename vvm::scalar_traits<T>::bool_type
    operator()(const T& a, const T& b) const {
      return a == b ? ~0 : 0;
    }
    template<typename T>
    vvm::vector<typename vvm::scalar_traits<T>::bool_type>
    operator()(
    const vvm::vector<T>& a, const vvm::vector<T>& b) const {
      return a == b;
    }
  };
```

Table 10.4 Speedup attained by executing VVIS's `equal_to` functor in AltiVec mode over scalar mode

	Single-Channel Image				RGB Image			
	320 ×960	640 ×960	960 ×960	1280 ×960	320 ×960	640 ×960	960 ×960	1280 ×960
`unsigned char`	1.3	1.3	1.4	1.4	6.8	6.3	7.3	6.9
`unsigned short int`	1.2	1.4	1.4	1.4	3.0	3.3	3.3	3.2
`unsigned int`	1.2	1.2	1.3	1.3	1.5	1.5	2.1	1.5
`float`	1.4	1.3	1.4	1.4	2.2	2.1	2.3	2.1
`double`	0.8	0.8	0.8	0.8	1.8	1.8	1.9	1.8

Table 10.4 shows the speedup attained by changing VVIS from scalar mode to AltiVec mode. The speedup factor is actually better than the arithmetic operations. This is probably because the logical functions in VVM returns `~0` and `0` for `true` and `false` respectively, whereas the scalar code returns `1` and `0`. The scalar code thus has to change the `1` to a `~0`.

Equalise

Equalise distributes a given grey-level interval [min, max] evenly over the full grey scale [0, 255]. *m* is the gradient of the line and *c* is the intersection.

$$L(X_0) = \begin{cases} 0 & \text{if} X_0 < \min \\ mX_0 + c & \text{if} X_0 \geq \min \text{AND} X_0 \leq \max \\ 255 & \text{if} X_0 > \max \end{cases} \tag{10.1}$$

$$m = \frac{254}{(\max - \min)} \tag{10.2}$$

$$c = 1 - m \times \min \tag{10.3}$$

The VVIS single and multi-channel versions of the equalise functor are presented in Algorithms 10.2.2 and 10.2.2 respectively. Equalise does not have an autodetect version because its constructor accepts parameters. Without specifying a type when it gets created, `equalize` would not know what type the minimum and maximum values should be. The `equalize` functor uses `vvm::madd` to perform multiplication and addition without promotion.

Algorithm 10.16: `equalize` functor (single-channel)

```
template<typename T> struct equalize {
    equalize(const T min, const T max)
    : _smin(min), _smax(max), _vmin(min), _vmax(max),
    _vzero_val(0), _vmax_val(std::numeric_limits<T>::max()) {
      // Calculate m & c
      _sm = (std::numeric_limits<T>::max() - 1) /
            (_smax - _smin);
      _sc = 1 - _sm * _smin;
      // Copy m & c to vvm::vector
      _vm = _sm;
      _vc = _sc;
    }
    T operator()(const T a) {
      if(a < _smin)
        return 0;
      else if(a > _smax)
        return std::numeric_limits<T>::max();
      else
        return _sm * a + _sc;
    }
    vvm::vector<T> operator()(const vvm::vector<T>& a) {
      vvm::vector<T> ret;
      ret = vvm::madd(_vm, a, _vc);
```

```
    ret = vvm::select(a < _vmin, ret, _vzero_val);
    ret = vvm::select(a > _vmax, ret, _vmax_val);
    return ret;
  }
private:
  T _smin, _smax, _sm, _sc;
  vvm::vector<T> _vmin, _vmax, _vm, _vc, _vzero_val, _vmax_val;
};
```

Algorithm 10.17: `equalize` functor (multi-channel)

```
template<typename TL> struct equalize<ct::tuple<TL> > {
  private:
    typedef TL scalar_tl;
    typedef typename ct::apply<TL, vvm::add_vector>::type
      vector_tl;
  public:
    equalize(const ct::tuple<TL>& min,
             const ct::tuple<TL>& max) {
      meta::EFOR3<0, meta::Less, ct::length<TL>::value, +1,
        do_init>::exec(*this, min, max);
    }
    ct::tuple<scalar_tl> operator()(const ct::tuple<TL> a) {
      ct::tuple<scalar_tl> ret;
      meta::EFOR3<0, meta::Less, ct::length<scalar_tl>::value,
        +1, do_op>::exec(*this, ret, a);
      return ret;
    }
    ct::tuple<vector_tl> operator()(
    const typename vvm::add_vector<ct::tuple<TL> >::type & a) {
      ct::tuple<vector_tl> ret;
      meta::EFOR3<0, meta::Less, ct::length<vector_tl>::value,
        +1, do_op>::exec(*this, ret, a);
      return ret;
    }
  private:
   struct do_init {
      template<int i> struct Code {
        static void exec(equalize& self, const ct::tuple<TL>& min,
```

```
    const ct::tuple<TL>&  max) {
      typedef typename ct::type_at<TL, i>::type scalar_type;
      self._smin.template get<i>() = min.template get<i>();
      self._smax.template get<i>() = max.template get<i>();
      self._vmin.template get<i>() = min.template get<i>();
      self._vmax.template get<i>() = max.template get<i>();
      self._vzero_val.template get<i>() = 0;
      self._vmax_val.template get<i>() =
        std::numeric_limits<scalar_type>::max();
      // Calculate m & c
      self._sm.template get<i>() =
        (std::numeric_limits<scalar_type>::max() - 1) /
        (self._smax.template get<i>() -
         self._smin.template get<i>());
      self._sc.template get<i>() = 1 -
        self._sm.template get<i>() *
        self._smin.template get<i>();
      // Copy m & c to vvm::vector
      self._vm.template get<i>() = self._sm.template get<i>();
      self._vc.template get<i>() = self._sc.template get<i>();
    }
  };
};
struct do_op {
  template<int i> struct Code {
    static void exec(equalize& self, ct::tuple<scalar_tl>& ret,
    const ct::tuple<TL>&  a) {
      typedef typename ct::type_at<TL, i>::type scalar_type;
      if(a.template get<i>() < self._smin.template get<i>())
        ret.template get<i>() = 0;
      else if(a.template get<i>() > self._smax.template get<i>())
        ret.template get<i>() =
          std::numeric_limits<scalar_type>::max();
      else
        ret.template get<i>() = self._sm.template get<i>() *
          a.template get<i>() + self._sc.template get<i>();
    }
    static void exec(equalize& self, ct::tuple<vector_tl>& ret,
    const typename vvm::add_vector<ct::tuple<TL> >::type& a) {
```

```
        typedef typename ct::type_at<TL, i>::type scalar_type;
        ret.template get<i>() = vvm::madd(
          self._vm.template get<i>(),
          a.template get<i>(),
          self._vc.template get<i>());
        ret.template get<i>() = vvm::select(
          a.template get<i>() < self._vmin.template get<i>(),
          ret.template get<i>(),
          self._vzero_val.template get<i>());
        ret.template get<i>() = vvm::select(
          a.template get<i>() > self._vmax.template get<i>(,
          ret.template get<i>(),
          self._vmax_val.template get<i>());
      }
    };
  };
private:
  ct::tuple<TL> _smin, _smax, _sm, _sc;
  typename vvm::add_vector<ct::tuple<TL> >::type _vmin, _vmax,
    _vm, _vc, _vzero_val, _ vmax_val;
};
```

Table 10.5 Speedup attained by executing VVIS's `equalize` functor in AltiVec mode over scalar mode

	Single-Channel Image				RGB Image			
	320 ×960	640 ×960	960 ×960	1280 ×960	320 ×960	640 ×960	960 ×960	1280 ×960
`unsigned char`	0.7	0.8	0.8	0.8	2.7	2.7	2.8	2.9
`unsigned short int`	0.9	0.9	0.9	0.9	1.9	2.1	2.2	2.2
`unsigned int`	0.5	0.6	0.6	0.6	0.9	0.9	0.9	0.9
`float`	0.7	0.8	0.8	0.8	1.0	1.1	1.1	1.1
`double`	0.3	0.3	0.3	0.3	0.3	0.4	0.4	0.4

Table 10.5 shows that `equalize` did not experience a speedup for `unsigned char` because there is no native vector support for `vvm::madd` in AltiVec; the VVM implementation used did not use AltiVec to perform multiplication and addition separately, which it could have done. AltiVec provides multiply and add instructions for `shorts`, `ints` and `floats`. See Appendix B for a more information on VVM to AltiVec mappings. Types without AltiVec multiply and add instructions, were processed with the scalar processor.

Since the number of scalars in a `vvm::vector` in AltiVec mode is not one, and the VVM implementation used is only zero-cost when the number of elements is one, this multiplication and addition had additional overheads. As a result, VVIS was always slower in AltiVec mode for those types.

Threshold

Thresholding divides an image into two distinct areas based solely on the pixel value. Values greater than or equal to the threshold are set to `1`, while other regions are set to `0` (Nat 1999, Young et al. n.d.). However, for vector programs, instead of `1` and `0`, it is more efficient to set the values to `~0` and `0` respectively, because these are the values that vector comparison functions usually return for `true` and `false`. The following equation summarises the vectorised threshold function:

$$f(X_0) = \begin{cases} 0 & \text{if } X_0 \geq \text{threshold} \\ 0 & \text{if } X_0 < \text{threshold} \end{cases} \tag{10.4}$$

VVIS does not have a dedicated threshold functor. Instead, a threshold operation is composed from binders and comparative functions like `greater_equal`. The following statement performs a threshold, setting all pixels with value `>= 50` to `~0`, and `0` for the rest.

```
vvis::transform(img1, dest, vvis::pixel_accessor(),
  vvis::bind2nd(
    vvis::greater_equal<typename image_t::pixel_type>(), 50));
```

Table 10.6 shows the speedup attained for this statement. Speedup attained for threshold was lower than the speedup attained for the simpler `plus` functor. Threshold was faster in AltiVec mode when operating on `unsigned char`, and `unsigned short int` for single-channel images. As usual, the speedup attained was greater for RGB images, because of the slow scalar performance of VVIS with RGB images in scalar mode.

10.3 Convolutive operations

Convolutive operations (see Section 8.2) accept a rectangle of elements per input set, and produce a single output element for one output set.

10.3.1 Algorithms

A single convolutive algorithm is provided by VVIS, which accepts one input set and one output set.

Table 10.6 Speedup attained by executing a VVIS threshold functor in AltiVec mode over scalar mode

	Single-Channel Image				RGB Image			
	320 ×960	640 ×960	960 ×960	1280 ×960	320 ×960	640 ×960	960 ×960	1280 ×960
`unsigned char`	2.1	1.6	1.8	1.7	7.4	6.5	7.0	6.8
`unsigned short int`	1.0	1.2	1.3	1.3	2.8	3.1	3.1	3.1
`unsigned int`	0.9	1.0	1.1	1.1	1.4	1.4	1.4	1.4
`float`	0.9	1.1	1.1	1.1	1.4	1.5	1.5	1.5
`double`	0.8	0.9	0.9	0.9	0.9	0.9	0.9	0.9

```
template<
  typename storageInT,
  typename accessorInT,
  typename storageOutT,
  typename accessorOutT,
  typename functorT
> void convolute(const storageInT& in, accessorInT ia,
                 storageOut& out, accessorOutT oa, functorT f);
```

Unlike quantitative or transformative operations, convolutive algorithms require spatial iterators. The VVIS convolutive algorithm does not accept contiguous or unknown storages since these are one-dimensional, neither does it accept Illife storages with Illife storages because the storages will have three or more dimensions. It only accepts Illife storages with contiguous or unknown storages.

Scalar algorithm

An example of a scalar implementation of the convolutive algorithm is shown in Algorithm 10.18. The functor has four methods — `kernel_width`, `kernel_height`, `accumulate`, and `output`. The `kernel_width` and `kernel_height` functions return the width and height of the rectangular input expected respectively. The `accumulate` function is used to pass the input values to the functor. Input values are passed in order, row by row, from top to bottom. Because the output operation requires exactly the same number of input values and the order of the values is consistent, there is no need to explicitly inform `accumulate` which value was passed. After passing all the input values required for the current output to the functor through `accumulate`, `output` is called to get the output value.

Examples of generic convolutive scalar algorithms in VIGRA are `vigra::convolveImage`, `vigra::convolveImageWithMask`, `vigra::convolveLine`, `vigra::separableConvolveX`, and `vigra::separableConvolveY`. VIGRA provides different algori-

Algorithm 10.18 Convolutive generic scalar algorithm

```
template<class T, class F>
void convolute(T* in, int width, int height, T* out, F f)
{
  for(int y = 0; y < height - 2; ++y)
  {
    for(int x = 0; x < width - 2; ++x)
    {
        int idx2 = (y + 1) * height + x;
        int idx3 = (y + 2) * height + x;
        for(int i = 0; i < f.kernel_height(); ++i)
        {
            int idx = (y + i) * height + x;
            for(int j = 0; j < f.kernel_width(); ++j)
                f.accumulate(in[idx + j]);
        }
        out[y * height + x] = f.output();
    }
  }
}
```

thms for 1-D and 2-D convolutions. In addition, VIGRA's convolutive algorithms also accept parameters for controlling how border conditions are handled.

AltiVec algorithm

An AltiVec convolutive function, modified from Fuller (1999), is shown in Algorithm 10.3.1. This algorithm only operates on aligned data. Algorithm 10.3.1 can be modified to load unaligned data using the same method used on transformative algorithms in Section 3.7.

Algorithm 10.19: AltiVec convolutive algorithm modified from Fuller (1999)

```
void convolute(signed short int* input, int width, int height,
__vector signed short kernel[9], signed short int* output) {
  const int vsize = 8;
  __vector signed short zero =
    (vector signed short)(0, 0, 0, 0, 0, 0, 0, 0);
  // Load input data
  for(int i = 0; i < height - 2; ++i) {
    // Loop through each row
    // Load the pixels from the row assumes Image is 128b
    // aligned
    // Generate row pointers
```

```
signed short *r1_ptr = input + i*width;
signed short *r2_ptr = r1_ptr + width;
signed short *r3_ptr = r2_ptr + width;
__vector signed short r11s0, r12s0, r21s0, r22s0,
  r31s0, r32s0;
// Now work across the row
for(int j = 0; j < width/vsize; ++j) {
  __vector signed short result = zero;
  // Main calculation
  // Loads, shifts, mladds are interleaved
  r11s0 = vec_ld(0, r1_ptr);
  r12s0 = vec_ld(16, r1_ptr);
  // Process First Row
  __vector signed short r11s2 = vec_sld(r11s0, r12s0, 2);
  __vector signed short r11s4 = vec_sld(r11s0, r12s0, 4);
  result = vec_mladd(r11s0, kernel[0], result);
  result = vec_mladd(r11s2, kernel[1], result);
  result = vec_mladd(r11s4, kernel[2], result);
  // Process Next Row
  r21s0 = vec_ld(0, r2_ptr);
  r22s0 = vec_ld(16, r2_ptr);
  __vector signed short r21s2 = vec_sld(r21s0, r22s0, 2);
  __vector signed short r21s4 = vec_sld(r21s0, r22s0, 4);
  result = vec_mladd(r21s0, kernel[3], result);
  result = vec_mladd(r21s2, kernel[4], result);
  result = vec_mladd(r21s4, kernel[5], result);
  // Process Last Row
  r31s0 = vec_ld(0, r3_ptr);
  r32s0 = vec_ld(16, r3_ptr);
  __vector signed short r31s2 = vec_sld(r31s0, r32s0, 2);
  __vector signed short r31s4 = vec_sld(r31s0, r32s0, 4);
  result = vec_mladd(r31s0, kernel[6], result);
  result = vec_mladd(r31s2, kernel[7], result);
  result = vec_mladd(r31s4, kernel[8], result);
  // Store in output
  // End Results contains eight shorts
  // which are the convolution of the input image
  // with the kernel
  vec_st(result, 0,
```

```
                (__vector signed short*)&output[i*width+j*vsize]);
          r1_ptr += VectorSize;
          r2_ptr += VectorSize;
          r3_ptr += VectorSize;
        }
      }
    }
```

Algorithm 10.3.1 assumes that the `kernel`'s size is 3×3. The 3×3 kernel is represented as a one-dimensional array. An example of how the kernel might be constructed is shown below:

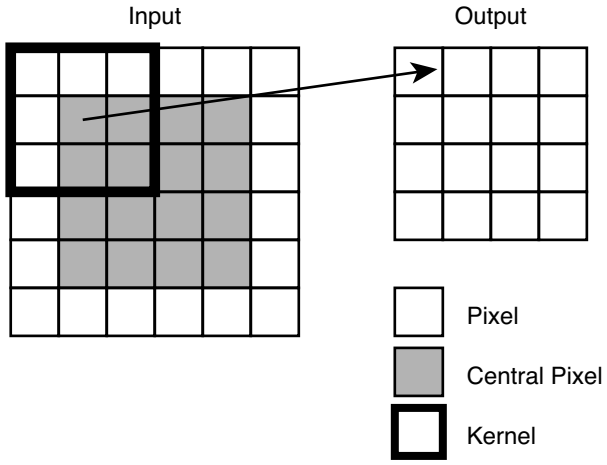
```
// Kernel is 3x3
//  1  -2   1
// -2   5  -2
//  1  -2   1
__vector signed short kernel[9] = {
  // Row 1
  (__vector signed short)( 1,  1,  1,  1,  1,  1,  1,  1),
  (__vector signed short)(-2, -2, -2, -2, -2, -2, -2, -2),
  (__vector signed short)( 1,  1,  1,  1,  1,  1,  1,  1),
  // Row 2
  (__vector signed short)(-2, -2, -2, -2, -2, -2, -2, -2),
  (__vector signed short)( 5,  5,  5,  5,  5,  5,  5,  5),
  (__vector signed short)(-2, -2, -2, -2, -2, -2, -2, -2),
  // Row 3
  (__vector signed short)( 1,  1,  1,  1,  1,  1,  1,  1),
  (__vector signed short)(-2, -2, -2, -2, -2, -2, -2, -2),
  (__vector signed short)( 1,  1,  1,  1,  1,  1,  1,  1)
};
```

Apart from supporting only kernel sizes of 3×3, Algorithm 10.3.1 also requires input and output images to be of the same type. Input and output images can only have a single channel, and the data must be aligned.

Generic VVIS algorithm

VVIS's convolutive algorithm has fewer features than VIGRA's. VIGRA's convolutive algorithms allow the user to choose how border conditions are handled and will offset the position that output is written to based on the size of the kernel automatically. VVIS's

Figure 10.5 Convolutive algorithm behaviour



convolutive algorithm is similar in functionality to the AltiVec convolution algorithm presented by Fuller (1999) (see Figure 10.5). There is actually no need to handle central pixels in the algorithm, since the user can control the destination by creating a region that offsets the storage automatically.

Converting the AltiVec algorithm discussed in Algorithm 10.3.1 to a generic version is unfortunately not as simple as converting the quantitative and transformative algorithms. A generic convolutive algorithm should be able to support different kernel sizes, different input and output image types, different input and output channel counts, and support unaligned rows.

Convolutive operations require spatial iterators. Because of this, convolutive algorithms only accept Illife storages with contiguous or unknown storages. The scalar convolutive algorithm operates on Illife storages with unknown storages, while the vectorised convolutive algorithm operates on Illife storages containing contiguous storages.

The convolutive algorithm for Illife storages containing unknown storages is pretty straightforward, because it only uses the scalar processor. As mentioned previously, scalar convolutive algorithms already exist in VIGRA. However, while in VIGRA, algorithms specify the operation applied to the pixels, in VVIS the operation is specified by the functor.

The generic VVIS convolutive algorithms for unknown and contiguous storages are presented in Algorithms 10.20 and 10.3.1 respectively. As discussed in Section 9.3.6, these algorithms are enabled using function enablers. The `priv::select_illife_storage` template metafunction will return `priv::using_contiguous_storage` if all the storages are Illife storages that contain contiguous storages. Otherwise, it will return `priv::using_unknown_storage` if all the storages are Illife storages that contain either contiguous or unknown storages. Since VVIS convolutive algorithms require storages to be rectangular, the enabler logic includes checks to ensure that all input and output

storages are rectangular.

As mentioned previously, Algorithm 10.3.1 requires kernel sizes of 3×3, input and output image types to be of the same type, input and output images to have only one channel, and the data must be aligned. On the other hand, a generic vectorised convolutive algorithm should support different input and output image types, multi-channel input and output images, unaligned data, and variable kernel sizes. Accessors handle multi-channel input and output, and allow the input and output types to be different. Because the `begin` function of contiguous storages always returns an aligned iterator, algorithms do not have to handle unaligned data or storages with different alignments. Unaligned data would be handled by the accessors and storages. Variable kernel sizes though are the responsibility of convolutive algorithm.

When supporting kernel sizes, variable kernel widths are more difficult to implement than variable kernel heights. To support variable kernel widths, the vectorised algorithm needs to load zero or more extra `vvm::vectors` in order to extract the required `vvm::vectors`. The exact number of extra loads required depends on the kernel width.

The basic code to retrieve the required `vvm::vector`, where `a` is the accessor, can be distilled to:

```
a.get_vector(FIRST_VECTOR, SECOND_VECTOR, SHIFT)
```

We only ever need to load from two `vvm::vectors` because the resultant `vvm::vector` cannot be longer than one `vvm::vector` length.

The first `vvm::vector` (`FIRST_VECTOR`) is as follows, where `pi` is the current iterator position and `kx` is *x*-coordinate in the kernel. If `kx` reaches `VVM_SCALAR_COUNT`, then we need to load the `vvm::vector` after `pi` instead of the `vvm::vector` located at `pi`.

```
a.get_vector(pi, kx / VVM_SCALAR_COUNT)
```

The second `vvm::vector` (`SECOND_VECTOR`) is:

```
a.get_vector(pi, kx / VVM_SCALAR_COUNT + 1)
```

`SHIFT` is therefore:

```
kx % VVM_SCALAR_COUNT
```

Thus the code required to load the appropriate `vvm::vectors` in the *x*-coordinate is

```
for(int kx = 0; kx < f.kernel_width; ++kx) {
  f.accumulate(
    a.get_vector(
      a.get_vector(pi, kx / VVM_SCALAR_COUNT),
      a.get_vector(pi, kx / VVM_SCALAR_COUNT + 1),
      kx % VVM_SCALAR_COUNT));
}
```

Algorithm 10.20 Convolutive generic VVIS algorithm for Illife storages containing unknown storages

```
template<
  typename storageInT,
  typename accessorInT,
  typename storageOutT,
  typename accessorOutT,
  typename functorT
> typename ct::enable_if<
  boost::is_same<
    typename priv::select_illife_unknown<
      CT_TYPELIST2(storageInT, storageOutT)>::type,
    priv::using_unknown_storage
  >::value &&
  is_rectangular<storageInT>::value &&
  is_rectangular<storageOutT>::value
>::type convolute(const regionInT& in, regionOutT& out,
                  accessorT a, functorT f) {
  typename regionInT::const_iterator iy = in.begin();
  typename regionInT::const_iterator iy_end = in.end() -
    (f.kernel().height() - 1);
  typename regionInT::iterator oy = out.begin();
  for(; iy != iy_end; ++iy, ++oy) {
    typename regionInT::value_type::const_iterator pi =
      iy->begin();
    typename regionInT::value_type::const_iterator end =
      iy->end();
    end -= f.kernel().width() - 1;
    typename regionOutT::value_type::iterator po = oy->begin();
    int vi = 0;
    for(int i = 0; pi != end; ++i, ++pi, ++po) {
      f.reset();
      for(int ky = 0; ky < f.kernel().height(); ++ky) {
        typename regionInT::value_type::const_iterator itr =
          (iy + ky)->begin() + i;
       for= 0; kx < f.kernel().width(); ++kx) {
          f.accumulate(a.get_scalar(itr + i));
        }
      }
      a.set_scalar(f.scalar_result(), po);
    }
  }
}
```

The next step is to calculate `pi`. The `pi` iterator contains the position in the row that we are interested in currently. To calculate each row of an output image, `ky` rows from the input image are needed. While it is easy enough to get the starting position of each required row using `in[y + ky]`, it is more difficult to get to the position required when it is not the starting position. Since each row in an Illife storage is contiguous storage, if we keep the number of `vvm::vector` steps walked so far, we can calculate the required `pi` as:

```
pi = in[y + ky].begin();
pi.vector += i;
```

Algorithm 10.3.1 shows the vectorised VVIS convolutive algorithm for Illife storages with contiguous storages. For performance, the final algorithm prefetchs, keeps an array of row iterators and keeps an array of `vvm::vectors`.

Prefetching: Since VVM prefetching instructions require a prefetch channel to be a literal, in order to prefetch all the rows required in different channels, code similar to the following is required:

```
for(int ky = 0; ky < f.kernel_height(); ++ky) {
  switch(ky) {
    case 0: /* Prefetch channel 0 */; break;
    case 1: /* Prefetch channel 1 */; break;
    case 2: /* Prefetch channel 2 */; break;
    // ... and so on
  default:
    // No more prefetch channels available
    goto continue_processing;
  }
}
continue_processing:
// Continue processing here
```

Since this prefetching method appears to be computationally expensive, Algorithm 10.3.1 only prefetchs the input row `in[y]` (`ky` is `0`) and the output row `out[y]`.

Keeps array of row iterators: The algorithm keeps an array with `f.kernel_height()` row iterators to avoid having to continuously recalculate `pi`. The array definition is as follows:

```
typename storageInT::value_type::const_iterator
  pi[kernel_height];
```

The array of `pi`s is initialised at the start of every output row as:

```
for(int i = 0; i < kernel_height; ++i) {
  pi[i] = in[y + i].begin();
}
```

They are advanced while accumulating:

```
f.reset()
for(int ky = 0; ky < kernel_height; ++ky) {
  // Load vectors and accumulate goes here ...
  // Advance pi
  ++pi[ky].vector;
}
// Write output f.vector_result() ...
```

Keeps array of vvm::vectors: Because `vvm::vector`s loaded from an iteration of `kx` are likely to be used again, the algorithm loads all the `vvm::vector`s required once. Since the number of `vvm::vector`s that need to be loaded is `kernel_width / VVM_SCALAR_COUNT + 2`, we can load all the `vvm::vector`s necessary into an array using code similar to the following:

```
const int req_vcount = kernel_width / VVM_SCALAR_COUNT + 2
typename accessorInT::vector_type vcache[req_vcount];
// Load vectors
for(int i = 0; i < req_vcount; ++i) {
  vcache[i] = ia.get_vector(pi[ky], i);
}
```

The accumulate loop thus becomes the following. Note that the first `kx` is accumulated directly.

```
f.accumulate(vcache[0]);
for(int kx = 1; kx < f.kernel_width(); ++kx) {
  f.accumulate(
    ia.get_vector(vcache[kx / VVM_SCALAR_COUNT],
                  vcache[kx / VVM_SCALAR_COUNT + 1],
                  kx % VVM_SCALAR_COUNT));
}
```

Algorithm 10.21: Convolutive generic VVIS algorithm for Illife storages containing contiguous storages

```
template<
  typename storageInT,
  typename accessorInT,
  typename storageOutT,
  typename accessorOutT,
  typename functorT
> typename ct::enable_if<
  boost::is_same<
    typename priv::select_illife_unknown<
      CT_TYPELIST2(storageInT, storageOutT)>::type,
    priv::using_contiguous_storage
  >::value &&
  is_rectangular<storageInT>::value &&
  is_rectangular<storageOutT>::value
>::type convolute(const regionInT& in, regionOutT& out,
                  accessorT a, functorT f) {
  const int kernel_width = f.kernel_width();
  const int kernel_height = f.kernel_height();
  const int height = in.size() - kernel_height + 1;
  const int req_vcount = kernel_width / VVM_SCALAR_COUNT + 2;
  typename accessorInT::vector_type vcache[req_vcount];
  typename storageInT::value_type::const_iterator
    pi[kernel_height];
  for(int y = 0; y < height; ++y) {
    for(int i = 0; i < kernel_height; ++i) {
      pi[i] = in[y + i].begin();
    }
    typename storageInT::value_type::const_iterator end =
      in[y].end() - (kernel_width - 1);
    typename storageOutT::value_type::iterator po =
      out[y].begin();
    // Apply functor the vector processor
    for(; pi[0].vector != end.vector; ++po.vector) {
      ia.template prefetch_read<0>(pi[0]);
      oa.template prefetch_write<
        accessorInT::prefetch_channel_count>(po);
```

```
        f.reset();
        for(int ky = 0; ky < kernel_height; ++ky) {
          // Load vectors
          for(int i = 0; i < req_vcount; ++i) {
            vcache[i] = ia.get_vector(pi[ky], i);
          }
          // Now accumulate
          f.accumulate(vcache[0]);
          for(int kx = 1; kx < kernel_width; ++kx) {
            f.accumulate(
              ia.get_vector(vcache[kx / VVM_SCALAR_COUNT],
                vcache[kx / VVM_SCALAR_COUNT + 1],
                kx % VVM_SCALAR_COUNT));
          }
          ++pi[ky].vector; // Advance pi
        }
        oa.set_vector(f.vector_result(), po);
      }
      // Apply functor using the scalar processor
      for(; pi[0].scalar != end.scalar; ++po.scalar) {
        f.reset();
        for(int ky = 0; ky < kernel_height; ++ky) {
          for(int kx = 0; kx < kernel_width; ++kx) {
            f.accumulate(ia.get_scalar(pi[ky] + kx));
          }
          ++pi[ky].scalar; // Advance pi
        }
        oa.set_scalar(f.scalar_result(), po);
      }
    }
  }
```

Results

Figures 10.6 and 10.7 show how the performance of the VVIS convolutive algorithm, presented in Section 10.3.1, compares with hand-coded scalar and AltiVec, and VIGRA programs. All implementations were compiled using Apple GCC 3.1 20021003, with the `-Os` (optimise for size) switch. Times presented include the cost of constructing the kernel functor, which used a 3×3 `signed char` kernel. Different source and output images were

used.

The implementations were evaluated for `unsigned char` and `signed short` images. Images of `signed shorts` were evaluated because Algorithm 10.3.1 used `signed short` images. In addition, AltiVec does not provide vector multiply and add instructions for `char` AltiVec types; AltiVec only provides vector multiply and add instructions for `short`, `int` and `float` AltiVec types.

VIGRA: This version used VIGRA's `convolveImage` algorithm. VIGRA uses a contiguous chunky scalar storage format. When processing RGB images, VIGRA uses `vigra::RGBColor` to represent a pixel.

Scalar: This version is a hand-coded scalar loop that processed contiguous planar storages. Since the kernel and the image types are different, automatic type conversions were performed.

Scalar Mode, VVIS (No Prefetch): This version uses VVIS's `convolve` algorithm, but without prefetching, and without a VPU. This version uses automatic unknown storages and specialised iterators for single-channel images. Since this version uses automatic unknown storages and there is no VPU, the convolve algorithm used processed unknown storages using the scalar implementation.

Both kernel and intermediate results were promoted to the same type explicitly. For `unsigned char` and `signed short` images, the promoted type used was `signed short` and `signed int` respectively.

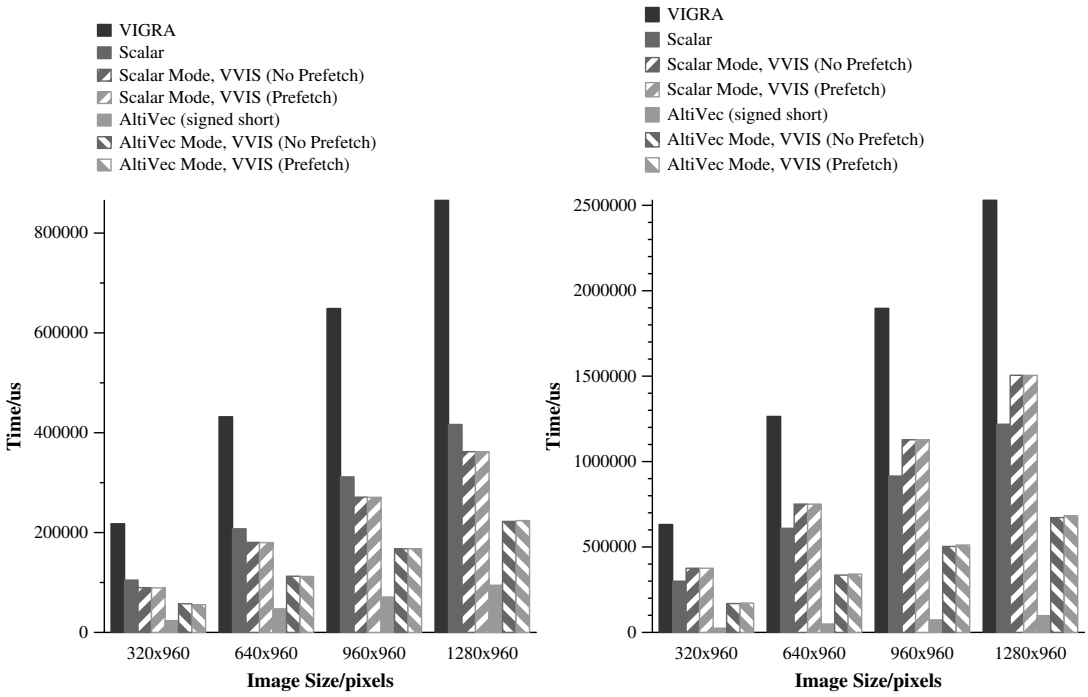
Scalar Mode, VVIS (Prefetch): This version is the same as Scalar Mode, VVIS (No Prefetch) except prefetching was requested. Since prefetching requests are ignored in scalar mode, prefetching is not actually enabled and thus this version is expected to perform identically to Scalar Mode, VVIS (No Prefetch).

AltiVec: This version is a hand-coded AltiVec program, similar to Algorithm 10.3.1 except the kernel accepted was an array of `signed chars` instead of `__vector signed shorts`. This version will convert this kernel to an array of `__vector signed shorts` using the `splat` function from Algorithm 3.6.

AltiVec Mode, VVIS (No Prefetch): This version uses VVIS's `convolve` algorithm, without prefetching, and with AltiVec. This version uses automatic unknown storages and specialised iterators for single-channel images.

Both kernel and intermediate results were promoted to the same type explicitly. For `unsigned char` and `signed short` images, the promoted type used was `signed short` and `signed int` respectively.

Figure 10.6 Performance of different `vvis::convolute` implementations when operating on `unsigned char` images



(a) Single-Channel Image

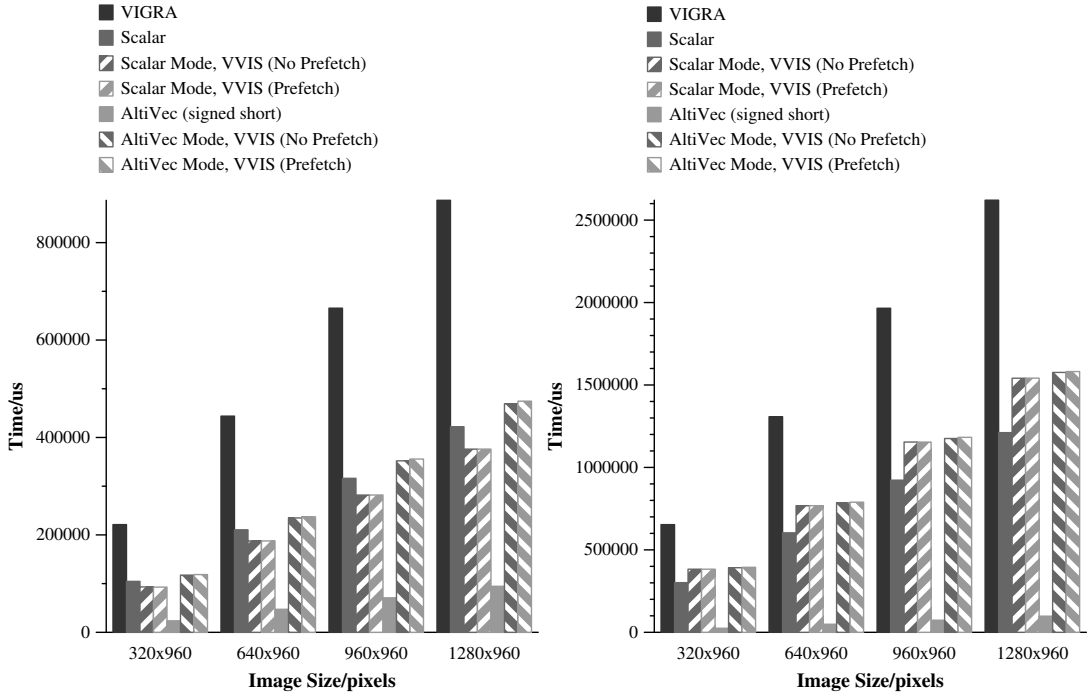
(b) RGB Image

AltiVec Mode, VVIS (Prefetch): This version is the same as AltiVec Mode, VVIS (No Prefetch) except prefetching was requested. As mentioned previously, VVIS's `convolute` algorithm only prefetchs the current input and output rows.

The promoted type in Figure 10.6 is `signed short`. This is the type that VVIS implementations are operating in. When processing single-channel images in scalar mode VVIS was slightly faster than the hand-coded scalar algorithm, which is consistent with observations on the VVIS transformative algorithm. When processing RGB images in scalar mode, VVIS was slower than the hand-coded scalar algorithm. While this is also consistent with the VVIS transformative algorithm, the difference was much smaller. In AltiVec mode, prefetching did not have any effect for either single channel or multi-channel storages. In scalar mode, prefetching requests are ignored, so as expected, prefetching had no effect. VVIS was faster than scalar hand-coded and VIGRA programs even though the `linear_filter` functor used is actually operating on `signed shorts`. This is despite the performance of the VVM implementation being approximately 17% to 20% slower than a hand-coded AltiVec program.

Figure 10.7 shows the same relationship in scalar mode between hand-coded scalar and VVIS programs as Figure 10.6. Since the images consisted of `signed shorts` and the kernel type was `signed char`, the promoted type was `signed int`. Despite the VVM

Figure 10.7 Performance of different `vvis::convolute` implementations when operating on `signed short` images



(a) Single-Channel Image

(b) RGB Image

implementation used having about the same relative overheads when processing `signed int` as `signed short` data in AltiVec mode and VVIS being faster in AltiVec mode when the promoted type was `signed short`, VVIS was slower in AltiVec mode than in scalar mode when operating on the promoted type, `signed int`, because the number of scalars in a VPU vector dropped from eight to four.

VVIS in AltiVec mode was always slower than AltiVec hand-coded program. This difference can be attributed to the algorithm supporting variable kernel sizes and VVM's load from two `vvm::vectors` function. Because this VVM load function allows the number of scalars to be selected from each `vvm::vectors` to be specified as a variable, VVM loads unaligned using the AltiVec `vec_perm` permutation function instead of the `vec_sld` used by the AltiVec hand-coded programs. The `vec_sld` function is faster than `vec_perm`.

Prefetching did not have any effect on convolutive algorithms. This suggests that either the data were always already available in the caches, or that it never was. While VIGRA was slower in all cases tested, this is probably because of the additional features provided by the VIGRA convolution algorithm. VIGRA's convolution algorithm, for example, allows the user to specify how the algorithm should process borders.

Table 10.7 Convolutive algorithms' required functor interface

Expression	Return Type	Notes
`A::kernel_type`	Type	Type of `a.kernel()`
`a.reset()`	`void`	Signals start of new rectangle input
`a.accumulate(s)`	`void`	Accumulate a scalar
`a.accumulate(v)`	`void`	Accumulate a vector
`a.scalar_result()`	scalar	Returns the scalar answer
`a.vector_result()`	vector	Returns the vector answer
`a.kernel_width()`	`const int`	Returns the width of the kernel
`a.kernel_height()`	`const int`	Returns the height of the kernel

10.3.2 Functors

Convolutive operations' functor's requirements are more involved than quantitative and transformative operations'. The expected interface for a convolutive functor in VVIS is shown in Table 10.7. As a result of using the categorisation scheme detailed in Chapter 8, in VVIS the operation to be performed on the data is part of the functor and not the algorithm; in VIGRA, the operation is part of the algorithm. Because of this, convolutive algorithms in VVIS do not require the contents of the kernel; they only require the size. Furthermore, because the operation is not part of the algorithm, fewer algorithms are required in VVIS than in VIGRA. Like the VVIS functors required for quantitative and transformative operations, scalar and `vvm::vector` versions of its accumulate and output functions are required.

This section starts with a discussion on the implementation of `base_filter` which is the super-class of the other filters featured in VVIS. The `base_filter` class handles the allocation and deallocation of the kernel and calculates the sum of all values in a kernel. The implementation of two other filters, `linear_filter` and `max_filter`, are then described. These filters differ in what operation they perform in the `accumulate` functions.

Base filter

The `base_filter` class handles the allocation and deallocation of the kernel and calculates the sum of all values in a kernel. The type that the filter should be working with is determined by `base_filter` via a promotion policy passed in through `promoteP`. By default, `promoteP` is `ct::promote`. The user can disable promotions by passing a different template metafunction through `promoteP`, if the user is sure that the intermediate results will not overflow. The `base_filter` class converts the kernel into an array of `vvm::vectors`.

Algorithms 10.3.2 and 10.3.2 show how `base_filter` is implemented for single-

channel and multi-channel storages respectively. The `base_filter` class for multi-channel images also handles kernel allocation and deallocation. However, in this case there is now a list of types. The `base_filter` class could have allocated an array for each channel separately, but this would have been a waste of memory since there are usually duplicate types in the typelist. Instead, `base_filter` uses `ct::no_duplicates` to reduce the list to unique types, and then creates and maintains an array for each of these unique types. Thus for a RGB image that has three `unsigned char` channels, `base_filter` will only keep one `vvm::vector<T>` array of kernel values, where `T` is the promoted type. The same applies to the sum of all values in the kernel. The `vector_kernel()` and `vector_sum()` template functions provide easy access to the `vvm::vector` kernel and sum for a given type respectively.

Algorithm 10.22: `base_filter` (single-channel)

```
template<
  typename kernelT,
  typename T,
  template<typename> class promoteP = ct::promote
> class base_filter {
protected:
  typedef typename promoteP<CT_TYPELIST2(kernelT, T)>::type
    scalar_type;
  typedef vvm::vector<scalar_type> vector_type;
public:
  typedef kernel2d<kernelT> kernel_type;
  base_filter() : _vector_kernel(0) {
  }
  base_filter(const base_filter& rhs)
  : _kernel(rhs._kernel) {
    allocate();
  }
  base_filter(kernel_type kernel)
  : _kernel(kernel) {
   allocate();
  }
  base_filter& operator=(const base_filter& rhs) {
   // Check if the same object
   if(&rhs == this)
      return *this;
    // Deallocate memory
    deallocate();
```

```
    // Then make copy
    _kernel = rhs._kernel;
    allocate();
    return *this;
  }
  const kernel_type kernel() const {
    return _kernel;
  }
protected:
  // Destructor is protected so that derived classes do not
  // have to be virtual. In addition, it prevent base_filter
  // from being used by the user.
  ~base_filter() {
    deallocate();
  }
private:
  void allocate() {
    _vector_kernel = new vector_type[_kernel.width()*
                                     _kernel.height()];
    typename kernel_type::const_iterator itr = _kernel.begin();
    for(int i = 0; itr != _kernel.end(); ++itr, ++i) {
      _vector_kernel[i] = *itr;
    }
    _vector_sum = _kernel.sum();
  }
  void deallocate() {
    if(_vector_kernel)
      delete[] _vector_kernel;
  }
protected:
  kernel_type _kernel;
  vector_type _vector_sum;
  vector_type* _vector_kernel;
};
```

Algorithm 10.23: `base_filter` (multi-channel)

```
template<
  typename kernelT,
```

```
  typename TL,
  template<typename> class promoteP
> class base_filter<kernelT, ct::tuple<TL>, promoteP> {
protected:
  typedef typename priv::apply_promote_with_kernel<kernelT,
    TL, promoteP>::type scalar _tl;
  typedef typename ct::apply<scalar_tl,
    vvm::add_vector>::type vector_tl;
private:
  typedef typename ct::no_duplicates<scalar_tl>::type
    scalar_kernel_tl;
  typedef typename ct::no_duplicates<vector_tl>::type
    vector_kernel_tl;
public:
  typedef kernel2d<kernelT> kernel_type;
  base_filter() : _vector_kernel(0) {
  }
  base_filter(const base_filter& rhs)
  : _kernel(rhs._kernel) {
    allocate();
  }
  base_filter(kernel_type kernel)
  : _kernel(kernel) {
    allocate();
  }
  ~base_filter() {
    deallocate();
  }
public:
  const kernel_type kernel() const {
    return _kernel;
  }
protected:
  // Retrieves the kernel for the type T
  template<typename T>
  vvm::vector<T>* vector_kernel() const {
    const int ki = ct::index_of<scalar_kernel_tl, T>::value;
    return _vector_kernel.template get<ki>();
  }
```

```
  // Retrieves the kernel sum for the type T
  template<typename T>
  vvm::vector<T> vector_sum() const {
    const int ki = ct::index_of<scalar_kernel_tl, T>::value;
    return _vector_sum.template get<ki>();
  }
private:
  void allocate() {
    meta::EFOR1<0, meta::Less,
      ct::length<vector_kernel_tl>::value,
      +1, do_allocate>::exec(*this);
  }
  void deallocate() {
    meta::EFOR1<0, meta::Less,
      ct::length<vector_kernel_tl>::value,
      +1, do_deallocate>::exec(*this);
  }
private:
  struct do_allocate {
    template<int i> struct Code {
      static void exec(base_filter& self) {
        typedef typename ct::type_at<vector_kernel_tl,
          i>::type vector_type;
        typedef typename ct::type_at<scalar_kernel_tl,
          i>::type scalar_type;
        self._vector_kernel.template get<i>() =
          new vector_type[
            self._kernel.width()*self._kernel.height()];
        typename kernel_type::const_iterator itr =
          self._kernel.begin();
        for(int j = 0;
            itr != self._kernel.end();
            ++itr, ++j) {
          self._vector_kernel.template get<i>()[j] = *itr;
        }
        self._vector_sum.template get<i>() =
          self._kernel.sum();
      }
    };
```

```
  };
  struct do_deallocate {
    template<int i> struct Code {
      static void exec(base_filter& self) {
        if(self._vector_kernel.template get<i>())
          delete[] self._vector_kernel.template get<i>();
      }
    };
  };
private:
  kernel_type _kernel;
  ct::tuple<typename ct::apply<vector_kernel_tl,
    boost::add_pointer>::type> _vector_kernel;
  ct::tuple<vector_kernel_tl> _vector_sum;
};
```

Linear filter

The `linear_filter` class multiplies the input pixel with the appropriate kernel value, sums up the results and returns the larger of the sum divided by the sum of the kernel values or 1. Examples of such operations are gradient filters, Laplacian filters, smoothing filters and Gaussian filters. The only difference between these operations is the value of the kernel.

Since the relationships between the kernel values are more important than the actual value, the user can use different kernel values to achieve the same result. For example, Nat (1999) and Fuller (1999) used integers, while some examples from Köthe (2001) used real numbers. Users have control over the kernel type and over the values used.

The scalar `accumulate` is similar to the following, where `_scalar_result` is the immediate sum of the multiplication of the pixels and the kernel, `v` is the pixel value, `_vector_kernel` is the kernel represented as an array of `vvm::vectors`, and `_i` is the current position in the kernel. This scalar version depends on automatic conversions.

```
_scalar_result += v * _vector_kernel[_i].scalar(0);
++_i;
```

The `vvm::vector accumulate` can use the `vvm::madd` function to perform a multiplication and an addition simultaneously. `_vector_result` is a `vvm::vector` containing the immediate sum of multiplication of pixels and the kernel. The `vector_type` type is the promoted `vvm::vector` type. It is the same type as `_vector_kernel`'s elements and `_vector_result`.

```
_vector_result = vvm::madd(vvm::vector_cast<vector_type>(v),
                 _vector_kernel[_i], _vector_result);
++_i;
```

Type conversions are required to support negative kernel values when operating on unsigned images and to prevent overflow.

The `linear_filter` class is shown in Algorithms 10.3.2 and 10.3.2. Algorithm 10.3.2 applies to single-channel storages while Algorithm 10.3.2 applies to multi-channel storages. In both cases, `linear_filter` inherits from `base_filter` and thus does not have to handle the allocation and deallocation of the kernel and the sum of values in the kernel. Like `base_filter`, by default, `linear_filter` uses `ct::promote` to promote the values before use. This promotion behaviour can be changed by passing a different promotion template metafunction as `promoteP`.

Algorithm 10.24: `linear_filter` functor (single-channel)

```
template<
  typename kernelT,
  typename T,
  template<typename> class promoteP = ct::promote
> class linear_filter
: public base_filter<kernelT, T, promoteP> {
private:
  typedef base_filter<kernelT, T, promoteP> parent_type;
  typedef typename parent_type::scalar_type scalar_type;
  typedef typename parent_type::vector_type vector_type;
public:
  typedef typename parent_type::kernel_type kernel_type;
  linear_filter() : parent_type() {
  }
  linear_filter(const linear_filter& rhs)
  : parent_type(rhs), _scalar_result(rhs._scalar_result),
  _vector_result(rhs._vector_result), _i(rhs._i) {
  }
  linear_filter(kernel_type kernel)
  : parent_type(kernel) {
  }
  ~linear_filter() {
  }
public:
  void reset() {
```

```
    _i = 0;
    _scalar_result = 0;
    _vector_result = 0;
  }
  const scalar_type scalar_result() const {
    return _scalar_result / _kernel.sum();
  }
  const vector_type vector_result() const {
    return _vector_result / _vector_sum;
  }
  void accumulate(const T& v) {
    _scalar_result += v * _vector_kernel[_i].scalar(0);
    ++_i;
  }
  void accumulate(const vvm::vector<T>& v) {
    _vector_result = madd(vvm::vector_cast<vector_type>(v),
      _vector_kernel[_i],
      _vector_result);
    ++_i;
  }
private:
  scalar_type _scalar_result;
  vector_type _vector_result;
  int _i;
};
```

Algorithm 10.25: `linear_filter` functor (multi-channel)

```
template<
  typename kernelT,
  typename TL,
  template<typename> class promoteP
> class linear_filter<kernelT, ct::tuple<TL>, promoteP>
: public base_filter<kernelT,  ct::tuple<TL>, promoteP> {
private:
  typedef base_filter<kernelT, ct::tuple<TL>, promoteP>
    parent_type;
  typedef typename parent_type::scalar_tl scalar_tl;
  typedef typename parent_type::vector_tl vector_tl;
```

```
public:
  typedef typename parent_type::kernel_type kernel_type;
  linear_filter() : parent_type() {
  }
  linear_filter(const linear_filter& rhs)
  : parent_type(rhs), _scalar_result(rhs._scalar_result),
  _vector_result(rhs._vector_result), _i(rhs._i) {
  }
  linear_filter(kernel_type kernel)
  : parent_type(kernel) {
  }
  ~linear_filter() {
  }
public:
  void reset() {
    _i = 0;
    _scalar_result = 0;
    _vector_result = 0;
  }
  const ct::tuple<scalar_tl> scalar_result() const {
    ct::tuple<scalar_tl> ret;
    meta::EFOR2<0, meta::Less,
      ct::length<scalar_tl>::value,
      +1, do_result>::exec(*this, ret);
    return ret;
  }
  const ct::tuple<vector_tl> vector_result() const {
    ct::tuple<vector_tl> ret;
    meta::EFOR2<0, meta::Less,
      ct::length<vector_tl>::value,
      +1, do_result>::exec(*this, ret);
    return ret;
  }
  void accumulate(const ct::tuple<TL>& v) {
    meta::EFOR2<0, meta::Less,
      ct::length<TL>::value,
      +1, do_accumulate>::exec(*this,  v);
    ++_i;
  }
```

```
  void accumulate(
  const typename vvm::add_vector<ct::tuple<TL> >::type& v) {
    meta::EFOR2<0, meta::Less, ct::length<TL>::value, +1,
      do_accumulate>::exec(*this,  v);
    ++_i;
  }
private:
  struct do_accumulate {
    template<int i> struct Code {
      static void exec(linear_filter& self,
                       const ct::tuple<TL>& v) {
        typedef typename ct::type_at<scalar_tl, i>::type
          scalar_type;
        self._scalar_result += v.template get<i>() *
          self.template vector_kernel<scalar_type>()
            [self._i].scalar(0);
      }
      static void exec(linear_filter& self,
      const typename vvm::add_vector<
      ct::tuple<T L> >::type& v) {
        typedef typename ct::type_at<scalar_tl, i>::type
          scalar_type;
        typedef typename ct::type_at<vector_tl, i>::type
          vector_type;
        self._vector_result.template get<i>() = madd(
          vvm::vector_cast<vector_type>(
            v.template get<i>()),
          self.template vector_kernel<scalar_type>()
            [self._i],
          self._vector_result.template get<i>());
      }
    };
  };
  struct do_result {
    template<int i> struct Code {
      static void exec(const linear_filter& self,
                       ct::tuple<scalar_tl>& result) {
        typedef typename ct::type_at<scalar_tl, i>::type
          scalar_type;
```

```
      result.template get<i>() =
        self._scalar_result.template get<i>() /
        self.template vector_sum<scalar_type>().
          scalar(0);
    }
    static void exec(const linear_filter& self,
                     ct::tuple<vector_tl>& result) {
      typedef typename ct::type_at<scalar_tl, i>::type
        scalar_type;
      result.template get<i>() =
        self._vector_result.template get<i>() /
        self.template vector_sum<scalar_type>();
    }
  };
 };
private:
 ct::tuple<scalar_tl> _scalar_result;
 ct::tuple<vector_tl> _vector_result;
 int _i;
};
```

Table 10.8 shows the speedup attained by executing `linear_filter` in AltiVec mode over scalar mode. While `linear_filter` was 1.6 times faster in AltiVec mode when operating on `unsigned char` images, its speedup was between 0.3 and 0.5 when operating on images with `vvm::vectors` that contained more than one VPU vector.

Table 10.8 Speedup attained by executing VVIS's `linear_filter` functor with a 3×3 `signed char` kernel in AltiVec mode over scalar mode

	Single-Channel Image				RGB Image			
	320 ×960	640 ×960	960 ×960	1280 ×960	320 ×960	640 ×960	960 ×960	1280 ×960
`unsigned char`	1.6	1.6	1.6	1.6	2.2	2.2	2.2	2.2
`unsigned short int`	0.7	0.7	0.7	0.7	0.8	0.8	0.8	0.8
`unsigned int`	0.5	0.5	0.5	0.5	0.5	0.5	0.6	0.6
`float`	0.6	0.6	0.6	0.6	0.7	0.7	0.8	0.7
`double`	0.5	0.5	0.5	0.5	0.5	0.6	0.6	0.6

Max filter

The main difference between `max_filter` and `linear_filter`, which was discussed in the previous section, is how it accumulates and returns the result. `max_filter` multiplies

the each input by the required kernel value, and selects the largest value as its output. Examples of `max_filter`s are nonlinear Prewitt filters and nonlinear Sobel filters.

The scalar `accumulate` is shown below:

```
scalar_type val = v * _vector_kernel[_i].scalar(0);
if(val > _scalar_result)
  _scalar_result = val;
++_i;
```

The `vvm::vector accumulate` uses the `vvm::select` operation to keep only the largest values:

```
vector_type val =
  vvm::vector_cast<vector_type>(v) * _vector_kernel[_i];
_vector_result =
  vvm::select(val > _vector_result, val, _vector_result);
++_i;
```

Algorithms 10.3.2 and 10.3.2 show how `max_filter` is implemented for single-channel and multi-channel storages respectively. Like `linear_filter`, it inherits from `base_filter`, which handles the kernel and the sum of all values in the kernel.

Algorithm 10.26: `max_filter` functor (single-channel)

```
template<
  typename kernelT,
  typename T,
  template<typename> class promoteP = ct::promote
> class max_filter : public base_filter<kernelT, T, promoteP> {
private:
  typedef base_filter<kernelT, T, promoteP> parent_type;
  typedef typename parent_type::scalar_type scalar_type;
  typedef typename parent_type::vector_type vector_type;
public:
  typedef typename parent_type::kernel_type kernel_type;
  max_filter() : parent_type() {
  }
  max_filter(const max_filter& rhs)
  : parent_type(rhs), _scalar_result(rhs._scalar_result),
  _vector_result(rhs._vector_result), _i(rhs._i) {
  }
  max_filter(kernel_type kernel)
```

```
  : parent_type(kernel) {
  }
  ~max_filter() {
  }
public:
  void reset() {
    _i = 0;
    _scalar_result = 0;
    _vector_result = 0;
  }
  const scalar_type scalar_result() const {
    return _scalar_result;
  }
  const vector_type vector_result() const {
    return _vector_result;
  }
  void accumulate(const T& v) {
    scalar_type val = v * _vector_kernel[_i].scalar(0);
    if(val > _scalar_result)
      _scalar_result = val;
    ++_i;
  }
  void accumulate(const vvm::vector<T>& v) {
    vector_type val = vvm::vector_cast<vector_type>(v) *
                      _vector_kernel[_i];
    _vector_result = vvm::select(val > _vector_result, val,
                                 _vector_result);
    ++_i;
  }
private:
  scalar_type _scalar_result;
  vector_type _vector_result;
  int _i;
};
```

Algorithm 10.27: `max_filter` functor (multi-channel)

```
template<
  typename kernelT,
```

```
  typename TL,
  template<typename> class promoteP
> class max_filter<kernelT, ct::tuple<TL>, promoteP>
: public base_filter<kernelT, ct: :tuple<TL>, promoteP> {
private:
  typedef base_filter<kernelT, ct::tuple<TL>, promoteP>
    parent_type;
  typedef typename parent_type::scalar_tl scalar_tl;
  typedef typename parent_type::vector_tl vector_tl;
public:
  typedef typename parent_type::kernel_type kernel_type;
  max_filter() : parent_type() {
  }
  max_filter(const max_filter& rhs)
  : parent_type(rhs), _scalar_result(rhs._scalar_result),
  _vector_result(rhs._vector_result), _i(rhs._i) {
  }
  max_filter(kernel_type kernel) : parent_type(kernel) {
  }
  ~max_filter() {
  }
public:
  void reset() {
    _i = 0;
    _scalar_result = 0;
    _vector_result = 0;
  }
  const ct::tuple<scalar_tl> scalar_result() const {
    return _scalar_result;
  }
  const ct::tuple<vector_tl> vector_result() const {
    return _vector_result;
  }
  void accumulate(const ct::tuple<TL>& v) {
    meta::EFOR2<0, meta::Less, ct::length<TL>::value, +1,
      do_accumulate>::exec(*this,  v);
    ++_i;
  }
  void accumulate(
```

```
  const typename vvm::add_vector<ct::tuple<TL> >::type& v) {
    meta::EFOR2<0, meta::Less, ct::length<TL>::value, +1,
      do_accumulate>::exec(*this,  v);
    ++_i;
  }
private:
  struct do_accumulate {
    template<int i> struct Code {
      static void exec(max_filter& self,
                       const ct::tuple<TL>& v) {
        typedef typename ct::type_at<scalar_tl, i>::type
          scalar_type;
        scalar_type val =  v.template get<i>() *
          self.template vector_kernel<scalar_type>()[self._i]
            .scalar(0);
        if(val > self._scalar_result.template get<i>())
          self._scalar_result.template get<i>() = val;
      }
      static void exec(max_filter& self,
      const typename vvm::add_vector<ct::tuple<TL> >::type& v) {
        typedef typename ct::type_at<scalar_tl, i>::type
          scalar_type;
        typedef typename ct::type_at<vector_tl, i>::type
          vector_type;
        // Multiply with kernel
        vector_type val =
          vvm::vector_cast<vector_type>(v.template get<i>()) *
          self.template vector_kernel<scalar_type>()[self._i];
        // Select largest
        self._vector_result.template get<i>() = vvm::select(
          val > self._vector_result.template get<i>(),
          val,
          self._vector_result.template get<i>());
      }
    };
  };
private:
  ct::tuple<scalar_tl> _scalar_result;
  ct::tuple<vector_tl> _vector_result;
```

```
  int _i;
};
```

Table 10.9 Speedup attained by executing VVIS's `max_filter` functor with a 3×3 `signed char` kernel in AltiVec mode over scalar mode

	Single-Channel Image				RGB Image			
	320 ×960	640 ×960	960 ×960	1280 ×960	320 ×960	640 ×960	960 ×960	1280 ×960
`unsigned char`	1.6	1.6	1.6	1.6	1.5	1.5	1.5	1.5
`unsigned short int`	0.7	0.7	0.7	0.7	0.6	0.6	0.6	0.6
`unsigned int`	0.6	0.6	0.6	0.6	0.4	0.4	0.5	0.4
`float`	0.4	0.4	0.4	0.4	0.3	0.3	0.3	0.3
`double`	0.3	0.3	0.3	0.3	0.3	0.3	0.3	0.3

Table 10.9 shows the speedup obtained by executing VVIS in AltiVec mode over scalar mode, when using the `max_filter` functor with a `sint8` kernel. Like `linear_filter`, `max_filter` was 1.6 times faster in AltiVec mode when operating on `unsigned char` images. When operating on other images that had `vvm::vectors` with more than one VPU vector, `max_filter`'s speedup was between 0.3 and 0.7.

10.4 Conclusion

This chapter discussed the algorithms and functors of the categories presented in Section 8.2. The categories are quantitative, transformative, and convolutive. For each category, scalar, AltiVec and VVIS generic algorithms, functor requirements, and some sample functors were presented.

When processing single-channel `unsigned char` storages, VVIS quantitative and transformative algorithms were comparable to hand-coded scalar and AltiVec programs in scalar and AltiVec mode respectively. When processing multi-channel images however, VVIS quantitative and transformative algorithms were much slower in scalar mode, but only a bit slower in AltiVec mode. VVIS convolutive algorithms were comparable to hand-coded scalar programs in scalar mode, when processing single-channel `unsigned char` images, but a bit slower when processing multi-channel images. VVIS convolutive algorithms were slower than the AltiVec hand-coded program when processing both single-channel and multi-channel images, because the AltiVec hand-coded program did not support variable kernel sizes and VVM's slower load unaligned function. VVIS had good scalar performance when operating on single-channel `unsigned char` images, because in scalar mode, VVIS treats all storages as unknown storages and thus avoids the use of vectorised algorithms.

For each functor discussed, the speedup attained by executing VVIS in AltiVec mode instead of scalar mode was presented. Because VVIS generic algorithms' performance were generally comparable to hand-coded scalar and AltiVec programs when operating on single-channel images, the speedup attained by moving to AltiVec is indicative of the speedup of AltiVec over scalar code. The speedup is larger for RGB images, because VVIS was much slower in scalar mode with RGB images. For the transformative operations tested, VVIS provides at most a four-fold speedup when processing single-channel `unsigned char` images. For the convolutive operations tested, VVIS provides a speedup of approximately 1.5 times when processing single-channel `unsigned char` images, despite using `signed short` internally.

This chapter provides some new insights into how to implement generic, vectorised algorithms: it is more efficient to have different algorithms for scalar and vector mode; it would have been more difficult to implement generic, vectorised algorithms without VVM; the use of typelists to represent channel types introduces complexity; and without the left edge, algorithms are easier to implement.

For efficiency, algorithms should have different implementations for scalar and vector mode. Without different implementations, the generic, vectorised library would be handicapped when there is no VPU available. In VVIS, this is solved by simply treating contiguous storages, which are processed using vectorised algorithms, as unknown storages, which are processed using scalar algorithms. When a VPU is available, VVIS is near optimal only if the VVM implementation has no significant overheads. Since optimising reduces the speedup of AltiVec over scalar code, at least with Apple GCC 3.1 20021003, having no significant overheads is important to the practicality of the abstract VPU. Without zero-cost, VVIS could actually run slower in AltiVec mode than in scalar mode. Examples of VVIS functors that run slower in AltiVec mode because of significant overheads when processing non-`char` storages in the abstract VPU are `linear_filter` and `max_filter`.

Programming a generic, vectorised library which uses the VPU directly would have been more difficult without VVM. Without the templated `vvm::vectors`, more specialised code for iterators, and functors would have been required. Without the constant scalar count provided by VVM, convolutions would have been more difficult to implement.

While the use of typelists to represent channel types provides great flexibility, it was more difficult to implement functors for such images. This was particularly evident during the implementation of the convolutive functors. The convolutive functors had to keep a copy of the kernel and sum of the values in the kernel for each different type in the typelist.

The VVIS algorithms created in this section show that without the left edge, the generic, vectorised algorithms were not overly difficult to implement. There is no need to handle misalignment between storages since without the left edge, all storages are aligned to each other. The convolutive algorithm created was much more involved than

the transformative and quantitative algorithms, because it had to calculate the number of `vvm::vectors` that need to be loaded and to load and shift them to obtain the correct `vvm::vectors`. In addition it also tried to reduce loads by caching the `vvm::vectors` that were previously loaded. If the left edge existed, convolute would have been much more difficult to implement.

Chapter 11

Examples

In this chapter, three VVIS examples are presented to provide more insight into how to use VVIS. The first example presented shows how to load two images, subtract one image from another, and write the result to a third image file. The second example shows how to capture an image from a camera attached to the computer, invert the image and then store the result to an image file. The last example discusses how to create a Cocoa application that applies a number of operations to a continuous feed from the camera.

11.1 Subtracting one image file from another

The program discussed in this section subtracts one image from another and writes the output to a third image. The program starts by checking the number of command-line arguments. After checking the command-line arguments, the program tries to open the two input files, exiting if any of the files fail to open. The program then checks if the sizes of the two input files are the same, aborting if they are different, reads the images as grey-scale images, performs a saturated subtraction, and writes the output to the image file which was specified by the third command-line argument. To change the sample program to process RGB images, simply change `vvis::image` to `vvis::rgb_image`.

```
#include <vvis/vvis.h>

int main(int argc, char *argv[]) {
  if(argc != 4) {
    std::cerr << argv[0] << " <input1> <input2> <output>"
              << std::endl;
    return 1;
  }
  // Open files
  vvis::fimporter in1(argv[1]);
```

```
  vvis::fimporter in2(argv[2]);
  if(!in1.good() || !in2.good()) {
    std::cerr << "Unable to open " << argv[1]
      << " or " << argv[2] << std::endl;
    return 1;
  }
  // Check if image sizes are the same
  if(in1.width() != in2.width() || in1.height() != in2.height()) {
    std::cerr << "Unable to process because image sizes are not "
      << "the same" << std::endl;
    return 1;
  }

  // Read in as grey scale images
  vvis::image<vvis::uint8> img1, img2;
  vvis::image<vvis::uint8> dest(in1.width(), in2.height());
  in1 >> img1;
  in2 >> img2;

  // Do saturated subtraction
  vvis::transform(
    img1, vvis::pixel_accessor<vvis::image<vvis::uint8> >,
    img2, vvis::pixel_accessor<vvis::image<vvis::uint8> >,
    dest, vvis::pixel_accessor<vvis::image<vvis::uint8> >,
    vvis::minus_saturated<>);

  // Write output
  vvis::fexporter out(argv[3]);
  if(!out.good()) {
    std::cerr << "Unable to open " << argv[2] << std::endl;
    return 1;
  }
  out << dest;

  return 0;
}
```

The program uses `fimporter` to load the images and `fexporter` to write the output to a file. The `img`, `img2`, and `dest` images all use the same storage type. Since the program does not explicitly state the storage type, VVIS selects the default storage type,

which is `illife_chunky_storage`. The `illife_chunky_storage` storage type uses a `std::vector` to hold a set of `contiguous_chunky_storages`. Since VVIS runs with automatic unknown storages optimisation, which was discussed in Section 10.1.1, enabled, `contiguous_chunky_storage` is treated as a contiguous storage only when a VPU, such as AltiVec, is available and as an unknown storage when VVIS runs in scalar mode. As a result, `transform` only uses VVM when AltiVec is enabled.

11.2 Inverting an image captured from a sequence grabber

The example discussed in this section captures an image from a sequence grabber, inverts the image, and then writes the inverted image to a file on disk. VVIS's `sgimporter` is used to capture images from the sequence grabber in synchronous mode. In synchronous mode, `sgimporter` will provide the image from the sequence grabber when requested. Synchronous mode was used because it makes the example easier to implement.

```
#include <vvis/vvis.h>

int main(int argc, char *argv[]) {
  if(argc != 2) {
    std::cerr << argv[0] << " <output>" << std::endl;
    return 1;
  }

  // Prepare sequence grabber to grab 640x480 images
  vvis::sgimporter<vvis::sync_capture>
    sgi(vvis::sgsettings(640, 480));
  if(!sgi.good()) {
    std::cerr << "Unable to connect to sequence grabber"
              << std::endl;
    return 1;
  }

  // Get an image
  typedef rgb_image<vvis:uint8> image_t;
  image_t img;
  sgi >> img;

  // Do invert
```

```
  vvis::transform(img, vvis::pixel_accessor<image_t>(),
    img, vvis::pixel_accessor<image_t>(),
    vvis::bind1st(vvis::minus<image_t::pixel_type>(),
      std::numeric_limits<vvis::uint8>::max()));

  // Write output
  vvis::fexporter out(argv[1]);
  if(!out.good()) {
    std::cerr << "Unable to open " << argv[1] << std::endl;
    return 1;
  }
  out << img;

  return 0;
}
```

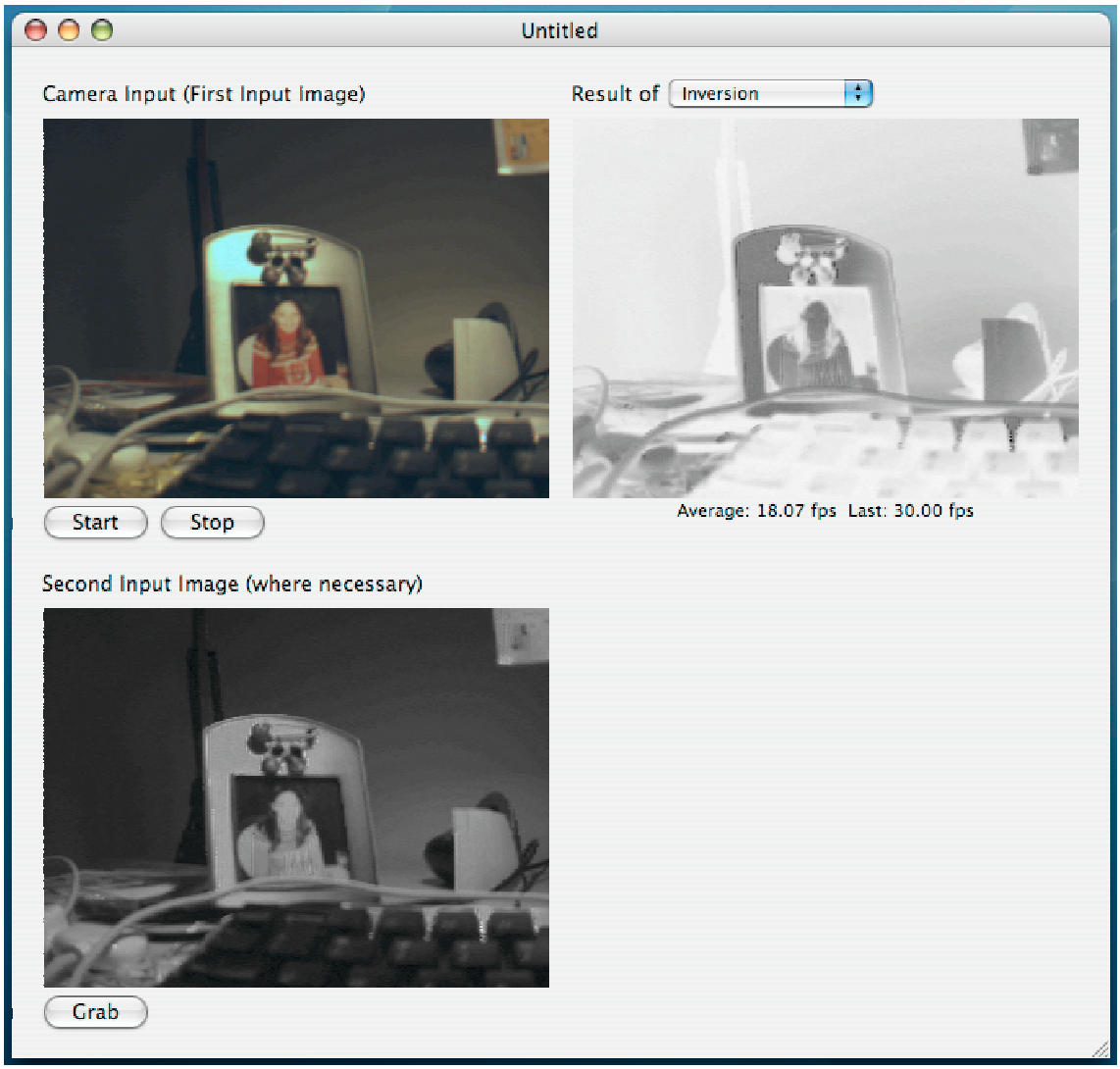
The example captured an RGB image. If `image_t` was defined as `image<vvis::uint8>` instead, then the example would capture a grey-scale image, without any further changes.

11.3 Grabbing continuously from a camera while outputting to the screen

Figure 11.1 shows the sample application that will be discussed in this section. This application was implemented using Objective-C++ and Cocoa. The application shows three images: camera input, a second image and the resulting image from an operation. The camera input image shows the image that the camera sees. Since the camera used captures colour images, the camera input image was in colour. The second image is used in operations that require two images, such as addition and subtraction. The second image is a grey-scale VVIS image. The resulting image shows the result of the operation selected. A frames-per-second label under the resulting image shows the average frame rate and the last frame rate. Start and Stop buttons start and stop camera capture respectively. The Grab button makes a copy of the camera input image to the second image. For brevity, the code is not presented here in its entirety. Instead, only important pieces will be discussed.

The two most important classes in this sample application are `CameraViewController` and `GWorldView`. `CameraViewController` responds to user input, while `GWorldView` is responsible for rendering GWorlds to the screen. There is one instance of `CameraViewController` for each window. Each of the three images shown in Figure 11.1 are `GWorldViews`.

Figure 11.1 Sample Cocoa application using VVIS to capture and process images from a camera



`CameraViewController` is a controller responsible for responding to user input. `CameraViewController`'s interface is shown below. Changing `image_t` to `vvis::rgb_image` instead of `vvis::image` will capture and process RGB images with any other changes.

```
typedef vvis::sgimporter<vvis::async_capture> sgimporter_t;
typedef vvis::image<vvis::uint8,
                    vvis::illife_planar_storage> image_t;
enum operation {
  noop = 0, invert, threshold, convolve,
  plus_op = 21, minus_op, multiply_op, divide_op,
    plus_saturated_op, minus_saturated_op, modulus_op,
    negate_op,
  equal_to_op = 41, not_equal_to_op, greater_op, less_op,
    greater_equal_op, less_equal_op,
  logical_and_op = 61, logical_or_op, logical_not_op,
  bitwise_and_op = 81, bitwise_or_op, bitwise_not_op
};
@interface CameraViewController : NSWindowController {
  sgimporter_t* seqGrabber;
  image_t* firstImage;
  image_t* secondImage;
  image_t* resultImage;
  GWorldPtr secondGWorld;
  GWorldPtr resultGWorld;

  int currentOp;
  NSLock* imageLock;

  IBOutlet GWorldView* cameraView;
  IBOutlet GWorldView* secondView;
  IBOutlet GWorldView* resultView;
  IBOutlet NSTextField* frameRate;

  long frameCount;
  float lastTime;
  float averageFramesPerSecond, currentFramesPerSecond;
}
- (void) fatalError:(NSNotification*)notification;
- (void) doNothing:(id)nothing;
- (void) updateFrameRate:(id)theTimer;
```

```
- (void) frameAvailable:(const TimeValue)time;
- (IBAction) changeOperation:(id)sender;
- (IBAction) startCapture:(id)sender;
- (IBAction) stopCapture:(id)sender;
- (IBAction) grabCapture:(id)sender;
@end
```

When the window is loaded, `CameraViewController` initialises the sequence grabber, VVIS images and GWorlds, detaches a `NSThread` to switch Cocoa to multi-threaded mode, and starts a timer to refresh the frames-per-second label twice a second. On failing to initialise the sequence grabber, `CameraViewController` posts a notification to display an error message and close the window later. The notification is required because closing the window in `windowDidLoad` does not work. A `NSThread` has to be detached to switch Cocoa to multi-threaded mode, because the sequence grabber is capturing asynchronously. The thread detached does nothing. A timer was required for updating the frames-per-second label to reduce the frequency at which the display is refreshed. Because VVIS connects to the first available sequence grabber by default, the sample application can connect to more than one camera simultaneously; each window will be connected to a different camera.

```
- (void) windowDidLoad {
  // Setup a notification for closing the window
  [[NSNotificationCenter defaultCenter] addObserver:self
    selector:@selector(fatalError:)
    name:@"fatalError" object:nil];

  // Detach a NSThread to switch Cocoa to Multi Threaded mode
  [NSThread detachNewThreadSelector:@selector(doNothing:)
    toTarget:self withObject:nil];

  // Initialise image Lock
  imageLock = [[NSLock alloc] init];

  // Set current operation to none
  currentOp = noop;
  // Initialize Sequence Grabber
  NSRect rect = [cameraView bounds];
  seqGrabber = new sgimporter_t(sgsettings(rect.size.width,
                                           rect.size.height));
  if(!(seqGrabber->good())) {
```

```
  [[NSNotificationQueue defaultQueue]
    enqueueNotification:[NSNotification
      notificationWithName:@"fatalError"
      object:self
      userInfo:[NSDictionary dictionaryWithObjectsAndKeys:
        @"Sequence Grabber could not be initialized",
        @"Message", nil]]
    postingStyle:NSPostASAP];
}

// Create images
firstImage = new image_t(rect.size.width, rect.size.height);
secondImage = new image_t(rect.size.width, rect.size.height);
resultImage = new image_t(rect.size.width, rect.size.height);

// Create new GWorld for second and result
// No need for first because first is showing straight
// camera output
Rect r;
SetRect(&r, 0, 0, rect.size.width, rect.size.height);
if(QTNewGWorld(&secondGWorld, k32ARGBPixelFormat, &r, 0, NULL, 0)
   != noErr) {
  [[NSNotificationQueue defaultQueue]
    enqueueNotification:[NSNotification
      notificationWithName:@"fatalError"
      object:self
      userInfo:[NSDictionary dictionaryWithObjectsAndKeys:
        @"Unable to create GWorld",
        @"Message", nil]]
    postingStyle:NSPostASAP];
}
if(QTNewGWorld(&resultGWorld, k32ARGBPixelFormat, &r, 0, NULL, 0)
   != noErr) {
  [[NSNotificationQueue defaultQueue]
    enqueueNotification:[NSNotification
      notificationWithName:@"fatalError"
      object:self
      userInfo:[NSDictionary dictionaryWithObjectsAndKeys:
        @"Unable to create GWorld",
```

```
            @"Message", nil]]
        postingStyle:NSPostASAP];
  }
  // Start Frames Per Second updater
  averageFramesPerSecond = 0;
  currentFramesPerSecond = 0;
  [NSTimer scheduledTimerWithTimeInterval:0.5 target:self
    selector:@selector(updateFrameRate:)
    userInfo:NULL repeats:YES];
}
```

On fatal errors, the `fatalError` function is called by the notification system. The `fatalError` function closes the window and removes itself from the notification queue, if the notification was from itself. It needs to check that the notification was from itself because all `CameraViewController` instances are informed when there is a `fatalError`.

```
- (void) fatalError:(NSNotification*)notification {
  if([notification object] == self) {
    NSRunCriticalAlertPanel(@"Error",
      [[notification userInfo] objectForKey:@"Message"],
      @"Close Window", NULL, NULL);
    [self close];
    // Since we are closing, we don't want anymore notifications
    [[NSNotificationCenter defaultCenter] removeObserver:self];
  }
}
```

When the window is closed, we stop the capture process, if any, and tell the document to close as well. The sequence grabber is closed so that it releases the sequence grabber that it is connected to.

```
- (void) windowWillClose:(NSNotification *)aNotification {
  [self stopCapture:self];
  seqGrabber->close();
}
```

The `updateFrameRate` function is called twice every second. It displays the last frames per second value that was calculated.

```
- (void) updateFrameRate:(id)theTimer {
  [frameRate setStringValue:
```

```
    [NSString stringWithFormat:
      @"Average: %.2f fps  Last: %.2f fps",
      averageFramesPerSecond, currentFramesPerSecond]];
}
```

When the user changes the operation, `CameraViewController`'s `changeOperation` method is called. Each operation's tag value is set to the appropriate value in the enumeration. So the new operation can be obtained by from the combo box's selected item's tag.

```
- (IBAction) changeOperation:(id)sender {
  currentOp = [[sender selectedItem] tag];
}
```

When the user clicks the Start button, `CameraViewController`'s `startCapture` method is called. This method tells the sequence grabber to start capturing and to call the `frame_available` function when a frame is ready. In addition, it gives the sequence grabber a pointer to itself to pass to the `frame_available` function. Note that the `frame_available` function is not the `frameAvailable` function defined in `CameraView`, but a normal C function.

```
- (IBAction) startCapture:(id)sender {
  seqGrabber->start(frame_available, self);
}
```

The `frame_available` function simply uses the pointer to `CameraViewController` passed to it through the sequence grabber to call `CameraViewController`'s `frameAvailable`.

```
void frame_available(sgimporter<async_capture>& sgi,
                     const TimeValue time, void* data) {
  CameraViewController* controller = (CameraViewController*)data;
  [controller frameAvailable];
}
```

`CameraViewController`'s `frameAvailable` gets the frame available from the sequence grabber, storing it in the VVIS image `firstImage`. Then, depending on the value of `currentOp`, different operations are performed to `firstImage`. If an operation requiring two input images was selected then `secondImage` is also used. The output is stored in `resultImage`. The VVIS images `secondImage` and `resultImage` are converted to `secondGWorld` and `resultGWorld` respectively using `vvis::draw_image`. The `GWorldViews` are then used to display the `secondGWorld` and `resultGWorld`. The sequence grabber's GWorld is displayed in the camera input `GWorldView`. The `frameAvailable` function's implementation is as follows:

```
- (void) frameAvailable {
  [imageLock lock];
  // Get the image
  (*seqGrabber) >> *firstImage;
  // Process Image
  switch(currentOp) {
  case noop:
    *resultImage = *firstImage;
    break;
  case invert:
    vvis::transform(*firstImage, vvis::pixel_accessor<image_t>(),
      *resultImage, vvis::pixel_accessor<image_t>(),
      vvis::bind1st(vvis::minus_saturated<image_t::pixel_type>(),
        std::numeric_limits<vvis::uint8>::max()));
    break;
  case threshold:
    vvis::transform(*firstImage, vvis::pixel_accessor<image_t>(),
      *resultImage, vvis::pixel_accessor<image_t>(),
      vvis::bind2nd(vvis::greater<image_t::pixel_type>(),
        std::numeric_limits<vvis::uint8>::max() / 2));
    break;
  // ... other operations
  }
  [cameraView displayGWorld:seqGrabber->gworld()];
  // Convert to GWorld for display
  draw_image(secondGWorld, *secondImage);
  draw_image(resultGWorld, *resultImage);
  // Show result
  [secondView displayGWorld:secondGWorld];
  [resultView displayGWorld:resultGWorld];
  [imageLock unlock];
}
```

When the user clicks the Stop button, `CameraViewController`'s `stopCapture` method is called. This method tells the sequence grabber to stop capturing.

```
- (IBAction) stopCapture:(id)sender {
  seqGrabber->stop();
}
```

Clicking on the Grab button calls `CameraViewController`'s `grabCapture`. This method locks `imageLock`, copies the `firstImage` to the `secondImage`, and then releases `imageLock`. The `imageLock` mutex is required to prevent `frameAvailable` from using `secondImage` while it is being copied.

```
- (IBAction) grabCapture:(id)sender {
  [imageLock lock];
  *secondImage = *firstImage;
  [imageLock unlock];
}
```

`GWorldView` was created to render QuickDraw GWorlds to a Cocoa view. Because VVIS is based on QuickTime, and QuickTime and Cocoa use different 2D display libraries (QuickTime uses QuickDraw, while Cocoa uses Quartz), VVIS cannot produce Quartz images directly. `GWorldView` inherits from `NSQuickDrawView`. Since `NSQuickDrawView` provides a GWorld that represents the screen, `GWorldView` only needs to render an off-screen GWorld to an on-screen GWorld. While it was was actually easier to use `Draw32BitARGBToContext` (App 2003*b*) to render a GWorld to a Quartz Graphics Context, `GWorldView` uses sample code from (App 2003*a*) to decompress an off-screen GWorld to an on-screen GWorld.

11.4 Conclusion

Three different VVIS examples were presented in this chapter: subtracting one image from another, inverting an image captured from a sequence grabber and a graphical Cocoa application that grabs continuously while displaying the output to the screen.

The examples show that it is easy to use VVIS to read and write image files, and to capture frames from a sequence grabber. In addition, changing a `vvis::image` to `vvis::rgb_image` is usually all that is needed to change a grey-scale program to a RGB program. VVIS is usable in Objective-C++ and with Cocoa. Rendering to the Cocoa application's window was difficult because Cocoa and QuickTime use different 2D display libraries. VVIS only provided functions to convert a VVIS image to an off-screen GWorld; on-screen GWorlds are not supported either. VVIS might have been easier to use with Cocoa if VVIS provided routines for rendering to Quartz.

Chapter 12

Conclusions

This thesis investigated the integration of generic programming with vector processing.

Integration of generic programming with vector processing: The integration of generic programming with vector processing resulted in a generic, vectorised, machine-vision library, called Vectorised Vision (VVIS). The existence of VVIS itself shows that generic programming with vector processing is feasible.

Divisions of duties used by other generic libraries, like STL and VIGRA, could not be used in a generic, vectorised library, because the iterator hides from the algorithm information about how pixels are arranged in memory. The storage concept was introduced to provide this information to algorithms. This information is used to determine whether to process the pixels using the scalar processor or the VPU. The information is also used to load vectors efficiently into the VPU. VVIS has different iterators and algorithms for different storages.

For a generic, vectorised library to exist, there must be some way of writing generic, vector programs. The abstract VPU was proposed to address this issue. Since vector algorithms are generally more difficult to implement efficiently, a categorisation scheme based on input-to-output correlation was used to reduce the number of algorithms that need to be implemented. This categorisation scheme allows VVIS to provide only four algorithms to cover a wide range of image-processing operations.

Other problems partially addressed as a by-product are the poor integration of image capture with machine-vision libraries and the non-portability of vectorised, machine-vision algorithms.

Poor image capture integration with machine-vision libraries: VVIS provides an easy-to-use syntax for capturing images from a camera, reading images from files, and writing images to files. This ease-of-use was illustrated in Chapter 11. Unfortunately, because the captured image's format was unsuitable for vector programs,

some conversion costs were present. Conversion costs were discussed in Section 9.4.

Non-portability of vectorised machine-vision algorithms: The abstract VPU was proposed to allow VVIS a mechanism for expressing generic vector programs. Since generic vector programs are expected to be independent of the real VPU and thus ideally executable on many real VPUs, the abstract VPU also allows for portable vector programs. Unfortunately, the performance results obtained do not make it suitable for all applications. However, because the primary aim of the abstract VPU used in this thesis, Virtual Vector Machine (VVM), was to allow for generic vector programs, the performance problems were not investigated further.

This chapter starts with a summary of what is new, followed by some possible future directions.

12.1 Summary of contributions

The main contribution from this thesis is the design of a generic, vectorised, machine-vision library.

Generic, vectorised, machine-vision library: The generic, vectorised, machine-vision library developed as part of this thesis is called Vectorised Vision (VVIS). VVIS provides functions to facilitate image acquisition, processing, analysis and output. Images can be either loaded from disk or streamed from a sequence grabber. Image processing and analysis is where VVIS uses the VPU. For output, VVIS is able to write to files and to convert images to QuickTime GWorlds for display.

Instead of using a real VPU, VVIS uses VVM, an abstract VPU. Using an abstract VPU like VVM allows VVIS to express its generic algorithms using VPU constructs. The abstract VPU also provides cross-platformability, and allows VVIS to support more vector-processor technologies more easily. A port requires only an additional vector-processor implementation for the abstract VPU. VVM's constant scalar count, templated vectors and traits allows VVIS to provide generic, vectorised algorithms. VVM also provides an easy-to-use interface, and consistent functions between scalars and vectors where possible. The easy-to-use interface makes coding VVIS and user-defined functors easier. Using an abstract VPU instead of a real VPU however introduces the possibility of speed degradation. The VVM implementation used by VVIS in this thesis is within 1% slower than a hand-coded program for `char` types when AltiVec is enabled, and when there is no VPU. This implementation was chosen because in machine vision, most images are either 8-bit or use 8-bit channels.

QuickTime is used to read images from sequence grabbers, and to read and write from and to files. Using QuickTime allows VVIS to easily acquire images from sequence grabbers, like cameras, and import and export image files. VVIS leverages QuickTime's ability to read and write a wide range of image file formats. QuickTime was chosen because the target platform is a Macintosh. Since QuickTime is also available on Windows, it should be possible for the input/output interface to operate on both platforms.

Two different divisions of duties were proposed and evaluated. The first division of duty gives all responsibility for handling VPU issues to the algorithm, while the second division of duty reduces the responsibilities of the algorithm by having storages ensure that the data are aligned on the left side. The second division of duty was chosen for VVIS because it provides good performance as long as there is no need to copy data around. Furthermore, it simplifies the creation of algorithms greatly, since it removes the need to be concerned about misalignment, not only with memory, but also between storages.

Facets of VVIS's design that enable the use of the VPU are the addition of the storage concept, different iterators and algorithms for different storages, and the support for both scalars and `vvm::vectors` by accessors and functors. The storage concept provides information about the relative positions of pixels in memory. This information is useful in determining when the data can be loaded efficiently into the VPU. Each storage type has different iterator requirements and is processed differently by the algorithm. Accessors and functors that support both scalars and `vvm::vectors` are required, because it is inefficient to use the VPU in all situations. The scalar version is required for processing edges and in situations where the storage format is not suitable for vector processing, namely when processing unknown storages.

When processing single-channel `unsigned char` storages, VVIS quantitative and transformative algorithms were comparable to hand-coded scalar and AltiVec programs in scalar and AltiVec mode respectively. When processing multi-channel images however, VVIS quantitative and transformative algorithms were much slower in scalar mode, but only a bit slower in AltiVec mode. VVIS convolutive algorithms were comparable to hand-coded scalar programs in scalar mode, when processing single-channel `unsigned char` images, but a bit slower when processing multi-channel images. VVIS convolutive algorithms were slower than the AltiVec hand-coded program when processing both single-channel and multi-channel images, because the AltiVec hand-coded program does not support variable kernel sizes and VVM's slower load unaligned function. VVIS's good scalar performance is because in scalar mode, VVIS treats all storages as unknown storages and thus

avoids the use of vector algorithms.

For the transformative operations tested, VVIS provides at most a four-fold speedup depending on the operation when processing single-channel `unsigned char` images. For the convolutive operations tested, VVIS provides a speedup of approximately 1.5 times when processing single-channel `unsigned char` images, despite using `signed short` internally.

Other significant, but less important contributions from this thesis are a categorisation scheme based on input-to-output correlation, and the abstract vector processor.

Categorisation based on input-to-output correlation: A categorisation based on input-to-output correlation was proposed for this thesis. This categorisation maps readily to generic programming, and provides insights into how operations might be implemented. For example, requiring only one input element per input set suggests that only a one-dimensional iteration is needed. This categorisation was initially developed to help reduce the number of algorithms required by a generic, vectorised machine-vision library. From this categorisation, it is clear that the algorithm's duty is only to coordinate the loading (and writing) of data. The writing of data can be delegated to accessors and functors.

Three categories were defined for use with the generic, vectorised, machine-vision library. Only two categories would have been required if the output responsibilities were removed from the algorithm. Output responsibilities were retained by the algorithms to be more consistent with other currently available generic libraries. The three categories are named quantitative, transformative and convolutive.

Abstract vector processor: The abstract VPU was proposed to allow for the creation of generic vector programs. The abstract VPU represents a family of real VPUs with a virtual VPU that has an idealised instruction set, and constraints common to the real VPUs being represented.

The idealised instruction set makes the abstract VPU easier to use and portable. It also removes the need for programmers to handle instruction availability, and allows the programmer to express his logic using ideal VPU constructs, free from any real VPU's inadequacies. Instructions should be provided for all types. Low-level instructions, such as prefetching memory, should also be available. An idealised instruction set implies a scalar implementation for all functions. This scalar implementation is referred to as the emulation layer. Vector-processor implementations provide mappings for abstract VPU functions to real VPU instructions where applicable. An abstract VPU should enable appropriate vector-processor implementations when real VPUs are available. Since a vector-processor implementation is

unlikely to provide mappings for all functions, expressions involving the use of both the emulation layer and the vector-processor implementation must be supported.

The abstract VPU should have the common constraints of the real VPUs that it represents. These constraints ensure that abstract VPU programs are comparable performance-wise to the real VPU programs. While ambiguity increases the number of real VPUs that can be represented, ambiguity can make zero-cost abstract VPU implementations more difficult. Examples of constraints common to desktop VPUs are short vectors, fixed vector sizes and efficient memory access only at aligned memory addresses. An abstract VPU should not try to represent a real VPU that has incompatible constraints because the resultant program is likely to be significantly slower than using the real VPU directly. For example, an abstract VPU with fixed vector sizes should not try to represent VPUs with variable vector sizes. VPUs, which support variable vector sizes, typically have larger start and stop overheads.

Apart from proposing the abstract VPU, an abstract VPU specification, called Virtual Vector Machine (VVM) was developed to support the creation of a generic, vectorised, machine-vision library. VVM aims to represent desktop VPUs, such as AltiVec, MMX and 3DNow!. It has three constraints common to desktop VPUs: short vector, fixed vector sizes, and fast access only to aligned memory addresses. While the size of a `vvm::vector` is fixed, unlike desktop VPUs, all `vvm::vectors` contain the same number of scalars regardless of type. This constant scalar count is important in template programming, which is required for the creation of a generic, vectorised library. Other features that support the creation of a generic, vectorised library include templated `vvm::vectors`, traits, and consistent functions for both scalar and `vvm::vector` operations. Because of the high cost of converting types in vector programs, VVM does not provide automatic type conversion; it only supports explicit type conversions.

The implementability of VVM using only Standard C++ was investigated. While VVM can be implemented using only Standard C++, there were some performance issues. The cost of the abstract VPU depends on how it is implemented, and the compiler used. For an abstract VPU's operation to be zero-cost, there should be no additional overheads when executing the appropriate instruction and when switching between the emulation layer and the active vector-processor implementation. With Apple's GCC 3.1 20021003, a VVM implementation based on function overloading and template switching was within 1% slower than a hand-coded program when the number of elements in a `vvm::vector` was one. For `vvm::vectors` with two and four elements, this implementation was within 11% to 20% and within 14% to 24% slower than a hand-coded program respectively. A VVM implemen-

tation that used expression templates based on Veldhuizen (1995*b*) and function switching was faster than function overloading when the number of elements in a `vvm::vector` was two for expressions with less than three additions and when the number of elements in a `vvm::vector` was four for expressions with less than four additions. When there is only a single element in a `vvm::vector`, expression templates were at most about 11 times slower. For `vvm::vectors` with two VPU vectors, the cost increased from within 1% to within 44% when the number of additions in the expression increased from one to five. For `vvm::vectors` with four VPU vectors, the cost was within 0% to 22%. Switching between the emulation layer and the active vector-processor implementation did not add any additional overheads if implemented correctly.

Since most image-processing operations operate on `char` types, a VVM implementation based on function overloading would be within 1% slower than a hand-coded program for most image processing operations. It will be within 1% slower than a hand-coded program for all types in scalar mode and for `char` types in AltiVec mode.

In summary, this thesis has demonstrated that generic, vectorised, machine-vision libraries are possible, through the addition of the storage concept, and the abstract VPU. Without the storage concept, there is no way to decide when to use or not use the VPU. Without the abstract VPU, algorithms and functors would not be generic, instead requiring specialised implementations for all the types supported.

12.2 Future directions

Suggested future work includes handling `const` and better performance in VVM, faster multi-channel images, unaligned load and store regions, better QuickTime support, and converting between Illife and unknown storages in VVIS.

1. Handling `const` in VVM

   The VVM specification handles `const` poorly. Support for `const` was added at a later stage, during the implementation of VVIS's `const_iterators`. While VVIS's `const_iterators` should provide only `const vvm::vectors`, providing `const vvm::vectors` was difficult because of the manner in which VVM was implemented. Consider the following:

```
template<typename scalarT>
vector<scalarT>
operator+(const vector<scalarT>& a,
```

```
        const vector<scalarT>& b) {
  vvm::vector<scalarT> ret;
  ret.vpvector(0) = a.vpvector(0) + b.vpvector(0);
  return ret;
}
```

If `ret` was a `const` object, then the addition operation on `vpvector(0)` would be illegal. Switching to using an unnamed return value gets around this problem but had worse performance with the compiler, Apple GCC 3.1 20021003, used. Because better performance was decided to be more important than `const`, `const_iterator` and `iterator` are currently identical in VVIS.

2. Better VVM performance

   The author believes that faster VVM implementations are still be possible. Two other techniques are proposed: combining expression templates with function overloading, and using metaobject protocols to implement VVM as a preprocessor. While using metaobject protocols to create a preprocessor is likely to provide more consistent performance, it will be less user-friendly than combining expression templates with function overloading because of the extra steps required during compilation. Combining expression templates with function overloading still only requires Standard C++.

   (a) Combining expression templates and function overloading

   Since the performance of expression templates appears to be inversely proportional to the performance of function overloading, combining expression templates with function overloading should provide better performance. Results from Section 7.7 however suggests that even though combining the two techniques should produce better performance, the overheads will not always be insignificant.

   Combining expression templates and function overloading means that we process the user's request using either function overloading or expression templates, based on the number of elements in a `vvm::vector`. Function overloading will be used to handle `vvm::vector`s with low element counts, while `vvm::vector`s with higher element counts will be handled using expression templates. A first attempt at combining expression templates and function overloading might produce code similar to the following:

```
template<typename scalarT> inline const vector<scalarT>
operator+(const vector<scalarT>& a,
```

```
          const vector<scalarT>& b) {
  // Use template metaprogramming
  // IF vector_traits<vector<scalarT> >::vpvector_count
  //    > threshold
  // THEN
  //   call expression template version
  // ELSE
  //   call simple operator overloading version
  // ENDIF
}
```

The code above fails because the return type of expression templates and function overloading is different. Adding the return type as a template parameter is unhelpful, because the user would need to specify the return type explicitly when it is called.

The `vvm::vector` can instead be augmented with information about its implementation. A template parameter can be added to `vvm::vectors` to determine which implementation it will use. The following code illustrates this idea:

```
template<
  typename scalarT,
  int implementation =
    _get_default_implementation<scalarT>::value>
struct vector {
  // ...
};
enum { _simple_op_overload, _expr_templates };
template<typename scalarT> _e<...>
operator+(const vector<scalarT, _expr_templates>& a,
          const vector<scalarT, _expr_templates>& b) {
}
template<typename scalarT>
const vector<scalarT, _simple_op_overload>
operator+(const vector<scalarT, _simple_op_overload>& a,
         const vector<scalarT, _simple_op_overload>& b) {
}
```

In this technique, each `vvm::vector` type is either processed using expression templates or function overloading, but never both. Since VVM does not allow expressions with different `vvm::vector` types, there is no need to handle cases where both expression templates and function overloading is used.

For expression involving different `vvm::vector` types, the programmer must explicitly cast one of the `vvm::vectors` types away.

(b) Metaobject protocol

A metaobject protocol is "an object-oriented interface for programmers to customise the behaviour [sic] and implementation of programming languages and other system software" (Chiba 1995). Metaobject protocols are used to provide meta-programming capabilities to a language. Some metaobject implementations rely on a runtime engine to perform their magic. Other implementations, like Open C++ (Chiba 1995, 1998*a*,*b*), are able to perform most of its meta-programming at compile-time instead of at run-time. Some special features in Open C++ however require the use of a runtime component.

Since any additional runtime components would introduce overheads, and thereby increase the overheads of the abstract VPU implementation, runtime components should be avoided. Since most features of Open C++ do not have a runtime component, it might be possible to produce a zero-cost abstract VPU. At the time of writing though, Open C++ does not support templates, and thus would be unsuitable for implementing VVM.

3. Faster multi-channel images in scalar mode

In scalar mode, while VVIS is comparable to hand-coded scalar programs when operating on single-channel images, it is much slower than hand-coded scalar programs when operating on multi-channel images. More work needs to be undertaken to ascertain how to increase VVIS's performance in scalar mode with multi-channel images.

4. Regions that perform unaligned loads and stores

In Section 9.2, regions that produced aligned data by copying were profiled. Tables 9.2 and 9.3 show that a significant number of operations would be required before the time saved by using an aligned algorithm would offset the cost of copying data. Thus, a region that performed unaligned loads and stores should be profiled. Using such a region should have performance similar to Unaligned Load and Unaligned Store from Section 3.7.

5. Improve QuickTime support in VVIS

While QuickTime supports planar images, the author was unable to get QuickTime to capture to planar images directly. More work is needed to investigate how to get QuickTime to capture images directly to planar images, and to investigate the costs involved. Such changes should reduce the cost of communicating between QuickTime and VVIS, and hopefully lead to faster imports, exports and image capture.

A storage class that provides access to data in an off-screen GWorld directly can be added to VVIS. Such a storage would either always be treated as unknown storages or VVIS needs to support runtime storage determination, since the data format of GWorlds is determined at run-time.

6. Converting Illife storages to unknown storages in VVIS

   Because Illife and unknown storage interfaces are currently not interchangeable, an operation involving an Illife storage and a contiguous or unknown storage will fail to compile. Because the Illife storages and unknown storages reuse the same functions, it is not possible to implement a single storage as both an unknown and an Illife storage.

   One solution is to change the interfaces so that they are no longer mutually exclusive. A contiguous or unknown storage would then support the Illife storage (of unknown storages) interface. Another solution would be to introduce regions that changed the storage type without requiring expensive copying.

7. Examining the assembly code generated by the compiler, and the compiler's source code

   Instead of relying only on actual program execution time, the assembly code generated by the compiler can be investigated. Checking the assembly code generated might provide more insight into the behaviour of the different programs. In addition, the assembly code generated can be used to verify that a VVM implementation is zero-cost.

   Examining the assembly code generated could help verify if the slower than expected performance of programs compiled with Apple GCC 3.3 20030304 in Chapter 7 was because of the failure to inline functions. Since the source code of all the compilers used is available, an even more direct method of investigating whether the compiler is inlining a function is to read the compiler's source code. Whereas the assembly code generated might only indicate whether a function is inlined or not, the compiler's source code would provide clues on why a function was not being inlined.

8. Processing images with different spatial resolutions

   This thesis did not consider the processing of images with different spatial resolutions. All solutions discussed in this thesis assume that all images have the same spatial resolution. To process images with different spatial resolutions using VVIS, the user needs to first change all images involved in the operation to the same spatial resolution.

9. Implementing `fimporter` and `fexporter` as streams

The VVIS classes `fimporter` and `fexporter` would be better implemented as streams. This would allow images to be loaded easily from places other than disk. This was not investigated further since this thesis is focused on how to process images generically using the VPU, instead of how to load and store images efficiently. Both classes were not implemented as streams because it seemed easier to not do so.

10. Processing incomplete images

    This thesis only considered cases where images were completely loaded in memory before use. As problems become larger, this method becomes more wasteful.

    It is expected that images that have not completely arrived could be hidden from algorithms by the iterator. For example, the iterator could pause the current thread when the algorithm iterates to a position that has not been loaded yet. Since quantitative and transformative operations can be processed out of order, as long as the input is correlated with the output, it would have been possible for the iterator to iterate to the next available pixel instead of the pixel in the next row or column. A solution that aims to provide such features would need to consider how iterators would correlate inputs and outputs, since there is no mechanism specified for this in the current design.

11. Processing video streams with consideration to time

    While VVIS provides access to sequence grabbers, allowing applications to easily processing real-time input, VVIS processes these images without any regard for time. The real-time feed is treated simply as a sequence of images. VVIS makes no guarantees regarding when each image in the sequence will be available. The time information provided via QuickTime was not used. For proper video support, time needs to be taken into consideration.

Appendix A

Glossary

abstract vector processor: A virtual vector processor that represents a family of vector processors. See Chapter 5 for more information.

aligned: Vector processors are not able to access all locations of memory directly. The locations that they can access directly are known as aligned. In AltiVec, an aligned memory address is always exactly divisible by 16. See Chapter 3 for more information on alignment.

AltiVec mode: In this thesis, it refers to a program being compiled with AltiVec instructions on. In GCC, the appropriate switch is `-faltivec`. A program that runs in AltiVec mode will use AltiVec instructions, if coded to use them.

argument: This refers to a value or variable that is passed in, when making a function call.

Carbon: An Apple C API, which was originally designed to be a migration path for developers switching from Mac OS 9 to Mac OS X. Carbon's services cover everything from file management to graphical interfaces.

Cocoa: An object-oriented interface for creating Mac OS X applications. Cocoa comes from NeXTSTEP. It is only usable in Objective-C (or Objective-C++) and Java.

chunky: This refers to a storage scheme for multi-channel images. It is also known as interleaved. A chunky RGB image would have its pixels stored as RGBRGBRGB and so on. In this thesis, chunky is also used to refer to RRRRGGGGBBBBRRRGGGGBBBB and so on, where the number of times a component is repeated is equal to the number of scalars in a corresponding vector. In a sense, we are interleaving vectors instead of scalars. Chunky scalar and chunky vector are used to refer to interleaving scalar and vectors respectively.

convolutive operations: Convolutive operations refer to a category of operations defined in this thesis. They accept a rectangle of elements per input set to produce a single output element for a single output set. See Chapter 8 for more details.

data streaming instructions: In AltiVec, data streaming instructions are responsible for prefetching data from memory into the caches. See also prefetch.

emulation layer: Part of an abstract vector processor that provides a scalar implementation for all operations. The emulation layer must also operate correctly when the type being operated on has native vector processor support. See Chapter 5 for more information.

enregister: A value that can be enregistered can be kept in a register inside the processor.

GWorld: Graphics World. QuickDraw represents images using this structure. A GWorld could be on screen or off screen and can have many different pixel formats.

integer type: This refers to types that represent whole numbers. In C++, they can be easily identified using `std::numeric_limits`. The type T is an integer type if `std::numeric_limits<T>::is_integer` is `true`. Examples of integer types include `char`, `short`, `int` and `long`. Boolean types are also considered integers.

naturally aligned: Data are naturally aligned if it is aligned appropriately for its size. The memory location of a naturally aligned datum is exactly divisible by the datum's size in bytes. For example, a `short` located at position `0x101` is not naturally aligned, while `0x102` is, assuming `short`s are two bytes long.

pair: This refers to Standard C++'s `std::pair` which is a tuple with two elements.

planar: Refers to a storage scheme for multi-channel images where each component is stored in a different location in memory. A planar RGB image, for example, would have its pixels stored in three arrays (RRRRRR, GGGGGG, BBBBBB) each containing a different channel.

prefetch: This refers to the process of loading data from memory into the caches before the processor has actually asked for it.

QuickDraw: A legacy two-dimensional drawing engine in Mac OS. QuickTime is based on QuickDraw.

quantitative operations: Quantitative operations refer to a category of operations defined in this thesis. They have one input element per input set, zero or more output elements per output set and a single output set. See Chapter 8 for more details.

Quartz: The native two-dimensional drawing engine in Mac OS X. Cocoa is based on Quartz.

scalar: A scalar is a single value. It can be a number, string or even a struct. The most important thing about a scalar is that the entire value represents a single entity. Scalar operations' speeds should largely be unaffected by the presence of the vector processor. Scalar operations are executed by scalar units which may or may not affect the vector processor. In MMX technology, the vector processor unit is actually the floating-point unit. In AltiVec, the vector processor unit is a separate unit and is capable of executing in parallel with the other scalar units. This means that in AltiVec, it is possible to perform a scalar operations for "free" when using the vector processor.

scalar processor: In this thesis, it refers to a processor (or part of a processor) that computes values one scalar at a time.

scalar mode: In this thesis, it refers to a program that has been compiled without any vector processor support. The resultant program will never use the vector processor.

template metafunctions: A template type that returns either a type or a constant value. It is used in template metaprogramming.

template metaprogramming: This refers to using C++ templates and other features to get the compiler to act as a interpreter. Using template metaprogramming in C++, the program can make decisions about itself while it is being compiled.

transformative operations: Transformative operations refer to a category of operations defined in this thesis. They accept one input element per input set. See Chapter 8 for more details.

tuple: A tuple is a fixed-size collection of elements. See Section 4.3 for more information.

unaligned: Vector processor typically can only load data directly from certain locations in memory. Memory locations which it cannot load directly are known as unaligned. See also "aligned".

undefined behaviour: The behaviour is not specified by VVM. Thus different VVM implementations might have different behaviours.

uniform vectors: This refers to a vector where all the values have the same meaning. For example, a vector containing RGB is not uniform, while a vector that contains only R is uniform.

vector: In this thesis, a vector is a fixed set of scalars. However, the word vector by itself has many different connotations. In the Standard C++ library, for example, the word "vector" refers to a re-sizable container that allows elements to be directly accessed. In vector processing, the word vector refers to a finite set of scalars. In AltiVec, vectors are called vectors. In MMX and its derivatives however, vectors are called packed data types.

vector processor: A vector processor is a microprocessor that has instructions that work on a set of scalars at a time. In virtually all desktop microprocessors, the vector processor is not a standalone microprocessor. Instead it is usually part of the microprocessor, and in some cases, it is actually one of the scalar units. In MMX technology for example, the vector processor unit is actually the floating-point unit. Unless otherwise mentioned, the term vector processor in this thesis only refers to vector processors on the desktop.

vector-processor implementation: A vector-processor implementation is part of a VVM implementation that implements VVM for a specific vector processor.

vector technology: The term vector technology refers to the method by which a VPU is integrated and the instruction set supported. A vector technology can be implemented in more than one microprocessor or family of microprocessors. For example, the AltiVec vector technology is implemented in Motorola's PowerPC G4 and IBM's Power4 microprocessors.

Appendix B

VVM's AltiVec Vector-Processor Implementation

The AltiVec vector-processor implementation of the VVM implementation developed for this thesis is discussed in this appendix. This appendix describes how VVM types are mapped to AltiVec types, how VVM instructions are mapped to AltiVec instructions, and how VVM type conversions are mapped to AltiVec instructions.

B.1 Type mappings

AltiVec vector mappings are discussed in this section. As discussed in Section 7.2.1, VVM maps scalars to VPU vectors based on the characteristics of the scalars. Table B.1 summarises the characteristics of AltiVec vectors that map to scalars in the VVM implementation. AltiVec vector types, `__vector signed long`, `__vector unsigned long` and `__vector pixel`, are not used by VVM and are thus not shown in the table. The AltiVec vector types `__vector signed long` and `__vector unsigned long` are omitted because they are identical to `__vector signed int` and `__vector unsigned int` respectively. In fact, `__vector signed long` and `__vector unsigned long` are deprecated (Mot 1999). The AltiVec vector type `__vector pixel` is omitted because VVM has no scalar pixel type.

Table B.2 shows the VPU vector and scalar types of each `vvm::vector` type in Apple GCC 3.1 20021003 and Apple GCC 3.3 20030304.

B.2 Function mappings

The functions executed by the AltiVec vector-processor implementation in relation to each VVM function are summarised in tables consisting of three columns: Expression, Types and AltiVec Operations. The Expression column contains an expression involving

Table B.1 VPU vector selection based on signedness and scalar size in bytes

VPU vector type	Element size	AltiVec vector type
signed integer	1	`__vector signed char`
unsigned integer	1	`__vector unsigned char`
signed integer	2	`__vector signed short`
unsigned integer	2	`__vector unsigned short`
signed integer	4	`__vector signed int`
unsigned integer	4	`__vector unsigned int`
real	4	`__vector float`
boolean	1	`__vector bool char`
boolean	2	`__vector bool short`
boolean	4	`__vector bool int`

Table B.2 The scalar and VPU vector types of `vvm::vectors` in Apple GCC 3.1 20021003 and Apple GCC 3.3 20030304.

VVM's vector	VPU vector	Scalar
`vector<char>`	`__vector signed char`	`signed char`
`vector<signed char>`		
`vector<unsigned char>`	`__vector unsigned char`	`unsigned char`
`vector<short int>`	`__vector signed short`	`short int`
`vector<unsigned short int>`	`__vector unsigned short`	`unsigned short int`
`vector<int>`	`__vector signed int`	`int`
`vector<long int>`		
`vector<unsigned int>`	`__vector unsigned int`	`unsigned int`
`vector<unsigned long int>`		
`vector<float>`	`__vector float`	`float`
`vector<double>`	`double`	`double`
`vector<long double>`	`long double`	`long double`
`vector<bool_char>`	`__vector bool char`	`bool_char`
`vector<bool_short>`	`__vector bool short`	`bool_short`
`vector<bool_int>`	`__vector bool int`	`bool_int`
`vector<bool_long>`		
`vector<bool_float>`	`__vector bool int`	`bool_float`
`vector<bool_double>`	`bool_double`	`bool_double`

a VVM function. For this expression, AltiVec Operations contains the functions used by the AltiVec vector-processor implementation for the VPU vector types specified in the Types column.

In the Expression and AltiVec Operations columns, the following variables are used:

A B C: Capital A to C represent `vvm::vectors`.

a b c: Lowercase a to c represent VPU vectors from the corresponding A to C `vvm::vector`.

p: Is a pointer to a scalar location that is aligned.

o: Refers to offset.

c: Refers to count.

In the Types column, five different entries are used. They are:

All: This refers to all AltiVec vector types. Integer and Float types combined together forms all AltiVec vector types.

None: This refers to no AltiVec vector type. Expressions that refer to None have no specialised implementation in the AltiVec vector-processor implementation.

Integer: This refers to AltiVec integer and boolean vector types. These types are `__vector signed char`, `__vector unsigned char`, `__vector bool char`, `__vector signed short`, `__vector unsigned short`, `__vector bool short`, `__vector signed int`, `__vector unsigned int`, and `__vector bool int`.

Unsigned Integer: This refers to unsigned AltiVec integer vector types.

Float: This refers to AltiVec floating-point vector types, namely `__vector float`.

Mladd: This refers to AltiVec vector types that support the `vec_mladd` function. These types are `__vector signed short` and `__vector unsigned short`.

Shift: This refers to `__vector unsigned char`, `__vector unsigned short`, and `__vector unsigned int`.

B.2.1 Memory management functions

All memory functions use AltiVec functions except for `store(A, B, p, o)`. The AltiVec vector-processor implementation only has prefetch channels 0 through 3.

In the table below, `EXPR(c)` refers to an expression involving `c` which returns the proper bitmask for prefetching.

Expression	Types	AltiVec Operation
`load(p)`	All	`vec_ld(0, p)`
`load(p, o)`	All	`vec_ld(o, p)`
`load(A, B, o)`	All	`switch, vec_sld(a, b, 0..15)`
`store(A, p)`	All	`vec_st(a, 0, p)`
`store(A, p, o)`	All	`vec_st(a, o, p)`
`store(A, B, p, o)`	None	
`prefetch<0..3, read_only>(p, c)`	All	`vec_dst(p, EXPR(c), 0..3)`
`prefetch<0..3, read_only_transient>(p, c)`	All	`vec_dstt(p, EXPR(c), 0..3)`
`prefetch<0..3, read_write>(p, c)`	All	`vec_dstst(p, EXPR(c), 0.33)`
`prefetch<0..3, read_write_transient>(p, c)`	All	`vec_dststt(p, EXPR(c), 0..3)`

B.2.2 VVM functions

While most VVM functions map to AltiVec functions in the AltiVec vector-processor implementation, in some cases, functions from the VecLib framework that comes with Mac OS X were used instead. The VecLib framework, which is provided by Apple Computer Inc, contains a set of vector libraries that cover many areas including digital signal processing, basic linear algebra subprograms (BLAS) and a range of numerical functions like square root.

Since VecLib's functions usually have different names for different types, wrappers were used when the same VecLib function was applied to more than one type. This was to enable one version of the code to refer to all functions. The VecLib wrapper functions are prefixed with `vecLib_`. Examples are `vecLib_mul` and `vecLib_div`.

Expression	Types	AltiVec Operation
`A + B`	All	`vec_add(a, b)`
`adds(A, B)`	Integer	`vec_adds(a, b)`
`A - B`	All	`vec_sub(a, b)`
`subs(A, B)`	Integer	`vec_subs(a, b)`
`A * B`	Integer	`vecLib_mul(a, b)`
`A / B`	Integer	`vecLib_div(a, b, 0)`

Expression	Types	AltiVec Operation
	Float	`vdivf(a, b)`
`A % B`[1]	Integer	`vecLib_div(a, b, &r)`
`++A, A++`	All	Same as `A + B`
`--A, A--`	All	Same as `A - 1`
`A += B`	All	`vec_add(a, b)`
`A -= B`	All	`vec_sub(a, b)`
`A *= B`	All	Same as `A * B`
`A /= B`	All	Same as `A / B`
`A %= B`[1]	Integer	Same as `A % B`
`-A`	None	
`+A`[2]	None	
`madd(A, B, C)`	Mladd	`vec_mladd(a, b, c)`
	Float	`vec_madd(a, b, c)`
`nmsub(A, B, C)`	Float	`vec_nmsub(a, b, c)`
`~A`	All	`vec_nor(a, a)`
`A ^ B`	All	`vec_xor(a, b)`
`A & B`	All	`vec_and(a, b)`
`A | B`	All	`vec_or(a, b)`
`A ^= B`	All	`vec_xor(a, b)`
`A &= B`	All	`vec_and(a, b)`
`A |= B`	All	`vec_or(a, b)`
`A << s`	Shift	`splat`, `vec_sel`, `vec_sll`, `vec_slo`, `vec_splat_u8`, `vec_srl`, `vec_sro`
`A <<= s`	Shift	`splat`, `vec_sel`, `vec_sll`, `vec_slo`, `vec_splat_u8`, `vec_srl`, `vec_sro`
`A >> s`	Shift	`splat`, `vec_sel`, `vec_sll`, `vec_slo`, `vec_splat_u8`, `vec_srl`, `vec_sro`
`A >>= s`	Shift	`splat`, `vec_sel`, `vec_sll`, `vec_slo`, `vec_splat_u8`, `vec_srl`, `vec_sro`
`A << B`	Shift	`vec_sl(a, b)`
`A <<= B`	Shift	`vec_sl(a, b)`
`A >> B`	Shift	`vec_sr(a, b)`
`A >>= B`	Shift	`vec_sr(a, b)`
`!A`	All	`vec_cmpeq(a, 0)`

[1]The modulus operator only applies to Integer types. So the AltiVec operation actually covers all required types.

[2]`+A` just returns `A`. So there is no need to optimise anything.

Expression	Types	AltiVec Operation
`A || B`	All	`vec_or(vec_cmpgt(a, 0), vec_cmpgt(b, 0))`
`A && B`	All	`vec_and(vec_cmpgt(a, 0), vec_cmpgt(b, 0))`
`A < B`	All	`vec_cmplt(a, b)`
`A > B`	All	`vec_cmpgt(a, b)`
`A <= B`	Integer	`vec_or(vec_cmplt(a, b), vec_cmpeq(a, b))`
	Float	`vec_cmple(a, b)`
`A >= B`	Integer	`vec_or(vec_cmpgt(a, b), vec_cmpeq(a, b))`
	Float	`vec_cmpge(a, b)`
`A == B`	All	`vec_cmpeq(a, b)`
`A != B`	All	`x = vec_cmpeq(a, b); vec_nor(x, x)`
`select(A, B, C)`	All	`vec_sel(a, b, c)`
`mergeh(A, B)`	All	`vec_mergeh(a, b, c)`
`mergel(A, B)`	All	`vec_mergel(a, b, c)`
`min(A)`	All	`vec_min(a, b)`
`max(A)`	All	`vec_max(a, b)`
`avg(A, B)`	Integer	`vec_avg(a, b)`
`all_eq(A, B)`	All	`vec_all_eq(a, b)`
`all_ge(A, B)`	All	`vec_all_ge(a, b)`
`all_gt(A, B)`	All	`vec_all_gt(a, b)`
`all_le(A, B)`	All	`vec_all_le(a, b)`
`all_lt(A, B)`	All	`vec_all_lt(a, b)`
`all_ne(A, B)`	All	`vec_all_ne(a, b)`
`all_nge(A, B)`	Integer	`vec_all_lt(a, b)`
	Float	`vec_all_nge(a, b)`
`all_ngt(A,B)`	Integer	`vec_all_le(a, b)`
	Float	`vec_all_ngt(a, b)`
`all_nle(A, B)`	Integer	`vec_all_gt(a, b)`
	Float	`vec_all_nle(a, b)`
`all_nlt(A, B)`	Integer	`vec_all_ge(a, b)`
	Float	`vec_all_nlt(a, b)`
`any_eq(A, B)`	All	`vec_any_eq(a, b)`
`any_ge(A, B)`	All	`vec_any_ge(a, b)`
`any_gt(A, B)`	All	`vec_any_gt(a, b)`

Expression	Types	AltiVec Operation
`any_le(A, B)`	All	`vec_any_le(a, b)`
`any_lt(A, B)`	All	`vec_any_lt(a, b)`
`any_ne(A, B)`	All	`vec_any_ne(a, b)`
`any_nge(A, B)`	Integer	`vec_any_lt(a, b)`
	Float	`vec_any_nge(a, b)`
`any_ngt(A,B)`	Integer	`vec_any_le(a, b)`
	Float	`vec_any_ngt(a, b)`
`any_nle(A, B)`	Integer	`vec_any_gt(a, b)`
	Float	`vec_any_nle(a, b)`
`any_nlt(A, B)`	Integer	`vec_any_ge(a, b)`
	Float	`vec_any_nlt(a, b)`
`abs(A)`	Float	`vec_abs(a)`
`acos(A)`	Float	`vacos(a)`
`asin(A)`	Float	`vasinf(a)`
`atan(A)`	Float	`vatanf(a)`
`atan2(A, B)`	Float	`vatan2f(a, b)`
`ceil(A)`	Float	`vec_ceil(a)`
`cos(A)`	Float	`vcosf(a)`
`cosh(A)`	Float	`vcoshf(a)`
`exp(A)`	Float	`vexpf(a)`
`floor(A)`	Float	`vec_floor(a)`
`fmod(A, B)`	Float	`vmodf(a, b)`
`log(A)`	Float	`vlog(a)`
`log10(A)`	Float	`vlog10f(a)`
`pow(A, B)`	Float	`vpowf(a, b)`
`pow(A, B)`[3]	Float	`vipowf(a, b)`
`sin(A, B)`	Float	`vsinf(a)`
`sinh(A, B)`	Float	`vsinhf(a)`
`sqrt(A)`	Float	`vsqrtf(a)`
`tan(A)`	Float	`vtanf(a)`
`tanh(A)`	Float	`vtanhf(a)`

[3] In this case B is a vvm::vector<int>

B.3 Type conversions

In this section, a table summarising the AltiVec functions used to perform `vvm::vector` type conversion is presented. Only the `vvm::vector` conversions listed use AltiVec functions. All other `vvm::vector` conversions use the emulation layer; each scalar is converted using `static_cast`. The table consists of three columns:

From: This column shows the VPU vector type of the `vvm::vector` that we want to convert from. See Section B.1 to derive the appropriate `vvm::vector` type.

To: This column shows the VPU vector type of the `vvm::vector` that we want to convert to. See Section B.1 to derive the appropriate `vvm::vector` type.

AltiVec Operation: This column lists the AltiVec functions used.

From	To	AltiVec Operation
`__vector signed char`	`__vector signed short`	`vec_unpackh`, `vec_unpackl`
`__vector bool char`	`__vector bool short`	
`__vector signed short`	`__vector signed int`	
`__vector bool short`	`__vector bool int`	
`__vector signed short`	`__vector signed char`	`vec_pack`
`__vector unsigned short`	`__vector unsigned char`	
`__vector bool short`	`__vector bool char`	
`__vector signed int`	`__vector signed short`	
`__vector unsigned int`	`__vector unsigned short`	
`__vector bool int`	`__vector bool short`	
`__vector signed int`	`__vector float`	`vec_cft`
`__vector unsigned int`	`__vector float`	
`__vector float`	`__vector signed int`	`vec_cts`
`__vector float`	`__vector unsigned int`	`vec_ctu`
`__vector unsigned int`	`__vector unsigned char`	3 * `vec_pack`
`__vector signed int`	`__vector signed char`	
`__vector bool int`	`__vector bool char`	
`__vector signed char`	`__vector signed int`	3 * `vec_unpackh`, 3 * `vec_unpackl`
`__vector bool char`	`__vector bool int`	

Appendix C

Constant Scalar Count

This appendix discusses why constant scalar count is required for generic programming. A simple example is discussed to show how generic algorithms would not be possible without constant scalar count.

Assume that we want to convert an 8-bit grey-scale image to a 16-bit grey-scale image. To perform this operation using a generic library, we would use a functor that converts an 8-bit grey-scale pixel to a 16-bit grey-scale pixel with a suitable algorithm. Assuming this functor is called `convert8to16`, the entire call will look something like the following:

```
// in: The input 8-bit grey-scale image
// out: The output 16-bit grey-scale image
transform(in, out, convert8to16);
```

A scalar implementation of `convert8to16` would be similar to the following:

```
// uint8: typedef for an 8-bit unsigned integer
// uint16: typedef for a 16-bit unsigned integer
uint16 convert8to16(uint8 in) {
  return in * 256;
}
```

With constant scalar count, a vectorised `convert8to16` would be similar to the following:

```
vvm::vuint16 convert8to16(vvm::vuint8 in) {
  return vvm::vector_cast<vvm::vuint16>(in) * vvm::vuint16(256);
}
```

Without constant scalar count, but fixed counts for each type, implementing a functor for `convert8to16` is not possible because of the possible mismatch between the number of elements in `vuint8` and `vuint16`. For example, assuming the normal VPU constraint where the vector size is constant across types, `vuint16` would have only half the number

of elements of `vuint8`. The `convert8to16` function would have to return two `vuint16`s instead of one. Such a change would require changing the `transform` algorithm, making it un-generic. A single `transform` algorithm would not be able to support all possibilities without constant scalar count.

Bibliography

Ablavsky, V., Stevens, M. R. & Pollak, J. B. (2003), 'Data-structure-independent algorithms for image processing', *C/C++ Users Journal* pp. 24–27,29,31.

Adv (2002), *AMD Athlon^{TM} Processor*.

Alexandrescu, A. (2001), *Modern C++ Design: Generic Programming and Design Patterns Applied*, Addison-Wesley.

Alexandrescu, A. (2002), 'Generic<programming>: Typelists and applications', *C/C++ Users Journal* .

AltiVec.org (2002), 'The AltiVec information source'.
URL: *http://www.altivec.org*

App (2002*a*), *Algorithms and Special Topics*.
URL: *http://developer.apple.com/hardware/ve/algorithms.html*

App (2002*b*), *AltiVec and the G3 and earlier processors*.
URL: *http://developer.apple.com/hardware/ve/g3_compatibility.html*

App (2002*c*), *Apple's AltiVec Home Page*.
URL: *http://developer.apple.com/hardware/ve*

App (2002*d*), *Carbon Developer Documentation, QuickDraw Reference*.

App (2002*e*), *Code Optimization*.
URL: *http://developer.apple.com/hardware/ve/code_optimization.html*

App (2002*f*), *Inside QuickTime*.

App (2002*g*), *Memory and Alignment*.
URL: *http://developer.apple.com/hardware/ve/alignment.html*

App (2002*h*), *The Objective-C Programming Language*.
URL: *http://developer.apple.com/documentation/Cocoa/Conceptual/ObjectiveC/-ObjC.pdf*

App (2002*i*), *Throughput and Latency*.
URL: *http://developer.apple.com/hardware/ve/throughput_vs_latency.html*

App (2003*a*), *CocoaVideoFrameToGWorld*.

App (2003*b*), *Graphics & Imaging: Quartz Primer: Code Samples*.

App (2003*c*), *PowerPC G5 Performance Primer*.
URL: *http://developer.apple.com/technotes/tn/tn2087.html*

App (2004*a*), *The Caches*.
URL: *http://developer.apple.com/hardware/ve/caches.html*

App (2004*b*), *Performance Issues: Memory Usage*.
URL: *http://developer.apple.com/hardware/ve/performance_memory.html*

App (2005), *vDSP Library*.

Bassetti, F., Davis, K. & Quinlan, D. (1998), C++ expression template performance issues in scientific computing, *in* 'Proceedings of the First Merged International Parallel Processing Symposium and Symposium on Parallel and Distributed Processing', pp. 635–639.

Becker, T. (2003), 'Expression templates', *C/C++ Users Journal* pp. 41–44.

Bettag, H. (2001), 'Quick and dirty AltiVec anchor'.
URL: *http://www.informatik.uni-bremen.de/ hobold/AltiVec.html*

Bik, A., Girkar, M., Grey, P. & Tian, X. (2003), 'Programming guidelines for vectorizing C/C++ compilers', *C/C++ Users Journal* pp. 26,28,30–31.

Blelloch, G. E., Chatterjee, S., Hardwick, J. C., Reid-Miller, M., Sipelstein, J. & Zagha, M. (1995), CVL: A C vector library, Technical Report CMU-CS-93-114 (Revised), School of Computer Science, Carnegie Mellon University.

Blelloch, G. E., Chatterjee, S., Hardwick, J. C., Sipelstein, J. & Zagha, M. (1994), 'Implementation of a portable nested data-parallel language', *Journal of Parallel and Distributed Computing* **21**(1), 4–14.

Blinn, J. F. (2000), 'Optimizing C++ vector expressions', *IEEE Computer Graphics and Applications* **20**, 97–103.

Boo (2003), *Boost C++ Libraries*.
URL: *http://www.boost.org*

Bulic, P. & Gustin, V. (2003), 'An extended ANSI C for processors with a multimedia extension', *International Journal of Parallel Programming* **31**(2), 107–136.

Chiba, S. (1995), A metaobject protocol for C++, *in* 'Proceedings of the ACM Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA)', pp. 285–299.

Chiba, S. (1998*a*), Macro processing in object-oriented languages, *in* 'Proc. of Technology of Object-Oriented Languages and Systems (TOOLS Pacific '98)'.

Chiba, S. (1998*b*), *Open C++ Tutorial*.
URL: *http://www.csg.is.titech.ac.jp/ chiba/opencxx/tutorial.pdf*

Cre (2003), *VAST/AltiVec*.
URL: *http://www.psrv.com/vast_altivec.html*

Czarnecki, K. & Eisenecker, U. W. (2000), *Generative Programming*, Addison-Wesley.

Davies, E. R. (1990), *Machine Vision: Theory, Algorithms, Practicalities*, Academic Press Limited.

Ded (2001), *DNA Computing Solutions Announces G4 Optimized VSIPL Library*.
URL: *http://www.realtime-info.be/vpr/layout/display/pr.asp?PRID=2316*

Dig (2004), *The D Programming Language*.
URL: *http://www.digitalmars.d*

DNA (2001), *VSIPL Library*.
URL: *http://www.dna-cs.com/products/DNA VSIPL.pdf*

Fuller, S. (1999), 'Motorola PowerPC™ microprocessor with AltiVec™ technology-the system engineer's answer to revolutionizing life cycle cots design paradigms with software-based high performance computing', *Digital Avionics Systems Conference, 1999. Proceedings. 18th* **C.2-7 vol.1**, 1.C.4–1 – 1.C.4–8.

Geo (2001), *VSIPL 1.01 API.*
URL: *http://www.vsipl.org/PubInfo/vsiplv1p01_final1.pdf*

Group, V. T. (2000), *The Microsoft Vision SDK, Version 1.2*.

Gurtovoy, A. & Abrahams, D. (2002), *The Boost C++ Metaprogramming Library*.

Hardwick, J. C. (1995), 'Porting a vector library: a comparison of MPI, Paris CMMD and PVM'.

Hatcher, P. J., Quinn, M. J., Anderson, R. J., Lapadula, A. J., Seevers, B. K. & Bennett, A. F. (1991), Architecture-independent scientific programming in data parallel C: three case studies, *in* 'Proceedings of the 1991 ACM/IEEE conference on Supercomputing', ACM Press, pp. 208–217.

Hew (2003), *Itanium Vector Math Library (HP-VML)*.

Hinnant, H., Järvi, J., Lumsdaine, A. & Willcock, J. (2003), 'Function overloading based on arbitrary properties of types', *C/C++ Users Journal* **21**(6), 25–26, 28–32.

IBM (2002), *VisualAge C++ for AIX Compiler Reference*.
URL: *http://hpcf.nersc.gov/vendor_docs/ibm/vac/sc094959.pdf*

Inf (1998), *ISO/IEC 14882 Programming languages – C++*, first edn.

Int (2000-2001), *Integrated Performance Primitives for Intel Architecture*.

Int (2005), *IA-32 Intel® Architecture Software Developer's Manual*.

Intel® Integrated Performance Primitives Product Features (2004).
URL: *http://www.intel.com/software/products/ipp/features.htm*

Jaenicke, R. (1999), 'An open systems approach to real-time signal processing using AltiVec technology'.

Köthe, U. (1998), 'On data access via iterators (working draft)'.
URL: *http://kogs-www.informatik.uni-hamburg.de/~koethe/vigra/doc/documents/DataAccessors.ps*

Köthe, U. (1999), 'Reusable software in computer vision', *B. Jähne, H. Haussecker, P. Geissler: "Handbook on Computer Vision and Applications"* **3**.
URL: *http://kogs-www.informatik.uni-hamburg.de/~koethe/papers/handbook.ps.gz*

Köthe, U. (2000*a*), 'Expression templates for automated functor creation'.
URL: *http://kogs-www.informatik.uni-hamburg.de/~koethe/vigra/doc/documents/FunctorFactory.ps*

Köthe, U. (2000*b*), Generische Programmierung für die Bildverarbeitung, PhD thesis, Universität Hamburg.
URL: *http://kogs-www.informatik.uni-hamburg.de/~koethe/papers/DissPrint.ps.gz*

Köthe, U. (2000*c*), 'STL-style generic programming with images', *C++ Report Magazine* pp. 24–30.
URL: *http://kogs-www.informatik.uni-hamburg.de/~koethe/papers/GenericProg2DC++Report.ps.gz*

Köthe, U. (2001), *VIGRA Reference Manual for 1.1.2*.
URL: *http://kogs-www.informatik.uni-hamburg.de/~koethe/vigra/doc/index.html*

Kühl, D. & Weihe, K. (1997), 'Data access templates', *C++ Report* **9/7**, 15, 18–21.

Lai, B.-C. & McKerrow, P. J. (2001), Programming the Velocity Engine, *in* N. Smythe, ed., 'e-Xplore 2001: a face to face odyssey', Apple University Consortium.

Lai, B.-C., McKerrow, P. J. & Abrantes, J. (2002), Vectorized machine vision algorithms using AltiVec, Australasian Conference on Robotics & Automation.

Langer, A. & Kreft, K. (2003), 'C++ expression templates', *C/C++ Users Journal* pp. 27–33.

Ma, W.-C. & Yang, C.-L. (2002), Using Intel Streaming SIMD Extensions for 3D geometry processing, The 3rd IEEE Pacific-Rim Conference on Multimedia, pp. 1080–1087.

Meyers, S. (1996), *More Effective C++. 35 New Ways to Improve Your Programs and Designs*, Addison-Wesley.

Meyers, S. (2002), 'Class template, member template – or both?', *C/C++ Users Journal* pp. 34,36–38.

Mittal, M., Peleg, A. & Weiser, U. (1997), 'MMX technology architecture overview', *Intel Technology Journal '97* .

Mot (1999), *AltiVec Technology Programming Interface Manual*.
URL: *http://e-www.motorola.com/brdata/PDFDB/MICROPROCESSORS/32_BIT/-POWERPC/ALTIVEC/ALTIVECPIM.pdf*

MPI (1999-2000), *VSI/Pro*.
URL: *http://www.mpi-softtech.com/products/vsipro/VSI_Pro9_01.pdf*

Musser, D. R. & Stepanov, A. A. (1994), 'Algorithm-oriented generic libraries', *Software - Practice and Experience* **24**(7), 623–642.
URL: *citeseer.nj.nec.com/article/musser94algorithmoriented.html*

Musser & Stepanov (1989), Generic programming, *in* 'ISSAC: Proceedings of the ACM SIGSAM International Symposium on Symbolic and Algebraic Computation (formerly SYMSAM, SYMSAC, EUROSAM, EUROCAL) (also sometimes in cooperation with the Symbolic and Algebraic Manipulation Groupe in Europe (SAME))'.
URL: *citeseer.nj.nec.com/musser88generic.html*

Nat (1999), *IMAQ Vision User Manual*.

Nat (2002*a*), *LabVIEW Vision Development Module*.
URL: *http://www.ni.com/pdf/products/us/3msw58_76.pdf*

Nat (2002*b*), *Vision Development Module*.
URL: *http://www.ni.com/pdf/products/us/3vis579-583.pdf*

Nat (2004), *Vision Development Modules for LabVIEW, LabWindows/CVI, and Measurement Studio*.
URL: *http://www.ni.com/pdf/products/us/pg596_600_vision_dev_module.pdf*

Nat (2005), *LabVIEW*.
URL: *http://wwww.ni.com/labview*

Oberdorfer, M. & Gutowski, J. (2004), 'CxC & parallel programming', *C/C++ Users Journal* pp. 42–47.

Ollmann, I. (2001), AltiVec.
URL: *http://www.alienorb/AltiVec/Altivec.pdf*

Optimizing Applications with Intel C++ and Fortran Compilers for Windows and Linux (2003), Technical report.

Ora (2002), *iBot FireWire Web Cam*.
URL: *http://www.orangemicro.com/ibot.html*

Othmer, K. & Reed, M. (1992), 'Drawing in GWorlds for speed and versatility', *MacTech* .

Parker, J. R. (1994), *Practical Computer Vision Using C*, John Wiley & Sons, Inc.

Performance Benchmarks for Intel® Integrated Performance Primitives (2003), Technical report.

Pix (2003), *macstl 0.1.2*.
URL: *http://www.pixelglow.com/macstl/*

Qureshi, S. (2004), 'Policy-driven design & the Intel IPP library', *C/C++ Users Journal* pp. 38,40–42.

Rizzoli, V., Ferlito, M. & Neri, A. (1986), 'Vectorized program architectures for supercomputer-aided circuit design', *IEEE Transactions on Microwave Theory and Techniques* **34**(1).

Scales, H. (2000), 'AltiVec extension to PowerPC accelerates media processing', *IEEE Micro* pp. 85–95.

Siek, J. G. (1999), A modern framework for portable high performance numerical linear algebra, Master's thesis, University of Notre Dame.

SKY (1999), *SKY/VSIPL*.
URL: *http://www.skycomputers.com/news/vsipl.html*

Thacker, N., Courtney, P., Walker, S., Evans, S. & Yates, R. (1994), Specification and design of a general purpose image processing chip, *in* 'Proceedings of the 12th IAPR International Conference on Signal Processing', Vol. 3, pp. 268–273.

Tra (2001), *Transtech DSP's VSIPL*.
URL: *http://www.transtech-dsp.com/vsipl.htm*

van Rossum, G. (2003), *Python Tutorial*.
URL: *http://www.python.org/doc/2.3.3/tut/tut.html*

Vandevoorde, D. & Josuttis, N. M. (2003), *C++ Templates*, Addison-Wesley.

Veldhuizen, T. (1995*a*), 'Using C++ template metaprograms', *C++ Report* **7**(4), 36–43. Reprinted in C++ Gems, ed. Stanley Lippman.

Veldhuizen, T. (2000), Techniques for scientific C++, Technical report, Indiana University.

Veldhuizen, T. (2001), Blitz++ user's guide, Technical report.

Veldhuizen, T. L. (1995*b*), 'Expression templates', *C++ Report* **7**(5), 26–31. Reprinted in C++ Gems, ed. Stanley Lippman.

Veldhuizen, T. L. & Gannon, D. (1998), Active libraries: Rethinking the roles of compilers and libraries, *in* 'Proceedings of the SIAM Workshop on Object Oriented Methods for Inter-operable Scientific and Engineering Computing (OO'98)', SIAM Press.

VSIPL website (2001). Maintained by Dr. Mark Richards of Georgia Tech Research Institue for DARPA and U.S. Navy.

Walls, K. & Fegreus, J. (2002), 'High Cmanship', *Open (http://www.open-mag.com)* .
URL: *http://www.open-mag.com/754088105111.htm*

Weihe, K. (2002), 'Towards improved static safety: Expressing meaning by type', *C/C++ Users Journal* pp. 32,34–36,38–40.

Weiser, U. (1996), 'MMX technology extension to the Intel architecture', *IEEE Micro* pp. 42–50.

Young, I. T., Gerbrands, J. J. & van Vliet, L. J. (n.d.), 'Image processing fundamentals'.
URL: *http://www.ph.tn.tudelft.nl/Courses/FIP/noframes/fip.html*