# EVE, an Object Oriented SIMD Library

Joel Falcou and Jocelyn Sérot

LASMEA, UMR 6602 CNRS/Univ. Blaise Pascal, Clermont-Ferrand, France,
{falcou,jserot}@lasmea.univ-bpclermont.fr

**Abstract.** This paper describes EVE (Expressive Velocity Engine), an object oriented C++ library designed to ease the process of writing efficient numerical applications using AltiVec, the SIMD extension designed by Apple, Motorola and IBM for PowerPC processors. Compared to the Altivec original C API, EVE, offers a significant improvement in terms of expressivity. By relying on template metaprogramming techniques, this is not obtained at the expense of efficiency.

## 1 Introduction

AltiVec [4] is an extension designed to enhance PowerPC[1] processor performance on applications handling large amounts of data. The AltiVec architecture is based on a SIMD processing unit integrated with the PowerPC architecture. It introduces a new set of 128 bit wide registers distinct from the existing general purpose or floating-point registers. These registers are accessible through 160 new "vector" instructions that can be freely mixed with other instructions (there are no restriction on how vector instructions can be intermixed with branch, integer or floating-point instructions with no context switching nor overhead for doing so). Altivec handles data as 128 bit vectors that can contain sixteen 8 bit integers, eight 16 bit integers, four 32 bit integers or four 32 bit floating points values. For example, any vector operation performed on a `vector char` is in fact performed on sixteen `char` simultaneously and is theoretically running sixteen times faster as the scalar equivalent operation. AltiVec vector functions cover a large spectrum, extending from simple arithmetic functions (additions, subtractions) to boolean evaluation or lookup table solving.

Altivec is natively programmed by means of a C API [2]. Programming at this level can offer significant speedups (from 4 to 12 for typical signal processing algorithms) but is a rather tedious and error-prone task, because this C API is really "assembly in disguise". The application-level vectors (arrays, in variable number and with variable sizes) must be explicitly mapped onto the Altivec vectors (fixed number, fixed size) and the programmer must deal with several low-level details such as vector padding and alignment. To address this programmability issue, we have investigated the possibility to provide a higher level API, in the form of a C++ class library. This class library should encapsulate all the low-level details related to the manipulation of Altivec `vectors`

[1] PPC 74xx (G4) and PPC 970 (G5).

and provide a fully abstract `Array/Vector` object and the associated functions as overloaded operators – allowing for example code to be written in the style of Fig. 1.

```
Vector<float> a(1000),b(1000),c(1000),r(1000);
r = a * b + c;
```

**Fig. 1.** Vector processing with a high-level API

It is well known, however, that the code generated by such "naive" class libraries is often very inefficient [7], due to the unwanted copies caused by temporaries[2]. This has led, at least in the domain of C++ scientific computing, to the development of *Active Libraries* [13,12,8,9], which both provide domain-specific abstractions and dedicated code optimization mechanisms. This paper describes how this approach can be applied to the specific problem of efficient Altivec code generation from a high-level C++ API. It is organized as follows. Sect. 2 explains why generating efficient code for vector expressions is not trivial and introduces the concept of template-based meta-programming. Sect. 3 explains how this can used to generate optimized Altivec code. Sect. 4 rapidly presents the API of the library we built upon these principles. Performance results are presented in Sect. 5. Sect. 6 is a brief survey of related work and Sect. 7 concludes.

## 2 Template Based Meta-programming

The problem of generating efficient code for vector expressions can be understood – in essence – starting with the code sample given in Fig. 1. Ideally, this code should be inlined as:

```
for ( i=0; i<1000; i++ ) r[i] = a[i] * b[i] + c[i];
```

In practice, due to the way overloaded operators are handled in C++, it is developed as:

```
Vector<float> __t1(1000), __t2(1000);
for (int i=0; i < 1000; ++i)  __t1[i] = a[i] * b[i];
for (i=0; i < 1000; ++i) __t2[i] = __t1[i] * c[i];
for (i=0; i < 1000; ++i) r[i] = __t2[i];
```

The allocation of temporary vectors and the redundant loops result in poor efficiency. For more complex expressions, the performance penalty can easily reach one order of magnitude. In this case, it is clear that expressiveness is obtained at a prohibitive cost. This problem can be overcome by using an advanced C++ technique known as *expression templates*. The basic idea of *expression templates*

[2] In our case, the observed speedup for code like the one given in Fig. 1 went down from 3.9 – when coded with the native C API – to less than 0.8 – with a "naive" `Vector` class.

is to create parse trees of vector expressions at compile time and to use these parse trees to generate customized loops. This is actually done in two steps[3]. In the first step, the parse trees – represented as C++ types – are constructed using recursive template instantiations and overloaded versions of the vector operators (`+`, `*`, etc.). In the second step, these parse trees are "evaluated" using overloaded versions of the assignment (`=`) and indexing (`[]`) operators to compute the left-hand side vector in a single pass, with no temporary. Both steps rely on the definition of two classes:

– an `Array` class, for representing application-level vectors (arrays); this class has a member `data_`, where the array elements are stored and an operator `[]` for accessing these elements:

```
float Array::operator[](int index) { return data_[index]; }
```

– an `Xpr` class, for representing (encoding) vector *expressions* :

```
template<class LEFT,class OP,class RIGHT> class Xpr {};
```

Consider for example, the statement `r=a*b+c` given in the code sample above. Its right-hand side expression (`a*b+c`, where `a`, `b` and `c` have type `Array`) will be encoded with the following C++ type:

```
Xpr<Xpr<Array,mul,Array>,add,Array>
```

This type will be automatically built from the expression syntax `a*b+c` using overloaded versions of the `+` and `*` operators:

```
template<class T> Xpr<T,add,Array> operator+(T, Array)
  { return Xpr<T,add,Array>(); }
template<class T> Xpr<T,mul,Array> operator*(T, Array)
  { return Xpr<T,mul,Array>(); }
```

The "evaluation" (at compile time) of the encoded vector expression is carried out by an overloaded version of the assignment operator (`=`):

```
template<class T> Array& Array::operator=( const T& xpr ) {
  for(int i=0;i<size;i++) data_[i] = xpr[i];
  return *this;
}
```

For this, the `Xpr` class provides an `operator[]` method, so that each element of the result vector (`data_[i]`) gets the value `xpr[i]`:

```
template<class LEFT,class OP,class RIGHT>
float X<LEFT,OP,RIGHT>::operator[](int index)
  { return OP::eval(left_[index],right_[index]); }
```

[3] The presentation given here is deliberately simplified, due to space limitations. More details can be found, for example, in Veldhuizen's papers [5,6,7].

where `left_` and `_right` are the members storing the left and right subexpressions of an `Xpr` object and `eval` the static method of the C++ functor associated with the vector operation `OP`. Such a functor will be defined for each possible operation. For example, the `add` and `mul` functors associated with the `+` and `*` vector operators are defined as:

```
class add { static float eval(float x,float y) { return x+y; } };
class mul { static float eval(float x,float y) { return x*y; } };
```

Using the mechanism described above, a standard C++ compiler can reduce the statement `r=a*b+c` to the following "optimal" code:

```
for ( i=0; i<1000; ++i) r[i] = a[i]*b[i]+c[i];
```

## 3 Application to the Generation of Efficient AltiVec Code

The template-based meta-programming technique described in the previous section can readily be adapted to support the generation of efficient Altivec code. For this:

- the `Array` class must provide a `load` method returning an Altivec vector instead of a scalar,
- the `add` (resp. `mul`, etc.) functor must call the native `vec_add` (resp. `vec_mul`, etc.) instruction,
- the assignment operator must use the native `vec_st` instruction to store the result:

```
vector float Array::load(int index) { return vec_ld(data_,index*16); }

class add {eval(vector float x,vector float y) { return vec_add(x,y); }};

template<class T>
Array& Array::operator=(T xpr) {
  for(int i=0;i<size/4;i++) vec_st(xpr.load(i),0,data);
  return *this;
}
```

With this approach, the assembly code resulting from the compilation of the previous example (`r=a*b+c`) contains three vector load operations, one vector addition, one vector multiplication and one vector store, which is clearly optimal.

## 4 The EVE Library

Using the code generation technique described in the previous section, we have produced a high-level array manipulation library aimed at scientific computing and taking advantage of the SIMD acceleration offered by the Altivec extension on PowerPC processors. This library, called EVE (for *Expressive Velocity Engine*) basically provides two classes, `array` and `matrix` – for 1D and 2D arrays –, and a rich set of operators and functions to manipulate them. This set can be roughly divided in four families:

1. **Arithmetic and boolean operators**, which are the direct vector extension of their C++ counterparts. For example:

   ```
   array<char>  a(64),b(64),c(64),d(64);
   d = (a+b)/c;  // d[i] = (a[i]+b[i])/c[i], for i=0...63
   ```

2. **Boolean predicates.** These functions can be used to manipulate boolean vectors and use them as selection masks. For example:

   ```
   array<char>  a(64),b(64),c(64);
   c = where(a<b, a, b); // c[i] = a[i]<b[i] ? a[i] : b[i], for i=0...63
   ```

3. **Mathematical and STL functions.** These functions work like their STL or `math.h` counterparts. The only difference is that they take an array (or matrix) as a whole argument instead of a couple of iterators. Apart from this difference, EVE functions and operators are very similar to their STL counterparts (the interface to the EVE `array` class is actually very similar to the one offered by the STL `valarray` class. This allows algorithms developed with the STL to be ported (and accelerated) with a minimum effort on a PowerPC platform with EVE. Example:

   ```
   array<float>  a(64),b(64);
   b = tan(a);  // b[i] = tan(a[i]) for each i=0..63
   float r = inner_product(a, b); // r = a[0]*b[0]+...+a[63]*b[63]
   ```

4. **Signal processing functions.** These functions allow the direct expression (without explicit decomposition into sums and products) of 1D and 2D FIR filters. For example:

   ```
   array<float>  a(64),b(64);
   filter< 3,1,mask<1,2,1> >  gaussian;
   res = gaussian(image);
   ```

## 5 Performance

Two kinds of performance tests have been performed: basic tests, involving only one vector operation and more complex tests, in which several vector operations are composed into more complex expressions. All tests involved vectors of different types (8 bit integers, 16 bit integers, 32 bit integers and 32 bit floats) but of the same total length (16 Kbytes) in order to reduce the impact of cache effects on the observed performances[4]. They have been conducted on a 1.2GHz PowerPC G4 with `gcc 3.3.1` and the following compilation switches: `-faltivec -ftemplate-deph-128 -O3`. A selection of performance results is given in Table 1. For each test, four numbers are given: the maximum theoretical speedup[5] (TM), the measured speedup for a hand-coded version of the test using the native C API (NC), the measured speedup with a "naive" vector library – which does not use the expression template mechanism described in Sect. 2 (NV), and the measured speedup with the EVE library.

[4] I.e. the vector size (in elements) was 16K for 8 bit integers, 8K for 16 bit integers and 4K for 32 bits integers or floats.

[5] This depends on the type of the vector elements : 16 for 8 bit integers, 8 for 16 bit integers and 4 for 32 bit integers and floats.

**Table 1.** Selected performance results

| Test | Vector type | TM | NC | NV | EVE |
|---|---|---|---|---|---|
| 1. `v3=v1+v2` | 8 bit integer | 16 | 15.7 | 8.0 | 15.4 |
| 2. `v2=tan(v1)` | 32 bit float | 4 | 3.6 | 2.0 | 3.5 |
| 3. `v3=v1/v2` | 32 bit float | 4 | 4.8 | 2.1 | 4.6 |
| 4. `v3=v1/v2` | 16 bit integer | 8(4) | 3.0 | 1.0 | 3.0 |
| 5. `v3=inner_prod(v1,v2)` | 8 bit integer | 8 | 7.8 | 4.5 | 7.2 |
| 6. `v3=inner_prod(v1,v2)` | 32 bit float | 4 | 14.1 | 4.8 | 13.8 |
| 7. 3x1 FIR | 8 bit integer | 8 | 7.9 | 0.1 | 7.8 |
| 8. 3x1 FIR | 32 bit float | 4 | 3.7 | 0.1 | 3.7 |
| 9. `v5=sqrt(tan(v1+v2)/cos(v3*v4))` | 32 bit float | 4 | 3.9 | 0.04 | 3.9 |
| 10. Image processing algorithm | 16 bit integer | 8 | 6.9 | 0.1 | 2.7 |

It can be observed that, for most of the tests, the speedup obtained with EVE is close to the one obtained with a hand-coded version of the algorithm using the native C API. By contrast, the performances of the "naive" class library are very disappointing (especially for tests 7-10). This clearly demonstrates the effectiveness of the metaprogramming-based optimization.

Tests 1-3 correspond to basic operations, which are mapped directly to a single AltiVec instruction. In this case, the measured speedup is very close to the theoretical maximum. For test 3, it is even greater. This effect can be explained by the fact that on G4 processors, and even for non-SIMD operations, the Altivec FPU is already faster than the scalar FPU[6]. When added to the speedup offered by the SIMD parallelism, this leads to super-linear speedups. The same effect explains the result obtained for test 6. By contrast, test 4 exhibits a situation in which the observed performances are significantly lower than expected. In this case, this is due to the asymmetry of the Altivec instruction set, which does not provide the basic operations for all types of vectors. In particular, it does not include division on 16 bit integers. This operation must therefore be emulated using vector float division. This involves several type casting operations and practically reduces the maximum theoretical speedup from 8 to 4.

Tests 5-9 correspond to more complex operations, involving *several* AltiVec instructions. Note that for tests 5 and 7, despite the fact that the operands are vectors of 8 bit integers, the computations are actually carried out on vectors of 16 bit integers, in order to keep a reasonable precision. The theoretical maximum speedup is therefore 8 instead of 16.

In order to show that EVE can be used to solve realistic problems, while still delivering significant speedups, we have used it to vectorize several complete image processing algorithms. Test 10, for example, give the performances obtained with an algorithm performing the detection of *points of interest* in grey scale images using the Harris filter [1]. This algorithm involves several filtering steps (on both directions of the image) and matrix computations. The measured speedup, while being lower than the one obtained with the hand-coded version,

[6] It has more pipeline stages and a shortest cycle time.

is still satisfactory if we take into account the large difference in code size and complexity between the two versions (15 lines of C++ with EVE, 80 lines of C with the Altivec native API).

## 6 Related Work

Since its introduction, most of the development for the Altivec has been conducted using the native C API and very few projects have proposed higher-level alternatives to this design flow.

Apple proposes the **VecLib** [3] library, as a complement to the native C API. This framework provides software equivalent of some missing functions and more complex operations such as FFT or convolution. Compared to EVE, the level of expressiveness is lower (VecLib does not support the construction of complex vector expressions by means of overloaded operators, in particular). Correlatively, no syntax directed optimization can be performed and performances drop when a lot of function calls are sequenced in the same program.

The **VAST** code optimizer [10] – which offers automatic vectorization and parallelization from source – has a specific back-end for generating Altivec code. This tool automatically replaces loops in C/C++ programs with inline vector extensions, and loops in Fortran programs with calls to newly-generated C functions with inline vector extensions. These vector extensions allow VAST to access the AltiVec unit at close to instruction level efficiency while remaining at the source code level. VAST's speedups are generally very close to those obtained with hand-vectorized code. VAST is a commercial product and costs 3000$.

The **Mac STL** [11] C++ library is very similar, in goals and design principles, to EVE. It provides a fast `valarray` class optimized for Altivec and relies on template-based metaprogramming techniques for code optimization. The only difference is that MacSTL only provides STL-compliant functions and operators (it is viewed as a specific implementation of the STL for G4/G5 computers) whereas EVE offers additional domain-specific functions (for signal processing, for example). Mac STL is available for a low-cost license.

## 7 Conclusion

We have shown how a classical technique – template-based metaprogramming – can be applied to the design and implementation of an efficient high-level array manipulation library aimed at scientific computing on PowerPC platforms. This library offers a significant improvement in terms of expressivity over the native C API traditionnaly used for taking advantage of the SIMD capabilities of this processor. It allows developers to obtain significant speedups without having to deal with low level implementation details. The EVE API is largely compliant with the STL standard and therefore provides a smooth transition path for applications written with other scientific computing libraries. A prototype version of the library can be downloaded from the following URL: http://wwwlasmea.univ-bpclermont.fr/Personnel/falcou/EVE/download.html. We are currently working on improving the performances obtained with this prototype for complex, realistic applications. This involves, for instance, globally minimizing the number of vector load and store operations, using more judiciously Altivec-specific cache manipulation instructions or taking advantage of fused operations (e.g. multiply/add). Finally, it can be noted that, although the current version of EVE has been designed for PowerPC processors with Altivec, it could be retargeted, with a moderate effort, to Pentium 4 processors with MMX/SSE2 because the code generator itself (using the expression template mechanism) can be made largely independent of the SIMD instruction set.

## References

1. C. Harris and M. Stephens. A combined corner and edge detector. In 4th Alvey Vision Conference, 1988.
2. Apple. AltiVec Instructions References, Tutorials and Presentation. http://developer.apple.com/hardware/ve.
3. Apple. The VecLib framework. http://developer.apple.com/hardware/ve/vector_libraries.html
4. I. Ollman. AltiVec Velocity Engine Tutorial. http://www.simdtech.org/altivec. March 2001.
5. T. Veldhuizen. Using C++ Template Meta-Programs. In C++ Report, vol. 7, p. 36-43,1995.
6. T. Veldhuizen. Expression Templates. In C++ Report, vol. 7, p. 26-31, 1995.
7. T. Veldhuizen. Techniques for Scientific C++. http://osl.iu.edu/~tveldhui/papers/techniques/
8. T. Veldhuizen. Arrays in Blitz++. In Dr Dobb's Journal of Software Tools, p. 238-44, 1996.
9. The BOOST Library. http://www.boost.org/.
10. VAST. http://www.psrv.com/vast_altivec.html/.
11. G. Low. Mac STL. http://www.pixelglow.com/macstl/.
12. The POOMA Library. http://www.codesourcery.com/pooma/.
13. T. Veldhuizen and D. Gannon. Active Libraries: Rethinking the roles of compilers and libraries Proc. of the SIAM Workshop on Object Oriented Methods for Interoperable Scientific and Engineering Computing SIAM Press, 1998