High level synthesis
performance analysis and code optimization

Hendriks, J.

Award date:
2012

Eindhoven University of Technology

Master Thesis Embedded Systems

High Level Synthesis: Performance Analysis and Code Optimization

August 14, 2012

Author:
Johan Hendriks

Supervisor:
Prof. Dr. H. Corporaal

Tutors:
Ir. Z. Ye
Ir. S. Fernando

Abstract

With increasing FPGA chip density, it is possible to implement more sophisticated algorithms on FPGA. However, programming an FPGA using a register transfer level (RTL) language is time-consuming and error-prone. To make use of the re-programmability of FPGAs for fast design space exploration and fast time-to-market it becomes more and more necessary to raise the abstraction level from structural design to behavioral design. High level synthesis (HLS) seems a promising solution for this issue.

The sequential nature of C as input specification is an issue in HLS and code transformations are often necessary to produce better quality of results. The question is if HLS can compete with handwritten RTL designs, with regard to several performance metrics such as resource utilization, execution time and design time. If HLS can compete with handwritten RTL design, the question is which code transformations are required to achieve good quality of results.

Using the HLS tools AutoESL and Synphony, two image processing kernels are benchmarked for an in-depth analysis of HLS performance. To avoid performance pitfalls by using only small toy examples, a real application called Fast Focus on Structures (FFoS) is ported to FPGA to investigate performance with regard to a comparable handwritten RTL design.

The benchmarks using two image processing kernels have shown that it is possible to efficiently mimic a datapath from a reference manual RTL design. However, the analysis also shows that in HLS it is impossible to describe fine-grained interface and memory control. Both AutoESL and Synphony are capable of producing a design similar to the reference design. AutoESL requires 5% less flip flops and 4% more LUTs whereas Synphony requires 62% more flip flops and 61% more LUTs. Design time, for both tools, is reduced by a factor 6 and the latency stays within 2% of the reference. The explored code transformations, in combination with an existing algorithm classification and source-to-source compiler, are used to create a skeleton library for HLS to decrease design time by a factor 15 compared to handwritten RTL design.

From the two HLS tools, AutoESL seems the most promising although Synphony might produce better results in a different application domain. Furthermore, it is shown that small modifications at behavioral level can have a large positive impact on the performance results without requiring time-consuming and error-prone RTL modifications. The code transformations show that in many cases tool specific code optimizations need to be done to achieve good results, making it debatable if it can be automated.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Embedded systems come in many shapes and sizes, offering solutions for a wide variety of problems in many different application domains. Their presence ranges from mobile phones and washing machines to avionics and medical equipment. The complexity of an embedded system varies from single core processors to multi-core and even many-core architectures, including a wide variety of possible peripherals. With the diminishing returns on more instruction level parallelism and a stall in uniprocessor performance increase, the community has moved to multi-core designs [1]. Although multi-core solutions offer more parallelism and look promising for general purpose processing, they might not be efficient in performance and flexible enough for certain specific tasks (for example real-time systems) within an embedded system. Reconfigurable computing platforms can overcome this problem by offering efficiency, performance and flexibility, often wanted in for example the image processing domain. One of the major problems with reconfigurable computing platforms is the design time necessary to get the required efficiency and performance. This chapter introduces a possible solution for the time-consuming design time and presents the problem description and contributions. The design time can be reduced by the acceptance of high level synthesis (HLS), a compiler for reconfigurable computing platforms, in the design flow. A requirement for its acceptance is that state-of-the-art HLS tools should be capable of producing high performance designs. Section 1.1 introduces reconfigurable computing platforms and section 1.2 discusses some history in compiling for reconfigurable platforms. Section 1.3 introduces high level synthesis as a new form of compiling for reconfigurable architectures, including a brief discussion about its history and covering several aspects of the high level synthesis design flow. In section 1.4, the image processing domain and algorithmic skeletons are introduced. Readers already known with previously mentioned concepts can skip to section 1.5 and section **??** for the problem description and contributions of the thesis.

1.1 Reconfigurable Computing Platforms

The idea of reconfigurable computing has been there since the 60's. While general purpose processors or microprocessors have a fixed architecture, reconfigurable computing architectures contain reconfigurable processing units (RPUs). In a normal processor, the data path and handling of control flow is fixed, but with a reconfigurable architecture the designer has the possibility to design a data path and control flow. There is many interest in reconfigurable platforms as they can be used to accelerate computations or even complete algorithms by use of parallel hardware structures which can be completely customized. General purpose processors have a fixed data path which may not be optimal for a certain computation. On the other side there's application specific integrated circuits (ASIC) and application specific instruction set processors (ASIP). ASIPs have a specially designed instruction set targeting a specific application, such a processor is a trade-off between the (application) flexibility of a general purpose processor and the performance of an ASIC which is a specially designed integrated circuit executing a specific functionality or application. The gap between flexibility and performance can be filled with reconfigurable computing platforms. The goal of reconfigurable computing is to fill the gap between hardware and software, aiming for more performance than a software implementation of an algorithm on a general purpose CPU while maintaining flexibility in hardware.

Reconfigurable computing platforms come in many different granularities. A platform might consist of a completely customizable reconfigurable architecture or it might be a hybrid form consisting of a (reduced) general purpose processor coupled with a reconfigurable array of reconfigurable processing units. The task of the general purpose processor in such an architecture is to control the hardware accelerator custom build in the reconfigurable part and to supply the accelerator with data to operate on. The reconfigurable part of the architecture consists of reconfigurable processing units which also come in many different granularities, ranging from coarse grained to fine grained RPUs. An example of a coarse grained RPU architecture is the KressArray [2], consisting of a reconfigurable data path array laid out as a rectangular grid of 32-bit ALUs with three levels of interconnect (nearest neighbor, row/column back-buses and a global bus). Field programmable gate arrays (FPGAs) are an example of reconfigurable platforms with fine-grained RPUs. Usually the RPUs in an FPGA, also called configurable logic blocks (CLB), contain one or two flip flops to temporarily store data and a lookup table used to implement a boolean function. By organizing these CLBs in a certain fashion, the designer is able to implement any digital circuit (as long as the design can be routed and there are enough resources). FPGA architectures can also contain, for example, distributed multiplier and memory blocks, creating a mix of coarse-grained and fine-grained configurable blocks. These kind of reconfigurable architectures are not only interesting because of their flexibility and their possibilities to exploit parallelism for specific tasks, they also provide a platform for design space exploration to test different architectures without major production costs.

1.2 Compiling for Reconfigurable Computing Platforms

The evolution of silicon technology and the ever increasing complexity of applications has given rise to a need for higher abstraction levels in programming architectures. Since the 1950's raising the abstraction level has been an evolving process as with each improved technology, enabling more complex applications, it became more and more difficult to generate and validate designs. This raise has been seen in both the software and the hardware community. In the software community, complex and diverse architectures resulted in the need for a raise in abstraction level from machine code (different for each architecture) to assembly language (instruction-set dependent) and later followed by even higher level languages such as C (architecture and instruction set independent) because of the ever increasing code complexity. The hardware community faced a similar evolution in the design process. As chips were growing in size it became harder to hand-craft complete designs with the increasing transistor count on chips. To exploit the increasing chip density, designers needed to raise the abstraction level from transistor level design to gate-level design, effectively reducing the amount of hardware components to deal with during the design process. The increase in transistor count did not stop however, resulting in the introduction of hardware description languages (HDL) such as Verilog and VHDL and the logic synthesis tools to synthesize (transform to gates) the design written in a HDL which is comparable to the compiler which translates the high level programming language to a specific target device.

The latest trend in hardware design is electronic system level design (ESL), which moves from behavioral or structural hardware description using an HDL to behavioral system modeling using languages such as C, SystemVerilog and SystemC. Reason for this is the large gap between hardware designers programming in VHDL for FPGAs and software designers programming in, for example, C. Programming an FPGA using a register transfer language such as VHDL can be a time-consuming and error-prone process for someone lacking the skills of experienced hardware designers (and often even for an experienced HDL programmer) [3] [4]. RTL descriptions are usually

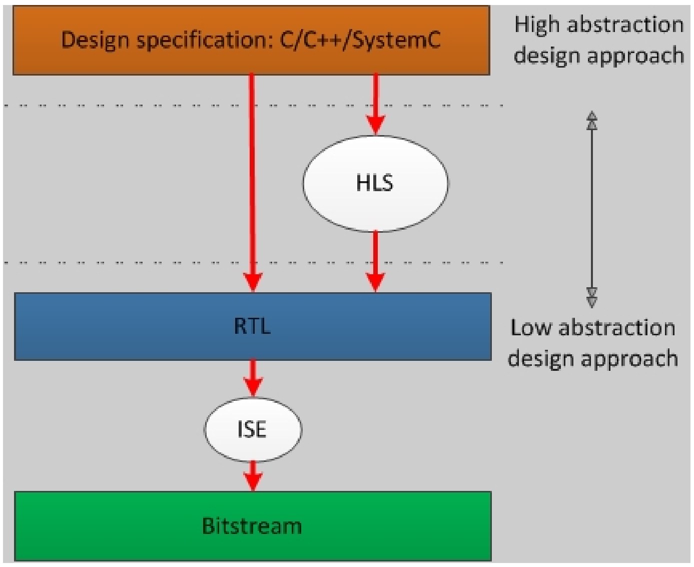

Design specification: C/C++/SystemC
High abstraction design approach
HLS
RTL
Low abstraction design approach
ISE
Bitstream

Figure 1.1: Design flow using HLS

heavily tweaked for a particular device (analogous to assembly language software programming), the cycle by cycle behavior needs to be specified for every register inside the design and because of FPGA hardware flexibility every aspect of the hardware needs to be designed. Moreover, designers need to meet specific timing requirements, often requiring several design iterations to achieve timing closure. All of this makes it hard for an experienced designer to use an FPGA as massive multi-core device to speed up applications, as for software engineers who think in a higher level language and not in hardware it will often be impossible to achieve an efficient design without a significant HDL learning curve. ESL design decreases the gap between software and hardware design and enables co-design and easier reuse of IP cores as cores written in RTL suffer from less flexibility and timing and performance constraints. Figure 1.1 presents the design flow including HLS, giving the designer the possibility to either implement the functional description on FPGA directly using HLS or manually coding using an RTL. The driving force for these developments are the increasing transistor count on chips and the fast-time-to-market driven companies who want to maintain or improve productivity [5].

1.3 High Level Synthesis

Raising the abstraction level from assembly to programming languages like C have enabled software engineers to develop more complex applications with improved productivity. This raise in the abstraction level resulted in languages which are platform independent and follow rules of the human language easing the design process. Hardware design is undergoing a similar process. High level synthesis tools, electronic design automation (EDA) tools supporting ESL design, try to bridge the hardware/software gap by supporting automatic transformations from high level programming models to RTL hardware descriptions, redirecting the time consuming HDL work to the compiler instead of the programmer.

High level synthesis is the process of transforming a behavioral untimed description of an application written in a high level language such as C, C++ or SystemC to a hardware description in a HDL. High level synthesis thus transforms untimed specifications to timed specifications which can then, automatically, be synthesized into gates. The power of high level synthesis is that it enables system-level design space exploration and reduces design and verification efforts. Architectural decisions on the hardware/software design (the partitioning), power consumption, area usage and speed can be made at system level. The same system level specification can be used to explore, and verify, a wide range of architectures on a wide range of different technologies such as a wide variety of FPGAs and ASICs [6]. Another powerful property of high level synthesis is the possibility to introduce behavioral IP cores, which are completely architecture independent, by applying source code transformations and tool specific constraints different architectures can be quickly generated and explored.

This section introduces the evolution of high level synthesis for a historical view, the general compilation flow used in current high level synthesis tools, the common way of specifying a design followed by the concept of interface synthesis and design verification.

1.3.1 Evolution

Compiler technology for high-level languages has been in practice since the 1950s, enabling researchers to use this technology in relation with high level synthesis. The high level synthesis concept started with CMU-DA [7] in the 1970s, before FPGAs started to emerge. The tool used an instruction set processor specification (ISPS) language to specify a design, targeting ASIC design. Many common code-transformations from compiler technology were already used at this time, such as dead-code elimination, constant propagation, sub-expression elimination and code motion. After this first step towards high level synthesis, many HLS tools emerged mainly for research and prototype purposes. Next to academic efforts in HLS tools such as ADAM [8], HAL [9] and MIMOLA [10], the industry started to show interest as well in the 1980s and ealy 1990. An examples of industry efforts is Cathedral-II [11], one of the first domain specific silicon compilers based on the idea that there is no generalized silicon compiler which can perform well for all types of applications. Cathedral focuses on the digital signal processing domain and they use a language, called Silage, optimized for high-level description of signal processing algorithms.

At this point, most HLS tools already decomposed synthesis tasks. Code transformation, module selection, scheduling, datapath and control path generation were typical steps in the synthesis process. Fundamental algorithms for scheduling in high level synthesis were developed during this time, such as list scheduling to solve resource constraint scheduling, force-directed scheduling to optimize resource utilization with a performance constraint and path-based scheduling to optimize conditional branch performance. Although both academic as industrial research was being done to high level synthesis, several generations of tools have failed to be accepted by the industry [12] for several reasons. As RTL synthesis was just emerging, it was not widely accepted yet and schematic design was still the optimal solution. RTL synthesis still had to improve before high level synthesis would become interesting to place in the front-end of RTL synthesis. Although HLS tools were developed prior to good RTL synthesis tools, the HLS tools themselves also lacked performance. Results from the tools were often variable and unpredictable, requiring an intensive learning process to get good results. The lack of formal verification to verify generated designs and the use of new languages increasing the learning curve also resulted in rejection of high level synthesis in the design flow.

When RTL synthesis tools were improved and widely adopted in RTL design flow during the 1990s, high level synthesis would become more practical and semiconductor design companies and EDA vendors would jump in on the high level synthesis effort. Companies such as IBM [13], Philips with the PHIDEO tools [14], Synopsys with the Behavioral Compiler [15], and Cadence Visual Architect based on SYNT [16], started developing and providing HLS tools. Although tools from now on started receiving wide attention, they did not yet replace manual RTL design. Still, designers were relying on manual RTL work for better performance and the use of behavioral HDLs as HLS input languages was not considered popular as it required a steep learning curve and designers were more comfortable with the VHDL/Verilog language.

The progress made in the 1990s influenced the new generation of HLS tools developed by academia and industry since 2000. As a major change, input languages for HLS tools would now include C/C++ or C-like languages such as SystemC to describe the design making the tools more accessible for algorithm and system designers and enabled software/hardware co-design and verification. The use of C-like languages also enabled HLS tools to incorporate the newest compiler technology which had undergone vast improvements in parallelization and optimization techniques.

Besides these advantages of the use of C-like languages, there is still no common language which serves as input for high level synthesis tools and there is even discussion if C-like languages are even suitable for HLS [17] [18]. To address short comings in C/C++, HLS tools introduce language restrictions and extensions to tailor the specification language towards hardware synthesis. Besides language restrictions and extensions to solve input language problems for synthesis, many tools use directives or pragmas to address concurrency (aiding the compiler) in the sequential input specification.

With the improved capacity and speed of state-of-the-art FPGAs, many HLS tools specifically target FPGAs and following the Cathedral approach many tools focus on a specific application domain to improve quality of results. Since 2000, many HLS tools have been designed to target FPGAs, such as C2H [19] from Altera, GAUT [20], ROCCC [21], AutoESL from Xilinx [22], Cadence C-to-Silicon Compiler [23], Forte [24], Catapult-C from Mentor [25] and Synopsys C-Compiler [26]. Current HLS tools solve many of the problems which caused previous generations to fail. The use of high level programming languages such as C/C++ overcomes the learning curve of behavioral HDL and enables reusable portable designs as a large subset of C/C++ is completely portable across HLS tools and the C/C++ language is independent of the target architecture. Many tools have extended their accepted input languages with SystemC, although not in the toolbox of most algorithm designers, which has more advantages in expressing concurrency. Furthermore, tools put effort in interface synthesis solving system integration problems and can make use of DSP blocks and on-chip memories which have vastly increased FPGA performance over the last decade.

1.3.2 General Compilation Flow

The general compilation flow for high level synthesis is shown in figure 1.2. The input to the front-end is a high level language such as C/C++ or SystemC, such languages usually adhere to the imperative programming model and do not explicitly address concurrency. To overcome this programming model problem, the front-end typically transforms the input description into a formal model. This formal model, the intermediate representation (IR), should explicitly address the notion of concurrency which is natural for reconfigurable architectures. Typically, the IR is represented using a control and data flow graph (CDFG), which is an extension to the DFG model which is only capable in addressing data flow. The nodes inside a CDFG are basic blocks of computations without any control statements and thus can be transformed to a DFG. The edges between the nodes represent the control flow. For each basic block, parallelism can be easily extracted by analyzing data dependencies. This is not the case for extracting parallelism across multiple basic blocks by uncovering the data-dependencies. To extract concurrency across basic blocks, compilers need to have the ability to transform a CDFG to other representations such as the hyperblock representation or the hierarchical task graph representation. Although a CDFG seems a good solution to represent concurrency not every high level synthesis tool uses a CDFG as the intermediate representation, extracting concurrency is still a hot research topic and there is no widely adopted standard input language [17] or intermediate representation. However, the front-end of high level synthesis will always involve some behavioral high level language as input and a formal model expressing the concurrency as output.

If the high level synthesis tool permits software/hardware co-design, it is likely that in the middle-end of the compiler decisions should be made upon the partitioning of the hardware/software. That is, by estimation and modeling the user receives feedback from the compiler about how to partition the original application into a software part running on a traditional processor and a part to be accelerated by hardware. Metrics involving the choice of the partitioning might include power consumption, resource utilization and latency of the application. For traditional accelerator generation, from an algorithm description, with a separate design path for the controlling processor this partitioning is ignored and the entire application is transformed into a hardware accelerator. Often the user has control over the applied optimizations, next to some automatic optimizations such as constant folding and subexpression elimination[27], applied in the middle-

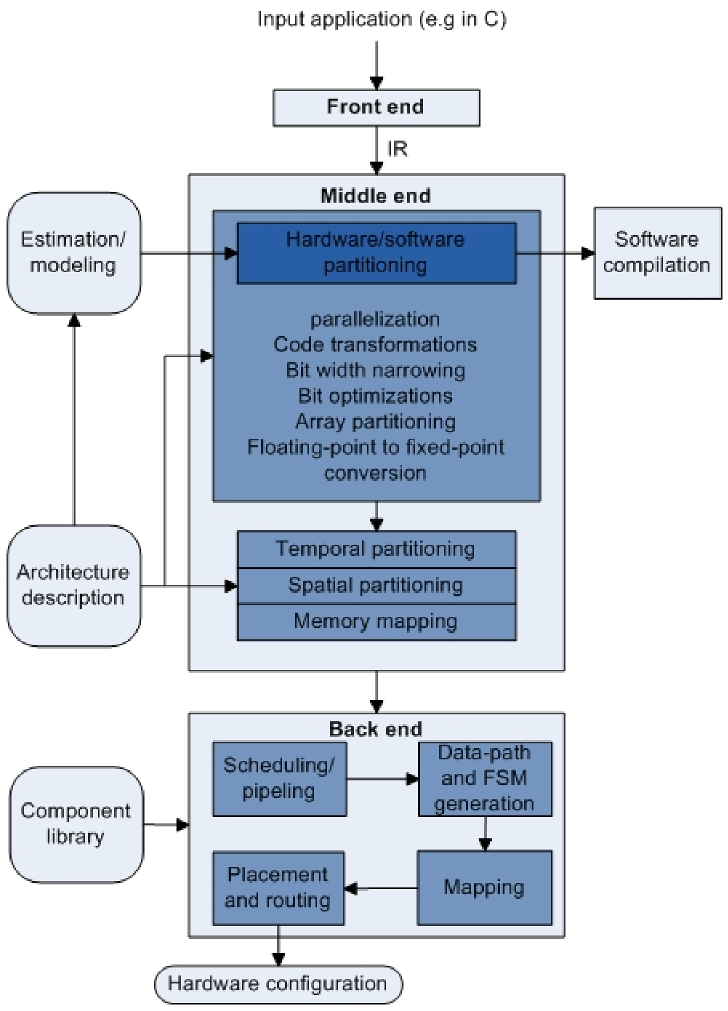


Figure 1.2: Generic compilation flow for high level synthesis [4]

end of the compilation. There are numerous tools which for example enable the user to apply directives to let the compiler apply loop transformations, pipelining and resolving memory collisions by increasing bandwidth (splitting BRAMs or mapping to flip flops). Such optimizations, and others, can also be achieved by manually applying source code transformations, on which will be elaborated in the next section. The basic functionality of the middle-end is to apply spatial and temporal partitioning on the computations in the application. Temporal partitioning refers to isolating computations which should be executed by the same RPU, for example sequentially executing a loop with a multiplication in the body to save area while sacrificing latency (there is only one multiplier). Spatial partitioning refers to isolating computations which should be executed in parallel using multiple RPUs, for example by unrolling a loop with a multiplication in the body to decrease latency while increasing area (since more multipliers are needed). Mapping data structures to memories involves a bandwidth versus area trade-off. Memories have only one or two read/write ports and accesses to memory form a bandwidth bottleneck when many memory operations need to be executed in parallel, in such cases it might be better to bind the data to flip flops as storage units which can be accessed in parallel. The architecture description informs the partition process about the available resources in order to increase the chances for design closure.

The back-end performs the scheduling of operations, effectively distributing the clock cycles among the operations to be executed. The operations are mapped to functional units, the RPUs, and the control FSM and data path are generated. In many cases the high level synthesis tools provide the option to use commercial tools to performs mapping, placement and routing. Most of the tasks involved in high level synthesis are interdependent and the order in which they are executed is not fixed. Each compiler uses their own algorithm to produce a design and this can range from an iterative approach applying small steps at a time or a complex interplay of many tasks all communicating with each other to generate the final design.

1.3.3 Design Specification

The C/C++ and SystemC specifications give the user an extensive set of constructs to write a design specification. High level synthesis poses several restrictions on this input language and adds several extensions which are a direct result of the target architecture. Next to the restrictions and extensions, an untimed specification in C/C++ potentially gives bad performance results when directly used for HLS. Often, several optimization steps need to be made (sometimes as a result of the limitations and extensions of the specification language) in order for HLS to produce an efficient design. The following sections discuss common restrictions/extensions and the common 'tools' provided by the HLS tool to optimize the input specification.

Limitations and Extensions

The most commonly supported input specifications for HLS are C/C++ and SystemC. High level synthesis imposes several restrictions on the supported constructs from the modeling language. Dynamic memory allocations is not supported as a synthesizable hardware implementation must be fully self-contained specifying all required resources. For that same reason, recursion often poses a problem if the recursion can be endless, in such cases tools might be unable to determine the required resources and the latency of the design. Although pointers are often supported, as long as the size of the structure pointed to is known, many tools discuss limitations or pitfalls during synthesis when using pointers such as multi-access pointer use in which a pointer in accessed multiple times in the same function or pointer casting to ensure correct bit-widths in the hardware design.

Next to the restrictions on the design specification language, high level synthesis tools often extend the set of native data types from an input language by supplying a library or directives to use arbitrary precision data types to take advantage of the possibility of arbitrary bit-width operations and signals in RTL which can vastly improve performance. The availability and extend of arbitrary precision integer and fixed point is highly HLS tool dependent as it is not part of the C/C++ specification. The support for floating point is also dependent on the HLS tool and the target technology which needs floating point cores to assign the floating point operations to. In general, HLS tools make their own choice of the provided support with regard to bit accuracy and support for native data types (integer/floating). Next to the scheduling and allocation performance of a high level synthesis, in other words the performance of the generated designs, the availability of functional libraries to support bit-level operations and support multiple data types is a part of high level synthesis tool performance as arbitrary precision and integer/floating implementations can have a big impact on design performance.

Besides the input specification limitations and supported extensions some tools use a specific architecture template, possibly posing limitations to the input specification and performance of the design, to which they map the input specification. The use of multiple clock domains, often only available with a SystemC input specification, streaming support and parallelization techniques are usually dependent on the target architecture template. Although is seems to be a drastic limitation when using architecture templates, it is often a choice by HLS tool designers to be able to generate predictable designs and target specific application domains. These are all design choices which influence tool performance but have a positive effect on predictability and make source-to-source transformations easy as the designer will be able to know in advantage what kind of transformations to apply to improve the design.

Design Optimization

The optimization process of a design using HLS involves source-to-source transformations and the use of directives and tool-libraries to direct the compiler to a specific design. In many cases, directives or architecture settings inform the compiler to pipeline or unroll particular loops in the input specification. Pipelining and unrolling often improves the throughput/latency of a design but in loops with multiple array accesses the pipelining or unrolling will bring bandwidth problems to light which prevent or limit the performance gain of pipelining and unrolling. In such cases, manual source-to-source transformations or directives can be used to enable performance gain of loop unrolling and pipelining. Some tools support automatic array, which are implemented in BRAM, partitioning or reshaping to increase bandwidth while in other cases the tool supports a certain coding style (possibly using tool libraries) to ensure an increase in bandwidth. Architecture settings, coding styles or pragma's/directives can be available to map operations to specific functional units, limit the number of used functional units at the cost of latency and map variables to BRAM with a certain amount of read/write ports or simply to registers.

The optimization process usually focuses on the available bandwidth. The compiler will take care of pipelining and unrolling, informing the designer about limiting factors in the design such as multiple BRAM accesses which need to be spread out across multiple clock cycles due to limited resources. It is up to the user to either increase bandwidth, by use of directives or a different coding style, or decrease the number of BRAM accesses by recoding the input specification (particularly useful when aiming for a streaming implementation). Each tool has its own approach to design optimization, from requiring many source transformations by the user to source transformations in combination with directives to take advantage of the transformations and to fully automated input specification optimization requiring the user to inform the compiler to partition, pipeline or unroll parts of the input specification.

As with manual RTL design, there's an area/latency trade-off for the designer to take into account. With an increasing unroll factor, more resources and bandwidth is needed to execute operations in parallel. Many factors influence the design performance and the difficulty of design optimization, such as the pipeline initiation interval parameter, the unroll factor, the difficulty of creating more bandwidth by packing data elements into wider vectors or distributing them across multiple storage elements, parameters to limit the use of certain functional units, streaming/interface support and the use of arbitrary precision types. All the discussed optimizations are summarized and combined in an optimization process in [28].

1.3.4 Basic Source Code Transformations

As mentioned in the previous section, it is up to the compiler to extract concurrency from the input program. Since the input is written according to the imperative programming model the compiler might not be able to extract all the possible parallelism from the input, since writing sequential programs is quiet different from explicitly specifying concurrency. Depending on the high level synthesis tool of choice, although it might even hold for every tool currently available, the coding style of the input program can drastically influence the end result of the generated design [29] [30]. Raising the abstraction level from HDLs like VHDL or Verilog to HLLs like C and SystemC does not mean the designer can ignore hardware completely. The designer still needs to be aware of the underlying hardware to understand the proper coding style to achieve a particular architecture. Some examples follow to illustrate this important fact of high level synthesis, these examples are taken from the High-Level Synthesis Blue Book written by Michael Fingeroff[31]. The book illustrates coding styles to obtain certain architectures, including the area/latency trade-offs which come with the design choices. Only two small examples are presented, as it is outside the scope of this report to elaborate on the vast amount of possible trade-offs in the coding style.

Example 1: Bit accuracy

When targeting coarse grained reconfigurable architectures containing programming arrays of 32-bit ALUs or general purpose processors with standard 8, 16 or 32-bit architectures, the bit-width of variables may not affect performance. When targeting fine-grained architectures such as FPGAs, the use of native types in C/C++ will drastically affect performance. The size and delay of functional units (adders, barrel shifters, floating point cores) can be substantially reduced by using bit accurate data types, in contrast to general purpose processors who's functional units cannot be customized. The control logic, the generated FSM, can also be substantially reduced by correct use of bit accurate types. While many compilers are able to extract bit-width information from the input code, there are situations where the user has to inform the compiler about properties of data types.

Listing 1.1 shows a code example for an accumulator, including a control variable in the interface to control the number of accumulations. As there exists a static upper loop bound, the loop counter in the control hardware is fixed to three bits. The control variable 'ctrl' on the interface however consists of 32 bits and the compiler will not be able to reduce the number of bits as the variable is on the control interface and theoretically can take up any value up to 32 bits. High level synthesis only reduces bit-width if it can prove that it is possible without changing the functionality. The left side of figure 1.3 shows an approximation of the resulting hardware after synthesis, it can be seen that the generated control logic (FSM) includes a 33-bit subtracter to test the conditional statement in the loop resulting in a larger area than required.

```
void accumulate( int din[4], int &dout, int ctrl ){
        int acc = 0;
        ACCUM: for(int i = 0; i < 4; i++){
                acc += din[i];
                if(i >= ctrl-1)
                        break;
        }
        dout = acc;
}
```

Listing 1.1: simple accumulate loop with control

Not only is it possible to reduce the bit-width of the subtracter, a minor code change can also eliminate the subtracter completely. Listing 1.2 presents this solution. Because the user puts a constraint on the interface variable 'ctrl', only a 4-bit subtracter would be needed if the conditional check would be done as in listing 1.1. By using a comparison between the current loop iteration and the previous loop iteration, only a 3-bit comparator would be needed. The right side of figure 1.3 shows an approximation of the resulting hardware after synthesis. For a small example such as this, area will not be reduced drastically, but such minor changes will result in a noticeable decrease in area when synthesizing larger applications.

```
void accumulate( int din[4], int &dout, ac_int<3, false> ctrl ){
        int acc = 0;
        int i_old = 0;
        ACCUM: for(int i = 0; i < 4; i++){
                if(i_old == ctrl)
                        break;
                acc += din[i];
                i_old = i;
        }
        dout = acc;
}
```

Listing 1.2: optimized accumulate for simplified control hardware

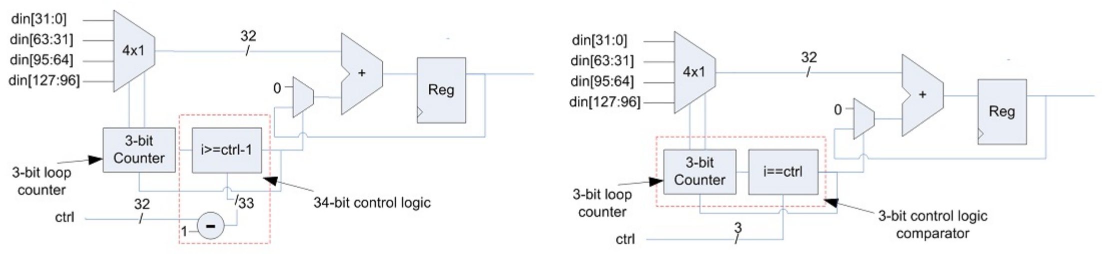


Figure 1.3: Generated architecture of listing 1.1 (left) and listing 1.2 (right)

Example 2: Pipelining and unrolling of loops in combination with memory accesses

Pipelining is a well known approach to exploit instruction level parallelism (ILP). By dividing an operation in several steps and creating a pipeline, different parts of the operation can be executed simultaneously and thus increasing throughput at the cost of area. There are various forms of pipelining such as resource pipelining, memory access pipelining and loop pipelining. Memory access pipelining is less interesting for FPGA as accessing a BRAM takes just one clock cycle. Resource pipelining involves the creation of functional units which usually have a large latency such a multipliers and dividers, some tools offer the possibility to create constraints on the created multipliers such as the number of stages within the unit. This is however not something which can be controlled with the source code. Loop pipelining, and function pipelining, allows multiple loop iterations to be overlapped, increasing throughput.

Loop unrolling is a common practice to increase parallelism within a design [32]. By decreasing loop overhead and increasing ILP opportunities, the scheduler is able to schedule more operations in parallel at the cost of area. Loop unrolling differs from pipelining in the sense that a fully unrolled loop can, theoretically, be executed in a single clock cycle whereas pipelining a loop, again theoretically, results in a start of a new loop iteration each clock cycle and pushing instructions through the pipeline just like water flows through a pipe.

The performance increase of both these optimization methods depends heavily on the loop body, in particular the memory accesses within a loop. There are many possible ways to code an application in a way that memory accesses prevent a gain in performance when using loop pipelining or unrolling. The fundamental problem however is always the same, if one tries to access the same memory multiple times in the same clock cycle the scheduler needs to insert an extra clock cycle to separate these memory accesses. Listing 1.3 gives a simple example to illustrate the problem.

```
void accumulate( int din[4], int &dout, bool flag[4] ){
        int acc = 0;
        if( flag[0] )
                acc += din[0];
        if( flag[1] )
                acc +=din[1];
        if( flag[2] )
                acc += din[2];
        if( flag[3] )
                acc += din[3];
        dout = acc;
}
```

Listing 1.3: accumulate with memory access bottleneck

The body of this function is basically a fully unrolled loop. If 'din' is mapped to a BRAM without any further optimizations, each element of 'din' will be placed in a single entry in the BRAM. Accessing 4 entries in a BRAM in the same clock cycle is not possible, so the expected parallelism will not be achieved. To overcome this problem, one might try to pipeline the function (the top level function can be seen as a main loop) to speed up the design and start a new iteration of the accumulator each cycle. Again, the memory access to 'din' prevents pipelining because it is impossible to schedule multiple reads in a single cycle. Listing 1.4 presents a possible code transformation to overcome the above mentioned problems. An internal copy of the input data is created. By completely unrolling the DIN loop and mapping the input data to a single entry in a BRAM or separate registers it is possible to read the input data in a single cycle. The ACCUM loop can now be fully unrolled to compute the sum of the elements in a single cycle, assuming the delay of the adders does not exceed the clock cycle time. The latency of the design has been reduced from 6 cycles to 4 cycles in this solution, moreover it is now also possible to pipeline the accumulate function. Note that it is assumed all other variables are stored in internal registers instead of BRAMs.

```
void accumulate(int din[4], int &dout, bool flag[4]){
        int acc=0;
        int din_int[4];
        DIN:for(int i=0;i<4;i++)
                din_int[i] = din[i];
        ACCUM:for(int i=0;i<4;i++){
                if(flag[i])
                acc += din_int[i];
        }
        dout = acc;
}
```

Listing 1.4: accumulate with faster memory access pattern

This example only highlights a small portion of the memory access problem, and only presents one of many solutions to this particular problem. Other solutions involve the removal of conditional statements, reshaping of memories (changing bit-widths of the entries) or partitioning a single memory into multiple memories to increase access possibilities. Although the memory access problem is a general problem because of the hardware implementation of memories in an FPGA, the solutions necessary to increase performance might differ among high level synthesis tools depending on the influence the user has on the mapping and resource choices.

1.3.5 Interface Synthesis

High level coding languages such as C make it easy to connect multiple blocks together to create a complete algorithm or application. The interface between these modules is automatically generated during synthesis by extracting the information from the function parameters, depending on the HLS tool capability the user can have an influence on synthesized interface between modules and the between the design and its environment. It is up to the designer to set constraints on the interface between modules by adding synchronization variables, such as start/stop/reset wires, or mapping the data to FIFOs or BRAMs. The interface to the environment is also an important design aspect, if the high level synthesis tool or environment only supports a single slow interface it is not useful to optimize the latency beyond the interface bandwidth bottleneck. The positive aspect of having untimed C as input for the high level synthesis is that it leaves the tool completely free to create a top level interface for the design. The amount of possible interfaces is dependent on the tool of choice, and it is up to the compiler to generate the proper streaming buffers, memories and handshaking. It is important that the designer recognizes the fact that designing an accelerator makes no sense if the designer puts no thought in the interface, as eventually the accelerator will have to be controlled by an external processor and have its data fed by some external source. Although not every HLS tool supports general interface protocols, it is possible to write a top level interface wrappers using a HDL which wraps around the generated design.

1.3.6 Verification

One of the time consuming tasks when designing an accelerator using a HDL such as VHDL is the verification. Tools such as ModelSim (Mentor) and Vsim (Xilinx) exist to help the designer validate the design. It is up to the designer to validate the cycle-by-cycle behavior of the design by either verifying the generated waveform showing internal and external values of the design or by implementing a test-bench supplying the design with data and verifying the output data. Care has to be taken that all possible inputs to the design are tested to ensure correct behavior. For high level synthesis to be successful, it is important that it overcomes the cumbersome task of verifying the design 'by hand'. There exist many different approaches to solve the verification problem, from supporting down-flow verification tools such as ModelSim and Vsim to simulate the design using automatically generated test-benches based on the C/C++ testbench used to verify the functional description, to intermediate SystemC models to simulate cycle-accurate behavior and formal verification using model checking methods.

1.4 Domain Specific Approach

Based on the lessons learned from decades of research, the research community is taking the domain-specific approach towards parallel computing [33] [34]. This approach explores patterns within, and across, application domains. The goal is to provide an easy way to write applications which can execute on highly parallel architectures. To avoid the programmer to have vast knowledge of the underlying hardware, programming models should be high-level and separate the programmer from the hardware itself while still supporting parallelism. The main goal is to make parallel computing productive, energy efficient, scalable, portable, performant and verifiable. Applications often contain naturally parallel parts and hardware is naturally parallel. Now we have access to multi-core, many-core and reconfigurable highly parallel platforms there is a need to simplify the programming model to efficiently program these parallel platforms. The high level synthesis section already indicated that raising the abstraction level to a high level language such as C does not mean the designer can ignore the underlying hardware. High level synthesis does speed up the design trajectory, but still requires that the designer is able to reason about what is going on in the underlying synthesis steps. Having an even higher abstraction level, by introducing a framework on top of high level synthesis might be a good solution to enable the software engineer to develop applications without understanding details of the underlying platform. Many applications contain specific patterns which can be mapped to parallel architectures in many different ways, depending on the underlying platform and the aimed performance. An application is a combination of many different computational patterns. Letting the software engineer develop applications in a functional manner, without having to apply code transformations to efficiently implement the application on hardware, might even further increase productivity and re-usability. The domain of interest, the image processing domain, is explained next as it will be the application domain used in this research.

1.4.1 Image Processing Domain and Skeletons

Modern day image processing applications become more and more computationally intensive, requiring special purpose hardware solutions to be created. The image processing domain is a hot research topic, especially within the parallel computing community as enormous speedup can be achieved by parallelizing such applications and implementing them on reconfigurable architectures maintaining the advantages of custom hardware solutions while enabling fast design space exploration (using HLS) and re-programmability. According to [35] most image processing applications can be divided and categorized into three different IP task categories:

- Pixel to pixel operations: an operation is performed on a single pixel in the input image to produce a single pixel in the output image.
- Neighborhood to pixel operations: the operation is performed on multiple input pixels to produce a single output pixel.
- Global operations: operations done on the complete image to produce either a single scalar value such as the maximum, or a vector such as a histogram.

Although this is a very global class definition, many image processing tasks can be easily recognized to fit in one of the three. The ease of distinguishing image processing tasks into IP classes makes the image processing domain an interesting research domain with regard to high level synthesis and identifying common code structures for certain IP classes. Skeletons can be introduced to capture certain memory access patterns such as neighborhood access or single-pixel access and the re-use of accessed data across several iterations of the operator performed on the data. Such a skeleton classification can be used to instantiate a particular parametrized code structure (theoretically in any imaginable programming language) while maintaining the functionality of the IP core which has been classified. Essentially, a high level coding language as C/C++, which is sequential, can be classified and transformed into a parallel description for a specific platform while maintaining functionality. Care has to be taken however that the used skeleton classification does not contain as much skeletons as there are image processing kernels (i.e. each kernel is a skeleton), while at the same time ensuring there are enough skeletons to capture all possible patterns of memory accesses such that each processing kernel can be classified.

Related to high level synthesis, a skeleton classification might include a class for window-based operations such as an erosion kernel removing noise from an image. High level synthesis requires a certain coding style, or a certain memory access pattern in the C-code, to generate a specific architecture. Using parameters as the image size, window size and a certain parallelization factor, the skeleton could automatically be instantiated by replacing the parameters in the skeleton code (the certain coding style for particular architectures) with the parameters available with the kernel classification.

1.5 Problem Description, Contributions and Overview

In figure 1.4, several design flows for the implementation of an algorithm on FPGA are shown. The current standard in design for FPGA is manually coding the application using RTL such as Verilog or VHDL, which is becoming more and more difficult and time consuming with technology scaling and increasing application complexity. This approach requires a designer to manually implement the algorithm using an RTL language, often based on a reference functional description (possibly in C/C++/SystemC) which is used to verify the correctness of the algorithm.

Using high level, behavioral, synthesis the RTL can be generated directly from the functional description, drastically reducing the design effort. However, the sequential nature of functional descriptions in C/C++ potentially results in bad performance of the generated design by HLS and the designer has to put effort into optimizing the functional description. These optimizations involve structural changes to separate the functional description in smaller computational kernels, optimizing and re-ordering memory accesses for parallelization, selecting internal memory/vector widths and sizes and tune interfaces between each computational kernel. The figure represents this optimized description as the optimized C' specification. *The main question is if state-of-the-art high level synthesis is capable of competing with manual RTL designs and if source transformations at C/C++ level are sufficient enough to produce good quality of results.* If high level synthesis produces good quality of results, the next step in the design process is to alleviate the designer from manually optimizing the design specification. *If the source transformations result in good HLS quality of results, algorithmic skeletons, based on common patterns in the embedded vision application domain, can be introduced to enable automatic code optimizations/transformations to generate*

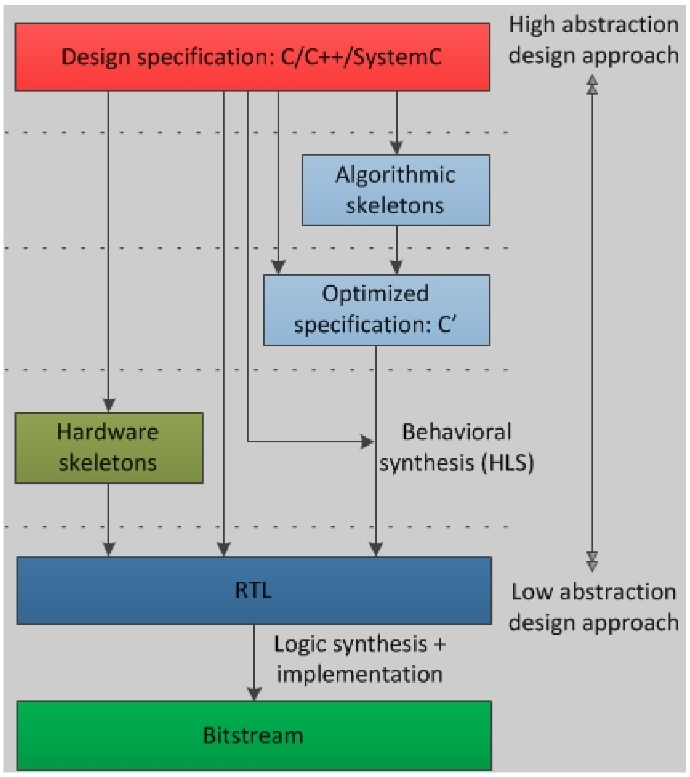

Design specification: C/C++/SystemC
High abstraction design approach
Algorithmic skeletons
Optimized specification: C'
Hardware skeletons
Behavioral synthesis (HLS)
RTL
Low abstraction design approach
Logic synthesis + implementation
Bitstream

Figure 1.4: Design trajectories for FPGA implementation

C' level descriptions. The algorithmic skeleton, based on a certain algorithmic classification, describes a structural body representing the datapath to be generated by HLS. By classifying parts of the design specification, algorithmic skeletons can be instantiated automatically to produce C/C++ resulting in good quality of results using HLS.

If the HLS quality of results is poor compared to manual design, another design flow remains a possibility. The hardware skeletons design path is at a lower abstraction level, closer to the hardware, and might produce better results. Although not part of this thesis, this path is mentioned for completeness. Just as with algorithmic (software) skeletons, hardware skeletons describe the structural body for a specific architecture and a front-end needs to take care of instantiating the skeleton with the proper parameters and operations to generate a computational kernel. The difference with algorithmic software skeletons is that hardware skeletons describe the structural body in an HDL such as VHDL or Verilog, being a lower abstraction level closer to the logic synthesis back-end. It is well-known that the lower the abstraction level, the better the performance can be but the more time consuming the design trajectory usually is.

The contributions of this work include the following:

- Benchmark of two small image processing kernels using two HLS tools, with as goal to mimic two RTL designs, which are common in handwritten RTL design, and observe performance metrics between different HLS tools and handwritten RTL.
- Benchmark of an application closely matching a handwritten RTL implementation observing HLS performance and highlighting issues.
- Implementation of a scope of designs for an application using two HLS tools to observe HLS performance.

- A design space exploration on two image processing kernels, showing parameterization of HLS input specifications and achievable scope and scalability of designs.
- An introduction towards algorithmic skeletons for HLS to shorten design time by enabling automatic source-to-source transformations for high quality of results.
- Coding styles for high level synthesis.

To analyze HLS performance, two small image processing kernels are ported to FPGA using two promising state-of-the-art HLS tools. This analysis serves as means to observe hardware generation in relation to the input specification, i.e. how to optimize the source code for good QoR. A complete application is ported to FPGA using the same HLS tools to analyze the performance (area, latency, design time). This complete application is required to use HLS for what it is intended for, the automatic generation of RTL for complex or large applications without the need for manual RTL coding. This HLS analysis, in combination with a design space exploration performed on two small kernels, will show several coding styles for good and predictable QoR. These coding styles, in combination with an algorithm classification, are then used to reason about the possibility for the introduction of algorithmic skeletons for HLS to decrease design time.

Acceptable performance metrics for the acceptance of HLS in the design flow are hard to express and are dependent on the designer's requirements. This work will show however that significant code transformations are required for good QoR, requiring hardware knowledge by the designer. For this reason, algorithmic skeletons are proposed and it is vital that coding styles can be related to a specific algorithm classification. This will be shown in the final part of this thesis.

This thesis will first focus on the HLS quality of results compared to manual RTL design, based on an in-depth datapath analysis of two small image processing kernels followed by a case study in which an algorithm is mapped to FPGA and compared to a reference manual implementation. Chapter 2 introduces related work on all design trajectories visualized in figure 1.4 and chapter 3 introduces the tools and platform used throughout this work. Chapters 4 and 5 discuss high level synthesis performance based on an in-depth comparison of two small computational kernels and a complete algorithm. In chapter 6, the process and power of design space exploration using high level synthesis is described. Chapter 7 introduces algorithmic skeletons for HLS based on conclusions from the high level synthesis performance investigation. Finally, chapter 8 presents an overall conclusion and future work of this thesis.

Chapter 2

Related Work

This work covers a large scope, ranging from algorithmic skeletons and it's classification to back-end RTL implementation. Although this research does not focus on creating a new algorithmic classification or improving high level synthesis tools itself, this chapter will present related work done in the entire scope of the design flows as shown in figure 1.4. Section 2.1 introduces related research done in the pattern based approach field for reasoning about and implementing for parallel computing platforms, section 2.2 discusses related work to each of the design flows in figure 1.4 and finally section 2.3 presents multiple branches of HLS related research.

2.1 Pattern Based Approach

The pattern based approach involves pattern matching and recognition during synthesis, referred to as pattern mining in [36]. Pattern mining does not focus on a specific domain, but tries to extract patterns across application domains. The following sections present related work done related to algorithmic classes, pattern and idioms. In general, these definitions are coarse-grained, used to reason about parallel computing rather then moving towards a higher implementation abstraction level. Most of these classifications do talk about memory access patterns, but refer to multiple applications domains and do not refer to specific parallel architectures but reason about the concept of parallel architectures in general.

2.1.1 Dwarfs

Berkeley has introduced 'Dwarfs' in a try to raising the abstraction level from standard applications to patterns of computation [33]. They reason that one of the biggest obstacles in parallel computing is the unclarity in how to express parallel computation in the best way. They claim it is unwise to let current benchmark applications, based on an imperative programming model, drive an investigation to parallel computing. The introduction of dwarfs, which capture certain patterns of computation and communication common to important applications, lets them reason about hardware requirements instead of focusing on individual applications and how to map them to certain hardware platforms. The claim is that the design community should focus on higher level abstractions instead of focusing on parallelizing legacy applications based on the imperative program model. The work of Berkeley is inspired by the definition of 7 particular computational methods by Phil Colella [37]. Next to these 7 dwarfs, Berkeley extends this list with 6 more dwarfs they feel are not included yet. The dwarfs do not identify parallelism but they identify computational and communication classes which are regarded as important for current and future applications such as database software, computer graphics, graph traversal and linear algebra. As applications in most cases will consist of a combination of several dwarfs, focus is not on the performance of individual dwarfs but on the composition of them.

2.1.2 Idioms

An idiom is a pattern of computation that a user may expect to occur frequently in certain applications. They have been used to classify which portions of C-code can benefit from being ported to hardware accelerators such as GPUs [38]. In [39] several common idioms are discussed such as a stream idiom describing the sequentially reading and writing of arrays in C-code. Other examples are so called gather and scatter idioms which describe the gathering of data in a random access order in memory and writing data to memory in a random order respectively. These idioms are very coarse-grained and besides indicating regions of C-code which could potentially benefit from parallelization, they do not describe or include properties which can be used for automatic code parallelization.

2.1.3 Algorithm Classifications

There are many variations of algorithm classifications used for code generation, reasoning, analysis and performance prediction, examples of which can be found in [40], [41], [42], [43] and [44]. This work will not focus on developing a new classification but will make use of an internally developed modular and parameterizable algorithmic classification [45]. This work includes a comparison with existing work as well as a new classification including well-defined grammer and vocabulary. The main advantages of this classification are the fine granularity and modularity to support construction of classes by use of the existing grammar and vocabulary.

The used classification is not final and the completeness and correctness of the classification is researched with the use of an internally developed source-to-source compiler called Bones [46]. The compiler currently generates high-performance CUDA and OpenCL code using the algorithmic skeleton technique. The goal is to generate code based on skeletons of parallel structures, or simply a library of parameterizable implementations for a particular target and algorithm class. It is already shown that the tool delivers better performance compared to other C-to-CUDA tools for a small set of image processing kernels. The tool is build in a way that makes it easy to add a target architecture such as an FPGA. The tool generates readable code allowing fine-tuning after code generation and requires minor modifications to the original C-code. Future work includes the implementation of more skeletons (and more target architectures), fine-tuning existing skeletons, applying kernel fusion of two algorithm classes if its gains performance and automatically identifying algorithm classes from the source code relieving the programmer of classifying the input code.

2.2 FPGA Implementation Approaches for the Image Processing Domain

There are several possible approaches to get to an FPGA implementation of an image processing algorithm. The, up till now, usual approach is the manual approach. Perhaps high level synthesis will be able to match the performance of the manual work. For high level synthesis there are still several possible design trajectories, such as manual C optimization to aim for a particular performance/architecture, parameterizable skeletons as input for the high level synthesis removing the need for manual code optimizations or hardware skeletons to directly go to an RTL implementation. The following sections explain these design trajectories, elaborating on existing work being done with regard to the design trajectories introduced in figure 1.4.

2.2.1 Manual RTL Implementation

Manually implementing algorithms on reconfigurable platforms has for long been the usual design approach. Although manually implementing RTL is a time-consuming and error-prone approach, it is possible to design high performance accelerators by completely specifying the architecture of choice. Experienced digital hardware designers create their design at gate-level using hard description languages (HDLs), while less experienced designers can make use of a module library with common interfaces to compose their design [47]. A lot of research has been to done in optimizing very specific algorithms and doing design space exploration for manual RTL implementations, such as can be found in [48] [49]. In the image processing domain, Donald Bailey is one of the leading researchers as group leader of the image and signal processing research group at Massey university. His work involves algorithm transformation for FPGA implementation [50] [51], efficient vision algorithms on FPGA [52] [53] [54] and the introduction of design patterns to overcome mapping problems due to processing mode dependent parallelization [55]. This last work is already a step towards pattern matching, limiting the number of design patterns applicable to the porting of an algorithm to FPGA based on a processing mode such as streaming, offline (data to operate on is available in memory) or a hybrid form.

2.2.2 Straightforward High Level Synthesis

Doing efficient high level synthesis, using either a commercial or open source HLS tool, is not as easy as just pushing a button to generate code. As is explained in section 1.3, the coding style of the application can significantly influence the performance of the generated design. The straightforward high level synthesis approach does not imply an efficient design and requires recoding to be done depending on the performance goal and the HLS tool of choice. Such code modifications might not be a problem for someone who is experienced with FPGA RTL design or has general knowledge of processor architectures, as they are assumed be able to find performance bottlenecks of C/C++ input code such as memory access collisions. For someone working in the software domain it might be harder to come up with an efficient design by re-coding the application for efficient hardware generation. In essence, sequentially written programs for a CPU will generate bad results in high level synthesis. This is not a problem of high level synthesis however, as it simply schedules operations based on the input specification which essentially describes the resulting architecture to be generated. This design approach is normally not acceptable, unless an algorithm's portability needs to be verified or there are no design constraints at all (proof of concept designs).

2.2.3 Algorithmic Software Skeletons for High Level Synthesis

The research of Wouter Caarls addresses the domain specific approach, in particular the embedded image processing domain [40]. The aim of the research is to bring the programming language closer to the algorithm instead of the architecture. Porting applications to new architectures is a time-consuming task if the application is programmed in a general-purpose language or an architecture specific language. Architecture independence is claimed to be resolved by the use of algorithm-specific languages (ASLs). The execution models of general-purpose languages such as C is sequential. Although it is possible to compile the application to different architectures, it still involves a restricted set of architectures adhering to the sequential execution model. Porting such applications to parallel architectures requires significant changes to the application/algorithm. Languages enabling writing parallel applications such as OpenMP, MPI and Handel-C are available but still pose restrictions on certain hardware features which have to be available and their performance differs among architectures. Algorithmic skeletons are introduced, which are higher-order functions written in an algorithm-specific language. These skeletons are then source-to-source translated (compiled) to an operation written in a target-specific language.

Input to a skeleton is a kernel definition, specifying inputs and outputs and the operation to be performed. The skeleton itself specifies the structure of the algorithm, for example a neighborhood to pixel operation or pixel to pixel transformation, directly classifying memory access patterns and data re-use patterns. The skeleton can be compiled to different architectures, generating different implementations depending on the target architecture. Code has to be generated in a specific structure to make optimal use of the target architecture, an interface has to be generated to, for example, make use of the target's memory organization and the kernel language has to be transformed to the language accepted by the compiler for the target architecture. The definition of algorithmic skeletons is abstracted at a lower level than the dwarf specification of Berkely. While dwarfs focus on applications, algorithmic skeletons focus on different patterns of communication and computation of an image processing algorithm and already focus on a general architecture independent language. The classification of skeletons is thus finer grained than the classification of the 13 dwarfs in previous section.

As discussed in section 2.1.3, the work of Cedric Nugteren addresses algorithmic skeletons for automatic parallelization to a parallel architecture using the Bones source-to-source compiler. The classification is finer-grained than the classification proposed by Wouter Caarls, furthermore the Bones source-to-source compiler has been released to encourage research on the algorithmic classification and the implementation of skeletons for different target architectures. As an FPGA's architecture is customizable and finer-grained than the architecture of GPUs, different skeleton implementations will be required. Analysis of coding styles for certain processing kernels related to the algorithmic specification will be done to reason about the completeness of the classification and the applicability of the classification to an FPGA based architecture.

2.2.4 Hardware Skeletons for Automatic RTL Generation

Besides having algorithmic software skeletons, research has also been done to the creation of hardware skeletons for automatic RTL generation. In [35] and [56], hardware skeletons are introduced. It is stated that behavioral synthesis tools might perform better in this last decade, but having more structural design information often still results in designs which are smaller and faster. The work presents a framework for developing efficient architectures for image processing applications, still representing the framework in a high level programming model without significantly sacrificing performance. Hardware skeletons are a parameterized description of an algorithm-specific architecture. Parameters can be values, functions or other skeletons to compose a higher level skeleton. Implementations of a skeleton can be generated for specific target architectures. There exists a library of hardware skeletons, each with a set of different implementations such as bit-serial and bit-parallel architectures. The library consists of 3 layers, an arithmetic core library at the lowest level, followed by a basic image operation library at the middle level and high level compound skeletons at the highest level. Rule based optimizations are applied for core-specific optimizations, often not performed by HLS tools. This approach differs from algorithmic software skeletons, as with hardware skeletons target-specific and a multiple of implementations are already available. It is up to the designer to compose the application from a set of hardware skeletons to come up with a design achieving the performance constraints.

2.2.5 Alternative Approaches

New programming models such as OpenMP, OpenCL and CUDA have been introduced to explicitly code coarse-grain parallelism to overcome the sequential semantics of languages such as C. FCUDA [57] is an FPGA design flow included in the multilevel granularity parallelism synthesis framework [58]. FCUDA performs source-to-source transformations on CUDA thread blocks, exposing coarse and fine grained parallelism, to generate C code which can be parallelized by the high level synthesis tool AutoESL. Using such a tool flow can be used to avoid reprogramming the initial C source code to be used across different platforms such as GPUs and FPGAs, but programming in CUDA requires knowledge of GPU hardware to achieve efficient results. OpenCL is a framework for task and data-based parallel computing on heterogeneous platforms consisting of CPUs and GPUs. OpenCL, as does CUDA, enables programmers to use the GPU for general purpose processing. Current research is done to OpenCL for FPGA and Altera has recently announced the first OpenCL program for FPGAs. The OpenCL distinction between host code and kernel code to be accelerated makes it possible for system designers to pick kernels which have to be accelerated in the FPGA. The host code is programmed with a standard programming language like C and the kernels are programmed using the parallel computing language OpenCL.

The advantage of using OpenCL for FPGA is that it is already focusing on parallel computing, explicitly expressing parallelism to the compiler for better performance results.

2.3 Related Work in High Level Synthesis

The newest generation of HLS tools have resulted in a wide interest in high level synthesis. Much research is done in order to improve performance of high level synthesis. In [59] it is observed that downstream power saving features such as clock-gating are not known during high level synthesis. An automated solution is presented to facilitate clock-gating at a number of granularities at a high level C description, including high level synthesis in the process of power optimization. In [60] an RTL power estimator for FPGAs is presented in combination with a low power high level synthesis system showing reduced power consumption compared to Synopsys Behavioral Compiler by introducing two power consumption reduction algorithms. Next to power aware scheduling, research is still being done in other objective-aware scheduling such as [61] introducing a new technique to perform interconnect-sensitive synthesis to reduce communication and multiplexer cost. [62] presents a technique for multi-objective optimization during high level synthesis to optimize power, area and delay simultaneously. The design space to be explored is vast and a weighted sum algorithm using a graded penalty cost function is used to provide the user with a large number of alternative datapath designs all meeting the user design constraints.

Because of increasing design complexity, enabled by technology scaling, high level synthesis can be used to solve the complexity problem. The routability problem due to technology scaling can potentially be solved by HLS as well according to [63]. They show that many interconnect problems can be avoided by adopting a layout friendly architecture generated by high level synthesis tools. Next to standard HLS tool estimates such as resource utilization, clock frequency and latency they introduce metrics which can be used to estimate routability impact of design decisions during high level synthesis design space exploration.

Next to research being done in further improving high level synthesis performance in different area's, there's also a branch of research being done on high level synthesis performance analysis to see if we are at a turning point to start adopting the HLS design methodology. High level synthesis performance is for example analyzed in [64] [65] based on a sphere decoder case study, [66] based on a stereo matching case study, [67] based on several case studies such as edge detection and matrix multiplication and in [68] by BDTI who do independent technology analysis and have a certification program for high level synthesis tools. With the increased interest in high level synthesis, [69] argues for a benchmark set for high level synthesis and [70] argues for the tipping point of high level synthesis explaining the need for high level synthesis and the reasons for its near acceptance.

There is a vast history of research being done to code optimizations, specifically with regard to the optimization of the memory hierarchy and required bandwidth by the application. Minimizing the required bandwidth by improving data locality, or improving available parallelism, is usually achieved by applying loop transformations, either manually or by the compiler [71] [72]. For code optimizations of applications to run on a CPU, the polyhedral model is a well known tool for efficient code generation by applying code transformations to the polyhedral representation and then generating code cohering to that polyhedral representation, an optimization flow often combined with existing compilers such as Open64 [73] [74] [75]. Loop transformation techniques often aim at a particular goal with a particular cost function, [76] identifies and evaluates different cost components and the trade-offs among them to direct the loop transformations to a particular platform with a certain datapath and memory hierarchy. In the field of high level synthesis, recognizing the importance of platform-dependent modeling, researchers also notice the importance of loop transformations for bandwidth and locality optimizations [77] [78] [67].

The topics in this work differ with previous mentioned work in the following ways:

- Previous work tends to focus on one HLS tool in particular whereas this work cross compares results between two state-of-the-art HLS tools and comparable manual designs.

- The commercially available HLS tool AutoESL, used in numerous HLS performance papers, is now part of Xilinx and has been updated numerous times already. The HLS tool has also been incorporated in their newest design suite, Vivado, which is an IP and system-centric design environment including all aspects of design integration and implementation. The commercially available HLS tool Synphony C-Compiler, also used for experiments, has also experienced some revisions, ensuring a performance research of state-of-the-art tools.
- An in-depth analysis is performed with regard to the possible spectrum of synthesizable architectures, exploring the scope of design space exploration and limiting factors of the design space.
- Extensive datapath analysis for small computational kernels is included in this work to search for possible improvements or drawbacks in high level synthesis, not limited to reporting the percentage of area and design time gain.
- Ideas are presented to raise the abstraction level to algorithmic skeletons to further ease the design flow.

Chapter 3

Methodology

This chapter describes the methodology, the HLS tools, the platform and the application used in the experiments. Section 3.1 shortly introduces the platform used for verifying the correctness of designs and the FPGA device used as target for logic synthesis, section 3.2 shortly discusses the logic synthesis and implementation tool of choice. In section 3.3, the high level synthesis tools used in this research are introduced, discussing relevant properties which are used throughout this work. Finally, section 3.4 shortly introduces the kernels and application used for the performed experiments.

3.1 Platform

The platform used for implementing and verifying all design is the ML605 Evaluation kit from Xilinx [1]. The target device on the development board is the Virtex-6 LX240T FPGA containing the MicroBlaze soft-core processor to supply hardware accelerators with data and verify the results coming back from the hardware accelerator.

3.2 Logic Synthesis and Implementation

Performance results presented throughout this work will be either logic synthesis and implementation results from the Xilinx ISE design suite embedded edition version 12.4 or from Synopsys Synplify premier version 2011-09.1. All resource utilization results are after logic synthesis unless stated otherwise and when needed the critical path will be extracted after implementation for an accurate maximum achievable clock frequency after routing and placement. In most cases, resource utilization results will be from the Xilinx toolsuite to keep the cross comparison as close as possible. In the case dividers are included in a design produced by Synphony (introduced in section 3.3, Synplify Premier Pro is used for synthesis and implementation as the generated dividers are not synthesizable by Xilinx ISE. A design including dividers is encountered in chapter 5.

3.3 High Level Synthesis

Because of the numerous high level synthesis tools available nowadays [79], the first goal is to focus on the most promising tools. An HLS meeting with the Parallel Architecture Research Eindhoven group (PARsE) was organized to establish a view of which HLS tools at that time were the most promising.

[1] http://www.xilinx.com/products/boards-and-kits/EK-V6-ML605-G.htm

Covering several aspects of high level synthesis, such as automatic parallelism extraction, supported data types and verification possibilities, the list was narrowed down. The following sections introduce two of the remaining tools of interest, AutoESL[2] from Xilinx and Synphony C Compiler[3] from Synopsys.

3.3.1 AutoESL

Xilinx's AutoESL High-Level Synthesis supports C, C++ and SystemC specifications as input language (with mainly restrictions on dynamic memory and pointers) and extends this by providing arbitrary precision data types libraries in C and C++, next to the native C data types, and supporting SystemC arbitrary data types. AutoESL synthesis starts with control and datapath extraction based on loops and conditional branches and the loop body's. Scheduling and binding is the core process during synthesis and in this step it is determined in which cycle operations are scheduled and it binds the operation to a resource. The user is able to override or assist the default synthesis behavior of AutoESL by use of design constraints and directives. Depending on the performance and area goals of the designer, directives can be applied to pipeline or unroll loops and several memory optimizations like array partitioning and reshaping enable concurrency by increasing the memory bandwidth. With the use of design constraints the user can specify the target clock frequency, specific target latencies across functions and loops and limit the number of a certain resource used in the design. AutoESL supports various kinds of interface types, such as FIFO's, FSL link, bus, memory, simple valid signals or user specified protocol blocks to enforce specific IO behavior.

Without going into too much detail, figure 3.1 presents AutoESL's input and output. Both the design and the test-bench are described at a high level language, either C, C++ or SystemC. Constraints and directives are supplied either by a script or by use of the user interface. Generated RTL is in the form of VHDL, Verilog and SystemC including a script containing logic synthesis constraints for the down-flow logic synthesis tool based on the user supplied constraints and directives. RTL simulation is possible using the RTL output of AutoESL, in combination with the automatically generated RTL wrapper for the design under test enabling the use of high level test-benches written in the same language as the design specification. Simulation is possible using either the SystemC RTL simulation, requiring no third party RTL simulator or by use of the third party tool Synopsys VCS or Mentor Graphics ModelSim. Besides Xilinx's own logic synthesis tool ISE, third party tool Synopsys Synplify Premier/Pro is also supported. The target technology library of AutoESL is limited to Xilinx-only FPGA devices, but being a leading company in FPGA design this poses no problem.

AutoESL will automatically apply certain optimizations, which can be turned off by the user. Each function is treated as a hardware block. Without user interaction, AutoESL will sequentially schedule these blocks without pipelining or exploiting data parallelism automatically. To execute independent loops or functions simultaneously, the user has merge them or can specify to execute them in a streaming fashion if the blocks operate as consumer-producer. An internal function of the design can either be in-lined to be in the same hierarchy as the top-level design or not in-lined to be in a separate hierarchy. By specifying separate functions and calling them from the top-level function, the same function specification can be used to generate multiple hardware blocks executing the same functionality in parallel reducing code overhead. In-lining a function into the caller function results in multiple hardware blocks being generated, even if they are executed sequentially.

3.3.2 Synphony C Compiler

Synopsys's Synphony C Compiler supports a restricted set of C and C++, with restrictions mainly on dynamic memory and pointers. Besides the native C/C++ data types up to 64 bits, the user can supply pragmas or use SystemC data types to ensure particular bitwidths up to 64 bits.

[2]http://www.xilinx.com/products/design-tools/autoesl/index.htm
[3]http://www.synopsys.com/Systems/BlockDesign/HLS/Pages/SynphonyC-Compiler.aspx

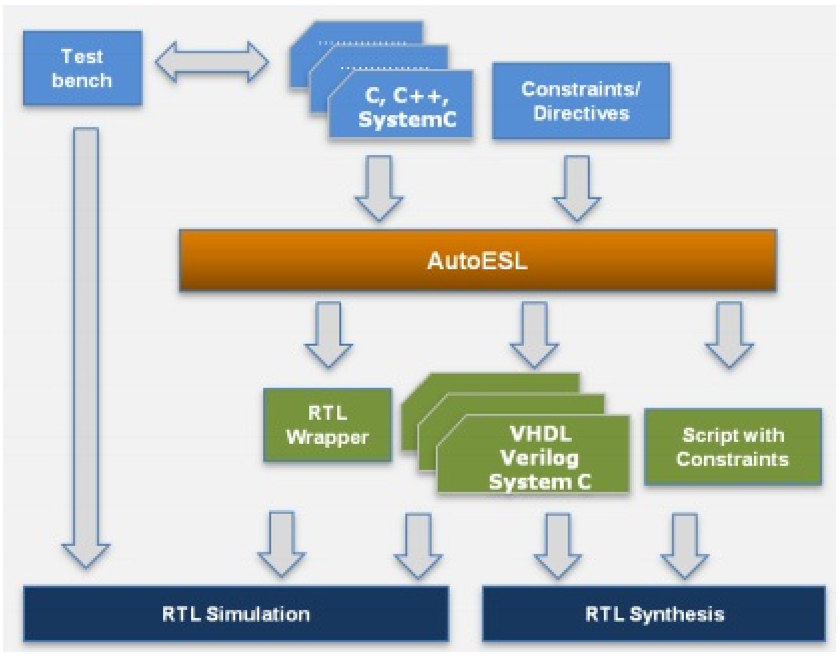


Figure 3.1: AutoESL use model [80]

Besides the SystemC data types, Synphony C Compiler does not accept other SystemC syntax. The compiler treats each top-level loop nest as a hardware block and will analyze dependencies between the blocks. Loop are automatically pipelined with the best possible initiation interval, the user can override the initiation interval to a higher value if necessary or has to apply source transformations or pragmas to improve the initiation interval. Synphony will automatically build control logic to run hardware blocks sequentially or in parallel, something which has to be done manually in AutoESL. The architecture, together with architectural parameters such as the architectural pipelinability enable Synphony to build task overlap capable hardware allowing the top-level hardware block to be pipelined.

Pragmas are supplied to specify optimizations such as loop unrolling, resource selection and block RAM reshaping to enable a higher bandwidth. A fundamental difference with AutoESL, regarding block RAM, is that partitioning an array over multiple RAMs or reshaping an array to have a wider bitwidth (pack array element in single block RAM entries) requires source-to-source transformations in Synphony whereas AutoESL automatically performs these actions using user supplied directives. Interfaces supplied by Synphony include a system interface for clock and resets, a raw signal host and data interface to connect with other hardware components, a three-signal handshake stream interface for external input/output and a FIFO interface for internal hardware blocks and a local memory interface to interface with block RAM/ROM not internally synthesized. Standard interfaces such as an FSL link are not supported and RTL wrappers need to be manually included to support such interfaces.

The design flow contains the usual high level synthesis steps such as preprocessing to transform the C/C++ to an intermediate representation, scheduling, allocating resources and actual RTL generation. After each step in the design flow it is possible to perform simulation at that level in order to verify correctness or trace the origin of errors. A golden C simulation is used to verify correctness after each synthesis step, with verification ranging from bit accurate SystemC simulation to RTL simulation using third party RTL simulators. The C/C++ test-bench can be re-used across all levels of simulation. Both Xilinx and Synplify synthesis tools are supported as down-flow EDA tools, RTL verification can be done using Synopsys VCS, Mentor ModelSim or Cadence NC-Verilog. The target technology can be either Xilinx or Altera FPGA devices.

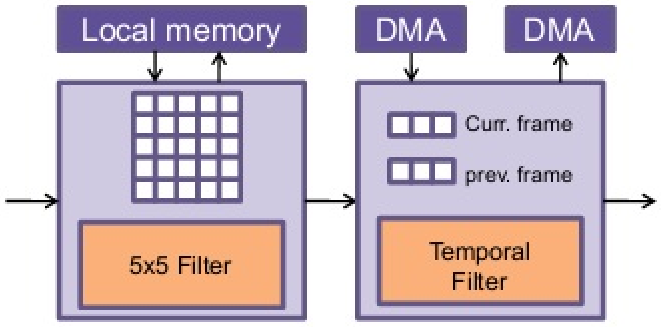


Figure 3.2: Example of library driven image processing

Synopsys identifies the importance of top-down template-based IP design. By applying architecture based coding styles for memory accesses, streaming data-flows or hardware sharing and mapping common types of processing from the reference algorithm to these architecture based coded template designs. Figure 3.2 shows the main idea with the purple blocks being template designs written in C/C++ for good quality of results using HLS and the orange blocks being kernels from the algorithm specification. This way, the modeling can happen at an algorithmic level reducing the programming effort to ensure good quality of results.

Architecture Template

Synphony maps C/C++ source code to flexible pre-defined hardware architectures for understandable and predictable synthesis. Although the resulting architecture generated by AutoESL is also understandable and predictable, Synphony includes pre-defined logic for control of the design. By default, Synphony will inline all functions at the point of the function call and handles all outer loops as hardware blocks, called processing arrays. A processing array (PA), shown in figure 3.3, is a low level template and each processing array in a design corresponds to a top-level loop nest. This case be seen as the instantiation of a hardware accelerator (processor) for each particular loop nest. The template corresponds to a simple processor, containing functional units, memory and memory access units all wired together with an interconnect. The interconnect consists of multiplexers controlled with a phase signal and control predicates, with the phase signal being the pipeline stage controlling which data goes to which functional unit. During synthesis, a modulo scheduler does resource and cycle binding for each operation in the PA. The registers in the PA come in two flavors. Rotating registers store multiple instances of the same variable for different iterations, a single variable rotating register for pipelined loops with an initiation interval of one or a so called *shiftq* rotating register chain for storage of one or more variables when the initiation interval is larger than one in order to share registers among variables. Second, static registers are used for static variables and include logic to control multiple writes to the register.

The stream access and memory access units are used for data accesses from memory or stream in the PA. The memory access unit is a separate logic block containing a replay buffer or FIFO to allow stalling of the PA. By allowing each PA to run or stall independently of other PA's, the throughput of the top-level design can be maximized, for example by allowing an other PA to access the block RAM until the stalled PA can run again.

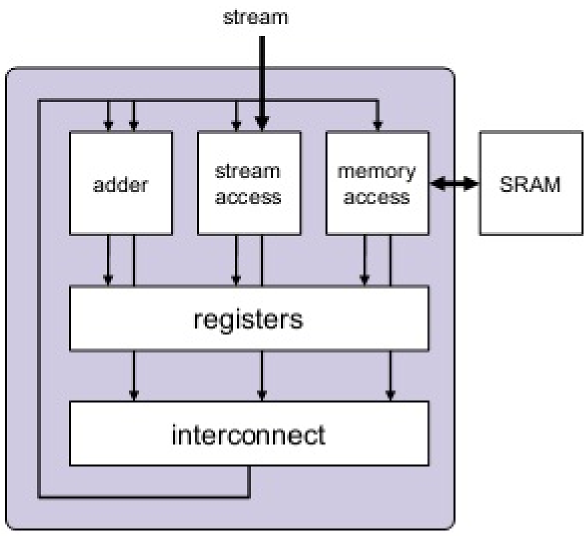


Figure 3.3: Low level template for a processing array

The high level template is a pipeline of processing arrays (PPA), containing the PAs of the design and control logic. Because each PA is in its own stall domain, allowing it to start or stop independently, the design allows FIFO flow control between PAs and a highly parallel design for mutual-exclusive hardware blocks. Figure 3.4 shows a top-level view of a PPA. The high level architecture template includes communication logic such as streams, shared memories or scalars used to let PAs communicate with each other. A global timing controller implements the control to start or stop PAs, the control is not static as control signals depends on stall signals coming from the PAs. A host interface is used to communicate with the environment, getting data from the environment or sending output data to other hardware accelerators or processors.

The PPA template architecture is used to try to achieve the highest possible throughput for a certain input description, allowing complex designs to be efficiently synthesized. In the vision domain, algorithms often are simple hardware pipelines of computational kernels executing after each other or with certain kernels executing in a streaming fashion. Synphony allows one or more functions to be synthesized as a tightly coupled accelerator block (TCAB) to avoid control logic for each individual stall domain. A TCAB can be used as standalone module or can be used as building block for high level designs, for example by importing a TCAB into a higher level PPA allowing hierarchical modular designs. In essence, a TCAB designs is a PPA with a single PA inside generated by wrapping an outer control loop over all loop bodies in the input specification. TCABs can be used for enhanced data path sharing. For a 2D-DCT kernel implemented as two 1D-DCT's in the PPA architecture template, each 1D-DCT is a separate hardware block. When build as TCAB the data-path of a 1D-DCT kernel can be shared in the 2D-DCT kernel at reduced area cost.

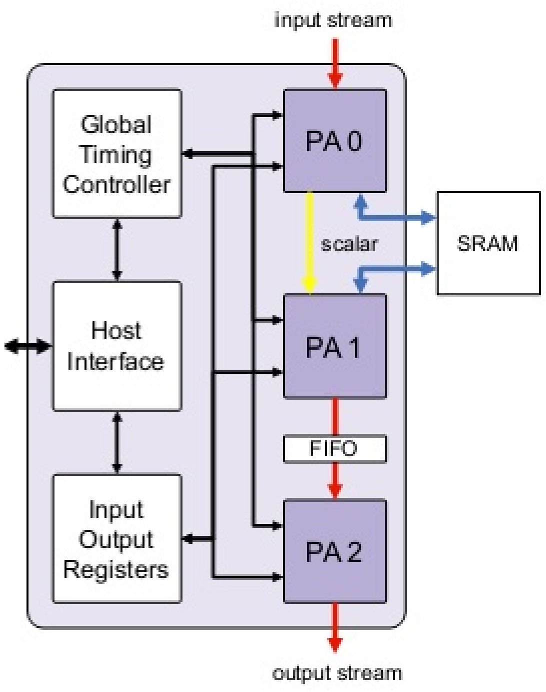


Figure 3.4: High level template for a pipeline of processing arrays

3.4 Kernels and application

An application called Fast Focus on Structures (FFoS) is used to perform the experiments regarding HLS performance and the introduction of algorithmic skeletons. From the application, two small image processing kernels are taken to perform the in-depth HLS analysis. These kernels are *histogram* and *erosion*, which will be discussed in more detail in chapter 4. In chapter 5, the complete FFoS application is mapped to FPGA using two HLS tools. The application is intended for finding centers in OLEDs on OLED wafers, and will be discussed in more detail in chapter 5. The two kernels *histogram* and *erosion* will be used again in chapter 6 to perform a design space exploration using HLS.

Chapter 4

HLS: A Kernel Comparison

In this chapter, two small image processing kernels are mapped to FPGA using AutoESL and the Synphony C Compiler. Goal is to mimic manual implementations of the kernels, which are taken from a manually written application in RTL which is later completely mapped to FPGA using HLS. The manually written RTL kernels are written by an experienced designer with vast knowledge of the application and RTL design. The kernel descriptions for HLS are written in C++, each kernel having a specific performance issue which can easily be solved when the kernels are manually implemented using RTL languages. This chapter evaluates the possibilities to solve these performance issues at C++ level using HLS and cross-compares different HLS tool implementations and the HLS implementations with the reference manual implementations. The following issues will be addressed regarding HLS performance compared to manual implementations:

- Latency
- Resource utilization
- Clock frequency
- Readability of the RTL
- High level synthesis early resource estimations
- Design time

Section 4.1 presents the analysis done for a histogram implementation, section 4.2 for an erosion implementation and finally section 4.3 summarizes the results to form a conclusion about RTL generation for small image processing kernels using HLS.

4.1 Histogram

Histogram is a common operation performed in the image processing domain. The histogram of an image is a representation of the distribution of the pixel values contained in the image. A gray-scale image for example contains 8-bit pixel values each of which can have 256 different possible intensities, the histogram contains 256 bins with each bin containing the number of pixels in the input image having the particular bin value. An example algorithm using histogram is Otsu's method [81], which uses a gray-level image histogram to calculate an optimum threshold separating foreground and background pixels to transform a gray-level image to a binary image.

A naive histogram implementation in C++ is shown in listing 4.1, which is histogram code as it would usually be programmed on a CPU. The performance issue for this kernel when targeting an FPGA is the random access on the *bins* array, which in FPGA design is usually implemented in a block RAM. Since the address of the memory access is dependent on the value of the input pixel, a compiler can not make the assumption that each access to the histogram bins is mutually exclusive, resulting in a high latency as the block RAM storing the histogram bins needs to be read and written before the next input pixel can be processed. In other words, there is a read-after-write (RaW) hazard requiring at least 3 clock cycles to execute the loop body.

```
for(x = 0; x < WIDTH*HEIGHT; x++){
        bins[ image[x]] ++;
    }
}
```

Listing 4.1: Reference C code for a histogram kernel

Sections 4.1.1, 4.1.2 and 4.1.3 present the manual, AutoESL and Synphony implementation respectively. The approach to solve the RaW hazard at C++ level using HLS and the efficiency of the generated data-paths compared to the manual implementation will be discussed.

4.1.1 Manual Implementation

The manual RTL implementation solves the RaW hazard by the introduction of an accumulation register and a data comparison to verify if the RaW hazard is true or false. Since the RaW hazard is only true when two succeeding read addresses on the histogram block RAM are the same, the data comparison result can trigger a multiplexer to increment in an accumulation register instead of reading data from the block RAM. Figure 4.1 shows the architecture of the manual implementation. The design is a 2-stage pipeline with a streaming input for the image pixels, a new pixel being available each clock cycle.

The histogram bins are 16 bit wide to support images up to 65536 pixels, with 8 bit input pixels for a total of 256 histogram bins. Although the *mem data* register is absorbed by the block RAM it is shown for clarity. The number of cycles required for the complete histogram operation, including initialization, is (N+1)+256, with N being the number of input pixels, 256 the number of cycles required for initialization and an extra cycle for filling the 2-stage pipeline. The path colored red in figure 4.1 is the critical path, resulting in a minimum clock period of 5.321 ns. Table 4.1 presents the logic distribution for the data-path of the histogram kernel and table 4.2 presents a top-level summary of the performance results.

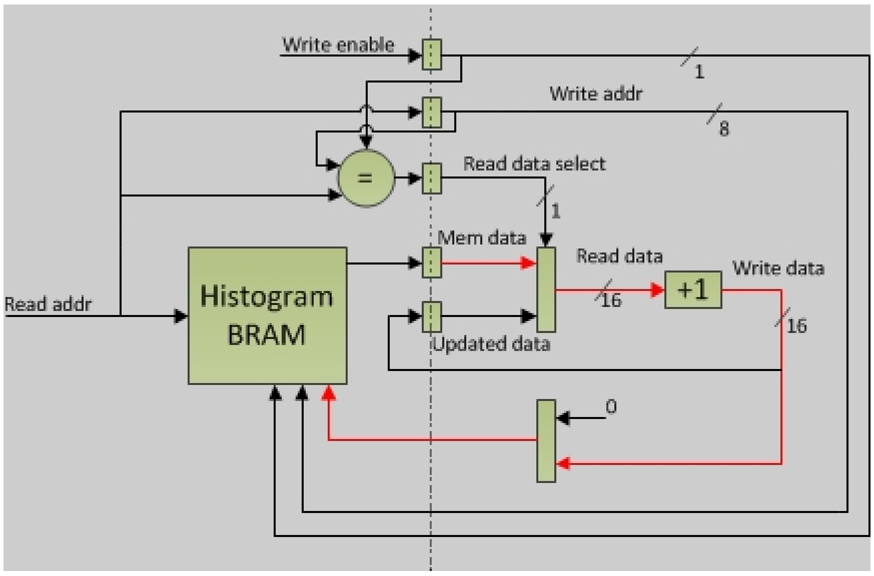


Figure 4.1: Main histogram datapath in manual implementation

Table 4.1: Logic distribution of the manual histogram implementation

Flip flops	**LUTs**
16-bit updated data	16-bit adder
8-bit write address	8-bit comparator
3-bit control	2x 16-bit mux
8-bit upcounter	2x 8-bit mux
	9x 1-bit control mux
	8-bit adder
Total: 35	**Total:** 69

Table 4.2: Summary performance results histogram implementation

Implementation	**Flip flops**	**LUTs**	**BRAM**	**Latency**	**Clock period**	**Design time**
Manual	35	69	1	N+1+256	5.321	2 days

4.1.2 AutoESL Implementation

This section discusses the results achieved with AutoESL, separated in the input specification, generated datapath and performance issues.

Input Specification

Just as with RTL, to improve performance the accumulation register and data comparison need to be explicitly coded but at a higher abstraction level (C++). By applying such source-to-source transformations and making use of optimization pragmas a similar architecture as the manual implementation (figure 4.1) is achieved. The source code which results in AutoESL generating the architecture of choice is shown in listing 4.2, leaving out some initialization and declaration statements for clarity. By conditionally executing the block RAM access, the designer can be sure a RaW hazard will not occur as each time the block RAM is accessed the read and write addresses differ. Although the code ensures mutual exclusive read and write addresses during block RAM accesses, the compiler does not automatically recognize this. The *dependence pragma* informs the AutoESL compiler that the RaW hazard it encounters on variable *bins* within each loop iteration is a false dependency, allowing the scheduler to schedule the read and write operations in any order and thus also in parallel. The *pipeline pragma* tells AutoESL to pipeline the loop to allow loop iterations to overlap. Variables are declared using the arbitrary precision library, with bit-widths equal to the manual design.

```
for(j = 0; j<WIDTH*HEIGHT+1; j++){
#pragma AP PIPELINE
#pragma AP dependence variable=bins intra RAW false
   index = image[j];
   if(old_index == index){
      accu = accu + 1;
   }else{
      bins[old_index] = accu;
      accu = bins[index] + 1;
   }
   old_index = index;
}
```

Listing 4.2: Optimized source code for the histogram kernel

Datapath

Figure 4.2 shows the generated datapath based on listing 4.2, with simplified control logic for clarity. The pipeline generated is a 3-stage pipeline. The pipeline is one stage deeper compared to the manual pipeline, a result of the use of for-loops in the high level programming language. Next to a pixel load operation scheduled in the first pipeline stage, loop control is executed in the first pipeline stage involving loop variable incrementing and exit condition checking. Block RAM initialization is separated from the histogram module because it is impossible to describe fine-grained multiplex behavior at C/C++ level without breaking the AutoESL imposed rules with regard to the *dependence pragma*. Combining initialization and the histogram module would involve changes to the loop bound and coding non-mutual exclusive block RAM access causing the dependency pragma to be invalid, reducing performance. Note that AutoESL generates an additional multiplexer compared to the manual design.

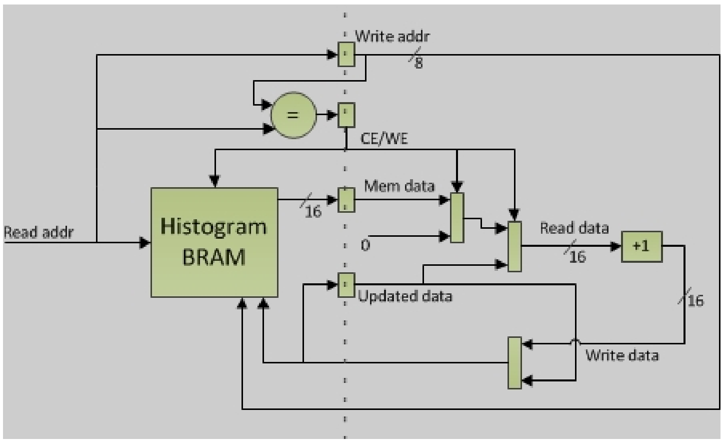


Figure 4.2: Main histogram datapath in AutoESL implementation

Table 4.3 presents the logic distribution for the datapath of the histogram kernel and table 4.4 presents a top-level summary of the performance results. From the results it can be seen that it is possible to re-create the data-path from the manual design with slightly more resource utilization. A few extra registers and LUTs are used for loop control, unavoidable at a high abstraction level like the C/C++ programming language.

Table 4.3: Logic distribution of the AutoESL histogram implementation

Flip flops	**LUTs**
16-bit updated data	16-bit adder
8-bit write address	8-bit comparator
13-bit loop counter	3x 16-bit mux
6-bit control	13-bit adder
2-bit FSM	
9-bit initialization counter	
Total: 54	**Total:** 92

Table 4.4: Summary performance results histogram implementation

Implementation	Flip flops	LUTs	BRAM	Latency	Clock period	Design time
Manual	35	69	1	N+1+256	5.321	2 days
AutoESL	54	92	1	N+2+256	4.829	0.5 days

Performance

In summary, using source transformations, a RaW hazard can be resolved at C++ level. The cost for using C++ and HLS instead of VHDL/Verilog is a small area overhead for increased multiplexing and static control logic for generated pipelines such as pipeline stage enable signals and exit conditions. Loop counters are common in manual RTL descriptions as well, however this implementation shows we can connect the manual histogram module to an FSL link and use an FSL control signal to enabled/disable the histogram module. Since AutoESL allows the designer to describe protocol regions, such a design appears to be feasible as well, however with the algorithmic skeletons and algorithm classification in mind the choice is made to continue with high level loop descriptions. Execution of the histogram kernel requires one extra cycle, resulting from the extra loop iteration for the delayed write to the histogram block RAM. The minimum clock period is 4.829 ns, the critical path being the path from a block RAM read to a store. The latency for this design is (N+2)+256 cycles.

The generated RTL by AutoESL is readable and modifiable. If the designer does not in-line functions in the main calling function (top-level design), AutoESL generates separate RTL files for each function. Generated RTL contains comments indicating the functionality of the code block, indicating for example the datapath assign process and the pipeline stage control assign processes. Wire and register names are related to the input specification whenever possible and AutoESL only introduces unrelated names when scheduling requires components to be introduced. These registers or wires can always be tracked down using the AutoESL schedule viewer or by tracing them through the generated RTL manually.

With regard to area estimation performed by AutoESL, register estimation is 100% accurate besides logic synthesis properties such as register duplication. The estimation of utilized LUTs is less accurate because logic synthesis is a complex operation which can introduce or remove logic based on its set of optimization algorithms. The estimation will however give a fairly accurate estimation of the number of arithmetic operations required and the level of multiplexing required to route signals through the datapath. In the end, AutoESL is a high level synthesis tool and not a logic optimization tool such as Xilinx ISE. AutoESL will leave room for optimizations as not to constrain logic synthesis too much, possibly enabling logic synthesis to find additional optimizations, and therefore the estimations are more of use during design space exploration. In the same way, the estimated clock frequency is based on the generated schedule and a library of component delays not taking into account logic optimizations in downflow tools. If clock frequency and resource utilization are hard constraints during the design, it is obvious that the designer should not base conclusions on HLS performance estimates.

4.1.3 Synphony Implementation

This section discusses the results achieved with Synphony, separated in the input specification, generated datapath and performance issues.

Input Specification

Besides from some tool-specific properties such as bit-width specification, memory properties and interface specification, the input specification to the tool is the same as for AutoESL as is shown in listing 4.2. A pragma can be used again to inform the compiler that there is no RaW hazard in the histogram description as it is not automatically recognized after the source transformation. In the case of Synphony, this pragma completely disables memory analysis allowing parallel read and writes. The tool performs memory tracing which logs all memory accesses during simulation after scheduling. To get the required latency/throughput, it is up to the user to verify the absence of access collisions on the block RAM throughout the entire algorithm. In AutoESL, the compiler is informed that the RaW hazard is a false dependency within a single loop, requiring the designer to only verify that this is indeed the case for the loop in question. Furthermore, Synphony does not support standard interfaces such as an FSL and streams have to be coded manually to generate them on the interface. To connect the design via an FSL interface with a Xilinx MicroBlaze, a separate wrapper needs to be written in RTL requiring some extra logic. This FSL wrapper is not included in this implementation.

Datapath

The architecture template chosen is a TCAB, as explained in section 3.3, as the design is just a simple hardware pipeline. The histogram kernel only contains an initialization and process loop, removing the need for complicated control logic for each loop as the dataflow is known beforehand. The TCAB design wraps a loop around all generated datapaths from inner loop bodies and introduces control to select in which stage the pipeline is, selecting the correct datapath to execute. In this case, the first stage initializes the histogram BRAM, the second stage performs histogramming and the last stage writes the result to the output using the output stream. The TCAB design, to reduce control logic, forces us to include hardware to write the histogram data to the output in order for Synphony to synthesize the design. All these datapaths are now within the same loop with control logic ensuring the correct *inner* data-path is enabled and controlling the counter to keep track of the state. The design is, just as the AutoESL design, a 3 stage pipeline of which a high level overview is shown in figure 4.3. A detailed datapath is not given because the generated RTL is unreadable, containing component/wire/register names with no correlation to the input specification. Using an RTL schematic viewer in combination with Synphony's operation schedule viewer a top-level overview can be presented. Generated designs have a deep modularity with unclear signal names, making it difficult to trace through the datapath. Each instantiated component is in a deeper part of the design hierarchy and possibly contains more instantiated components.

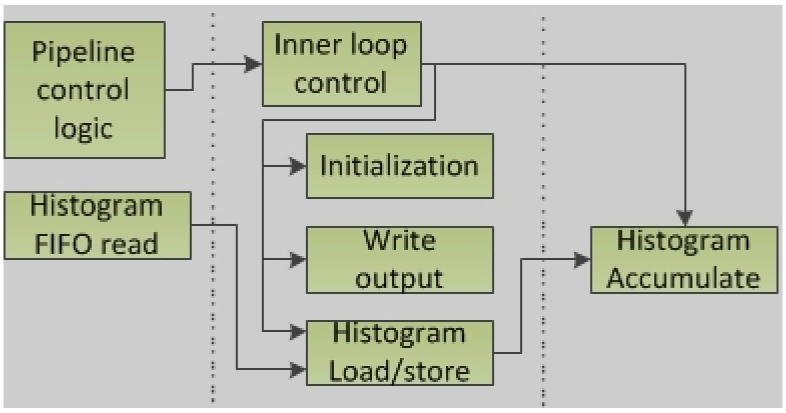


Figure 4.3: Histogram top-level design overview

In figure 4.3, the first pipeline stage implements top-level control for selection of inner datapath to be executed and depending on the state a FIFO read is performed. The second pipeline stage contains the datapath for block RAM initialization, the first pipeline stage of histogramming (loading and storing from and to the histogram *bins*), or a FIFO write to the controller of next IP. The final stage performs addition when the histogram kernel is executed, accumulating either in the accumulation register or on loaded *bin* data from block RAM. It should be noted that although just as in AutoESL the final histogram pipeline stage performs the addition, the Synphony implementation requires two adders to either increment in the register or on the block RAM data. This confirms the hardware architecture template approach, which limits the amount of resource sharing.

Table 4.5 presents the macro statistics for the datapath of the histogram kernel and table 4.6 presents a top-level summary of the performance results. Table 4.5 shows macro statistics extracted after elaboration and table 4.6 presents the post-synthesis results. Macro statistics are used to get a feeling of the area utilization, information which is lost after logic synthesis because the number of components is simply too large and results are untraceable caused by no correlation in component names and input description.

Table 4.5: Macro statistics of the Synphony histogram implementation

Flip flops	**LUTs**
36x 1-bit	1-bit adder
1x 13-bit	13-bit adder
2x 16-bit	2x 16-bit adder
1x 17-bit	2-bit adder
1x 2-bit	6x 1-bit comparator
2x 8-bit	2x 13-bit comparator
	5x 2-bit comparator
	2x 5-bit comparator
	1x 8-bit comparator
	8x 1-bit mux
	2x 14-bit mux
	2x 16-bit mux
Total: 116	**Total:** 320

Table 4.6: Summary performance results histogram implementation

Implementation	**Flip flops**	**LUTs**	**BRAM**	**Latency**	**Clock period**	**Design time**
Manual	35	69	1	N+1+256	5.321	2 days
AutoESL	54	92	1	N+2+256	4.829	0.5 days
Synphony	109	231	1	N+2+256	6.089	0.5 days

Two 16-bit adders are used for the histogram *bin* and accumulation register incrementing. The 16-bit multiplexers, extra registers and small adders are used in FIFO and block RAM load/store units. These load/store units are always synthesized, even if the design does not need to stall the logic it is still included and sharing capabilities for functional units is limited. Whereas AutoESL implements most loop control with simple *and* gates, Synphony includes more comparators, registers and adders for control. Overall, table 4.5 shows a large area overhead for the TCAB architecture template and extra overhead in the histogram datapath (extra adders and registers) because of less sharing capabilities. A case-study later on will show if this area overhead is scalable with the application or not, and section 4.2.3 discusses a kernel implemented in the PPA architecture template.

Performance

The design latency is N+2+256 cycles, with N being the number of input pixels, one extra cycle for pipeline ramp up, one extra cycle for the delayed write and 256 cycles for block RAM initialization. The design latency is identical to the AutoESL implementation. The minimum clock period is 6.089 ns, higher compared to AutoESL as could be expected with the area overhead. The critical path in this case is not caused by a block RAM access, but is in fact a control path in the design.

The generated RTL is not readable and therefore also not easily modifiable. In the TCAB design, the functionality is implemented in one file with wrappers to control the processor. Since there is some information about the scheduled time of functional unit instantiations, mappings to source code and pseudo code (generated during scheduling), it is possible to track certain statements down to the RTL but due to the deep hierarchy changes and full understanding are impossible in a reasonable amount of time. Area estimation by Synphony is more extensive than in AutoESL. A separate register report is generated explaining the use and location of each register bit. A cost report gives a complete resource utilization estimate. Register estimation is only off by 17% as again HLS is not logic synthesis and does not take logic optimizations into account. Synphony has made the choice to estimate small components such as *and* gates and *xor* gates to use zero LUTs as it is dependent on the logic synthesis process. AutoESL estimates these small components to use one LUT each. For this reason, LUT estimations in Synphony are usually under-estimated.

4.2 Erosion

Erosion is a typical example of a morphological operation in which the value of each pixel of the output image is determined based on the corresponding input pixel and a set of neighbors. The operation involves a structural element, for example an NxN window for 2D erosion, which 'slides' over the input image. The value of the output pixel in an erosion kernel is the minimum value of all pixels in the structuring element, meaning that the image does not necessarily has to be in binary format. The algorithm thus removes pixels on the boundaries of objects of interest, removing noise.

A naive erosion implementation in C/C++ is shown in listing 4.3. When targeting this code to FPGA, the main performance issue is the structural element which has overlapping image accesses. If the image is stored in block RAM, the loop iterating across the width of image requires 9 input pixels each iteration of which 6 pixels could potentially be re-used. When no optimizations are applied, and the input image being in a single block RAM, at most 2 input pixels can be loaded each cycle resulting in a poor design.

Sections 4.2.1, 4.2.2 and 4.2.3 present the manual, AutoESL and Synphony implementation respectively. The approach to solve the limited bandwidth issue at C++ level using HLS and the efficiency of the generated data-paths compared to the manual implementation will be discussed.

```
int condition;
for(h=0;h<height;h++){
    for(w=0;w<width;w++){
        if( h<1 || h>= height-1 || w<1 || w>=width-1){
            condition = 0;
        }
        else{
            condition = 1;
            for(a=-1;a<=1;a++){
                condition = condition & image[h-1][w+a]
                                      & image[h   ][w+a]
                                      & image[h+1][w+a];
            }
        }
        output[h][w] = condition;
    }
}
```

Listing 4.3: Reference C code for an erosion kernel

4.2.1 Manual Implementation

The manual implementation improves the performance by increasing bandwidth to the input image. Performance in increased by fetching a complete line from the input image each cycle, exploiting fine-grained parallelism available in the FPGA. Figure 4.4 shows the architecture of the manual implementation. The design is implemented for a 120x45 input image but can be easily parameterized. Input to the design is a 120-bit vector, being a complete line from the input image, from external block RAM. The output is also a 120-bit vector, going to an external block RAM. Counters are used to keep track of the block RAM read and write addresses, intermediate shift registers are used to buffer 3 input lines. Buffering 3 lines implies a window size of 3x3 and the shift registers are extended with zero-bits to handle the borders of the image when calculating horizontal erosion. A bit-wise *and* operation is performed in the vertical direction on the shift registers, followed by a horizontal bit-wise *and* operation for the horizontal erosion operation.

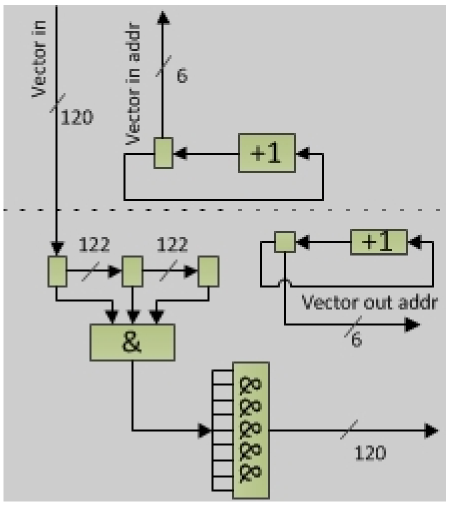


Figure 4.4: Main erosion data-path in manual implementation

The number of cycles required for the complete erosion operation, with an input image of 120x45 and a window size of 3x3, is 48. Initialization takes one cycle, followed by two prefetch cycles to load data in the shift registers. In total, 44 cycles are required for the calculation of output and a final cycle is used to write the last delayed output line. The critical path in the design is a control path from FSM to shift register control, indicating a small and balanced datapath is generated. The minimum clock period is 5.139 ns. Table 4.7 presents the logic distribution for the datapath of the erosion kernel and table 4.8 presents a top-level summary of the performance results.

Table 4.7: Logic distribution of the manual erosion implementation

Flip flops	**LUTs**
1-bit ready	2x 6-bit adder
6-bit read address	1-bit mux
6-bit write address	
1-bit write enable	
3x 120-bit line buffers	
3-bit one-hot FSM	
Total: 377	**Total:** 258

Table 4.8: Summary performance results erosion implementation

Implementation	**Flip flops**	**LUTs**	**BRAM**	**Latency**	**Clock period**	**Design time**
Manual	377	258	0	48	5.139	2 days

4.2.2 AutoESL Implementation

This section discusses the results achieved with AutoESL, separated in the input specification, generated datapath and performance issues.

Input Specification

Listing 4.4 presents optimized source code for an erosion kernel using AutoESL. The AutoESL arbitrary precision library is used to vectorize the input and output and access specific bits within vectors. Some initialization is not shown for clarity, such as initializing borders of the row buffers to zero to handle the borders of the image. Because of the large vectors, the numbers of multiplexers is kept to a minimum by padding the input data with an extra line to prevent a conditional input read statement which results in generation of wide multiplexers.

As mentioned in section 3.3, AutoESL supports pragmas for automated block RAM reshaping and partitioning. Applying these pragmas to the original input specification for erosion gives unexpected results, showing the limitation of automatic internal transformations by AutoESL and showing that manual re-coding is required for better quality of results. The unexpected results include the generation of a large number of DSP units to calculate bit-indexes in the output vectors as the compiler does not recognize the inner unrolled loop writes to complete reshaped output vectors, eliminating the need for DSPs to calculate bit positions. The large amount of area overhead and the cycles required for DSP calculations result in both a large area overhead as a large latency overhead.

The code shown in listing 4.4 implements vertical and horizontal (inner loop) erosion, with the inner loop unrolled to achieve line-by-line processing just as in the manual implementation.

```
void erosion(ap_uint<120> * data, ap_uint<120> * output){
unsigned int i,j;
ap_uint<120> row1;
ap_uint<120> row2;
ap_uint<120> row3;
ap_uint<120> tmp;

for(i=0; i<HEIGHT+1; i++){
#pragma AP PIPELINE
    row1 = row2;
    row2 = row3;
    row3 = data[i];
    tmp = row1 & row2 & row3;
    if(i>0){
        for(j=1;j<WIDTH-1;j++){
        #pragma AP UNROLL
            output[i-1].set_bit(j, (tmp[j-1] & tmp[j] & tmp[j+1]));
        }
    }
}
```

Listing 4.4: Optimized source code for the erosion kernel

Datapath

Figure 4.5 shows the generated datapath based on listing 4.4, with simplified control logic for clarity. The pipeline generated is a 3-stage pipeline. The first stage loads the input vector and output vector. The output vector is loaded in this stage because AutoESL implements a read-modify-write operation for block RAMs which data elements are written using arbitrary precision library bit accurate operations. The implementation requires that the input block RAM needs one extra 120-bit element to represent a *zero line* to prevent the use of conditional statements in the input specification. The read-modify-write arithmetic operation requires the output block RAM to be dual ported to facilitate a read and write operation in the same cycle. Next to calculation of read and write addresses in the first pipeline stage, outer loop control is also executed in this stage although it is not shown for clarity. The second stage implements row shifting through the row buffers and the final stage performs vertical and horizontal erosion by applying the *and* operator and writes the result to the output.

Table 4.9 presents the logic distribution of the erosion kernel after logic synthesis and table 4.10 presents a top-level summary of the performance results. Only a few extra registers are required, mainly for the read-modify-write behavior of the 120-bit vectors and automatically generated pipeline enable and exit bits.

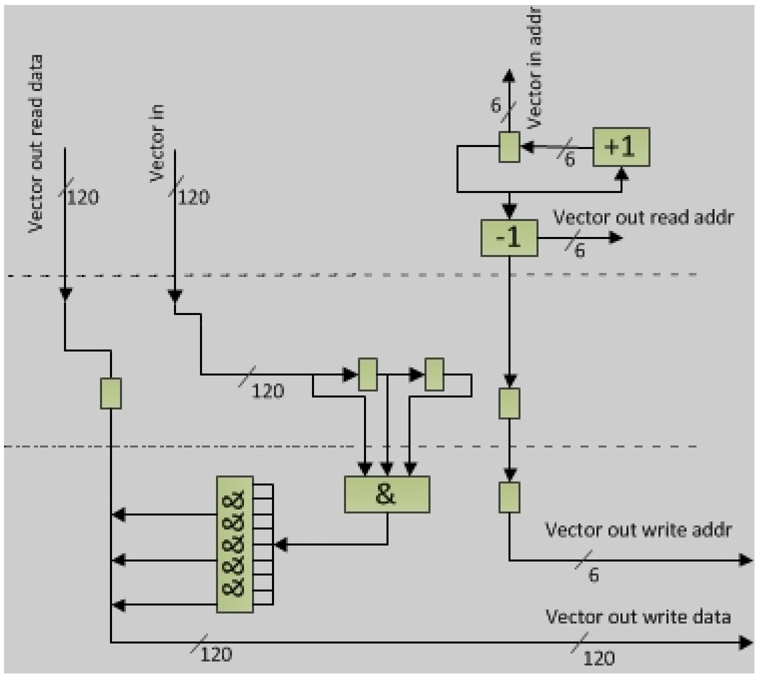


Figure 4.5: Main erosion data-path in AutoESL implementation

Table 4.9: Logic distribution of the AutoESL erosion implementation

Flip flops	**LUTs**
4x 1-bit pipeline enable	2x 6-bit adder
2x 1-bit pipeline exit conditions	1-bit mux
2x 6-bit read address	
6-bit write address	
3x 120-bit line buffers	
1-bit conditional statement control	
2-bit one-hot FSM	
Total: 387	**Total:** 277

Table 4.10: Summary performance results erosion implementation

Implementation	**Flip flops**	**LUTs**	**BRAM**	**Latency**	**Clock period**	**Design time**
Manual	377	258	0	48	5.139	2 days
AutoESL	387	277	0	48	5.482	0.5 days

Performance

The latency of the design is 48 cycles, calculated as 46 iterations with a pipeline ramp-up of 2 cycles. The minimum period for the design is 5.482 ns, with the critical path being just as with the manual design in the control path of the design. Code readability becomes more complex with large designs including unrolled loops, but the main functionality and control is still implemented in separate processes making it easy to make small modifications or understand the generated

RTL. Analysis of generated RTL made it possible to trace the origin of the generated multiplexers back to the C++ description and solve the issue by adding a single block RAM entry and loop iteration. The resource estimation with regard to flip flops is again 100% accurate and the LUT estimation suffers from lack of logic synthesis knowledge estimating a single 6-input LUT required for implementation of an *and* gate. After synthesis, the number of LUTs required decreases because of logic optimizations applied at this design level.

4.2.3 Synphony Implementation

This section discusses the results achieved with Synphony, separated in the input specification, generated datapath and performance issues.

Input Specification

Listing 4.5 and 4.6 present optimized source code for an erosion kernel using Synphony. Listing 4.5 shows the approach to vectorize data in Synphony beyond its 64 bit limit, although Synphony allows this approach the tool is running in a beta mode when using such large vectors. By declaring arrays with the *struct* type declared in listing 4.5 it is possible to use vectorized 120-bit data. Synphony calls this *packed data*, and the tool puts certain constraints on the use of packed data. Packed memory reads return the entire *struct* contents, and writes to single *struct* fields is not legal. This means for the erosion kernel that an extra 120-bit buffer is required to explicitly write to the entire *struct*, something which was not necessary in the AutoESL implementation. In listing 4.6, the *internal_fast* pragma is used to indicate that the 120 element array containing single-bit elements needs to implemented in flip flops instead of a block RAM in order to support the parallel operation. The use of an array representation of vectors requires an extra inner loop to do vertical erosion and the use of an extra output buffer, required by packed memories, requires a slightly different C++ implementation compared to AutoESL.

```
typedef struct{
   unsigned int line[120];
   #pragma bitsize line 1
} image_line;
```

Listing 4.5: Vectorize data in Synphony

Datapath

The chosen architecture template is a PPA, or pipelined processing array as explained in section 3.3. Since the focus is on the erosion kernel in isolation, it is required to have a modular design to separate the erosion kernel from the data-preparation in hardware. The binary input to the design is first parallelized and after the erosion kernel serialized to communicate with the test-bench and these kernels would all be merged in a TCAB design. By applying the PPA architecture template it is possible to separate the erosion kernel for further analysis while keeping it comparable to the manual and AutoESL implementations as all PPA control logic is at the top-level design and both a processing array as a TCAB design always includes the stall domain logic.

```
void erosion(image_line * data, image_line * output){
unsigned int i,j;
image_line row1;
#pragma internal_fast row1.line
image_line row2;
#pragma internal_fast row2.line
image_line row3;
#pragma internal_fast row3.line
image_line tmp;
#pragma internal_fast tmp.line
image_line tmp_out;
#pragma internal_fast tmp_out.line

for(i=0; i<HEIGHT+1; i++){
#pragma AP PIPELINE
    row1 = row2;
    row2 = row3;
    row3 = data[i];
    #pragma unroll
    for(j=0;j<WIDTH;j++){
        tmp.line[j]=row1.line[j] & row2.line[j] & row3.line[j];
    }
    if(i>0){
        #pragma unroll
        for(j=1;j<WIDTH-1;j++){
            tmp_out.line[j]=tmp.line[j-1]&tmp.line[j]&tmp.line[j+1];
        }
        output[i-1] = tmp_out;
    }
}
```

Listing 4.6: Optimized source code for the erosion kernel

Table 4.11 presents the macro statistics for the datapath of the erosion kernel and table 4.12 presents a top-level summary of the performance results after synthesis. Again, macro statistics are shown to highlight area consumption in Synphony as most information is lost after synthesis due to unrelated functional unit and register names.

Table 4.11: Macro statistics of the Synphony erosion implementation

Flip flops	**LUTs**
3x 120-bit line buffer	1-bit adder
2x 6-bit address buffer	6-bit adder
1x 7-bit	7-bit adder
16x 1-bit	7-bit subtracter
	6-bit comparator
	7-bit comparator
	8-bit comparator
	120-bit mux
	7x 1-bit mux
	2x 6-bit mux
Total: 392	**Total:** 449

Table 4.12: Summary performance results erosion implementation

Implementation	Flip flops	LUTs	BRAM	Latency	Clock period	Design time
Manual	377	258	0	48	5.139	2 days
AutoESL	387	277	0	48	5.482	0.5 days
Synphony	392	449	0	47	6.037	0.5 days

Although Synphony requires the explicit coding of an extra line buffer to write to a *packed memory*, the RTL generation process maps it to a wire which is similar to the AutoESL implementation. Compared to AutoESL, only two block RAM address registers are needed because Synphony does not implement the read-modify-write operation. Similar to the histogram implementation in Synphony, more functional units are generated as they are part of the architecture template. Figure 4.6 shows the main idea of the erosion datapath. The architecture is now a two-stage pipeline instead of a 3-stage in the case of AutoESL. Reason for this is the absence of a read-modify-write operation. Again, most storage is used for row buffers and the *and* gates take up most of the LUTs.

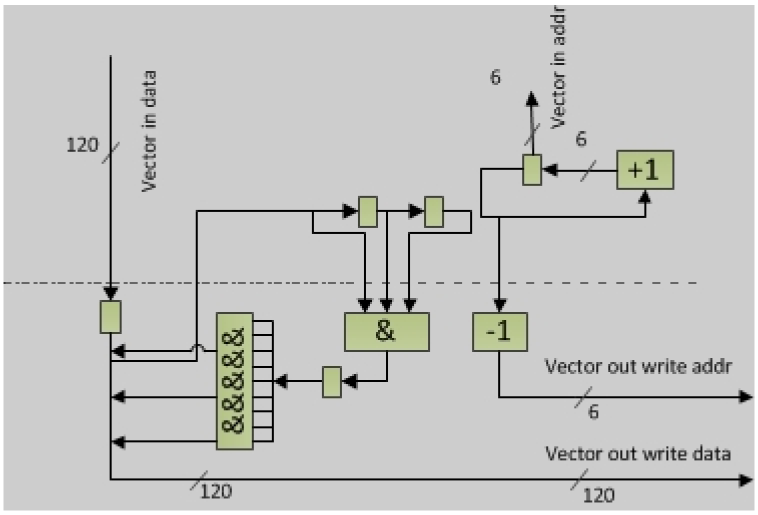


Figure 4.6: Main erosion data-path in Synphony implementation

Performance

The latency is 47 cycles, 46 cycles for the number of loop iterations and an extra cycle for pipeline ramp-up. The minimum clock period is 6.037 ns, with the critical path again being in the control part of the design. The critical path is slightly longer because of a longer combinatorial path, indicating again that Synphony's architecture template does not generate optimal datapaths for these kind of vision kernels.

As an indication of the effort required to manually debug, change or analyze the generated RTL, the erosion kernel consists of more than 12000 lines of code compared to 1100 lines in the AutoESL RTL and 300 lines in the manual implementation. The resource estimation is comparable to the histogram implementation. The estimated number of flip flop is overestimated by only 5 as the RTL generation process is unaware of logic synthesis optimizations and leaves room for further optimizations. Lookup-table estimation is vastly underestimated again, in this case by 153% caused by the large number of *and* gates which are estimated as zero-cost by Synphony.

4.3 Conclusion

Overall, AutoESL performs better with regard to resource utilization compared to Synphony. The stall domain logic, including the logic to control the stalling architecture template, cause an area overhead in Synphony. Latencies of all discussed designs are comparable, although AutoESL tends to include a separate pipeline stage for loop control increasing the end-to-end delay with one cycle. The most important observation about the design latencies is that for both designs a similar datapath with similar latency can be achieved, and an extra few cycles for loop control seems an acceptable cost for moving to C/C++ instead of using RTL. Next to comparable latencies, there is also no shocking difference in minimum achievable clock period, although Synphony tends to have a higher minimum clock period because of longer combinatorial paths caused by area overhead of the architecture template.

After this comparison, the first indication is that Synphony works with an architecture template which can be suitable for a wide range of applications but performs worse (in area utilization) compared to AutoESL when dealing with mentioned image processing kernels (simple pipelines).

The generated RTL from Synphony is hard to read, removing the possibility of modifications and further fine-tuning. The tool however does supply the user with a very complete schedule viewer and intermediate pseudo code related to the generated RTL. The kernel description in C can be highly tool dependent, as was seen in the erosion comparison. The tools both support important optimization pragmas to steer the design in the right direction. It should be noted that AutoESL provides pragmas for automatic memory reshaping and partitioning, which can aid the user by requiring minimum source transformations. However, in the case of erosion, these pragmas fail to result in the expected datapath and care has to be taken during the optimization process of the input specification.

In conclusion, HLS seems to have become a promising design tool. Especially with the increasing chip densities and shortening time-to-markets, some area overheads become more acceptable with as trade-off a much shorter design time and the introduction of IP re-use at C level. This last factor enables designers to make large architectural changes by applying just a few changes to the source code (think of re-coding 10 lines of C++ compared to 100/200 lines of RTL code), while still being close to the performance of manual RTL implementations. The kernels can be manually implemented using RTL in one or two days, whereas the HLS designs can be implemented in just a few hours or even less when the naive C description for the kernels is already available. Furthermore, architectural changes at the C behavioral level are implemented faster once the initial design is finished.

Chapter 5

HLS: Application Mapping

In this chapter a complete application is mapped to FPGA using HLS. In the previous chapter it is shown that it is possible to generate highly parallel solutions from sequential C++ and solve a RaW hazard. Area overheads may become excessive area overheads when multiple kernels are combined to form a complete application. It could also be that resources encountered in the single kernel HLS will be shared across kernels and therefore 'hidden' in the end result. As seen in section 2.3, other comparisons are based on preliminary commercial HLS tools and do not cross-compare between HLS tools. Next to serving as an evaluation purpose, the progress of optimizing the kernels has shown various code transformations for different datapath styles on which will be elaborated in a later chapter in an attempt to automate the code transformation process.

Section 5.1 presents the application of choice to be implemented on FPGA. Section 5.2 introduces the manually written design followed by sections 5.3 and 5.4 discussing the AutoESL and Synphony implementations respectively. Section 5.5 contains a discussion about the observations done during the process. The analysis will mainly focus on the end-result of the application implementation. Section 5.6 presents some alternative solutions implemented in AutoESL and Synphony, based on observations from the earlier discussion. Finally, section 5.7 presents a conclusion on HLS regarding AutoESL and Synphony based on the application mapping. This section will also conclude with a global overview of generated designs and their performance related to each other.

5.1 The Application

The application used is Fast Focus on Structures (FFoS) and was manually implemented by Yifan He, a PhD student at the electronic system group. The application is a typical vision application, fitting the domain specific approach. The algorithm is intended to find the centers of OLEDs, as during fabrication organic materials have to be injected into the OLEDs. For improved quality and yield it is important to find these centers fast and accurately. The vision pipeline for this application is shown in figure 7.2.

At the start of the pipeline, the OTSU algorithm is applied to the grey level input image to find the optimal threshold for binarization. In the following stage, binarization creates a binary image based on the threshold result of the OTSU algorithm. Erosion then removes the noise from the image, leaving the binary representation of the OLEDs intact. Row and column projection then counts the number of OLED pixels in each direction and finally the center is found by thresholding the row and column vectors. The center of each remaining line segment is the center of an OLED, with the y position in the row vector and the x position in the column vector.

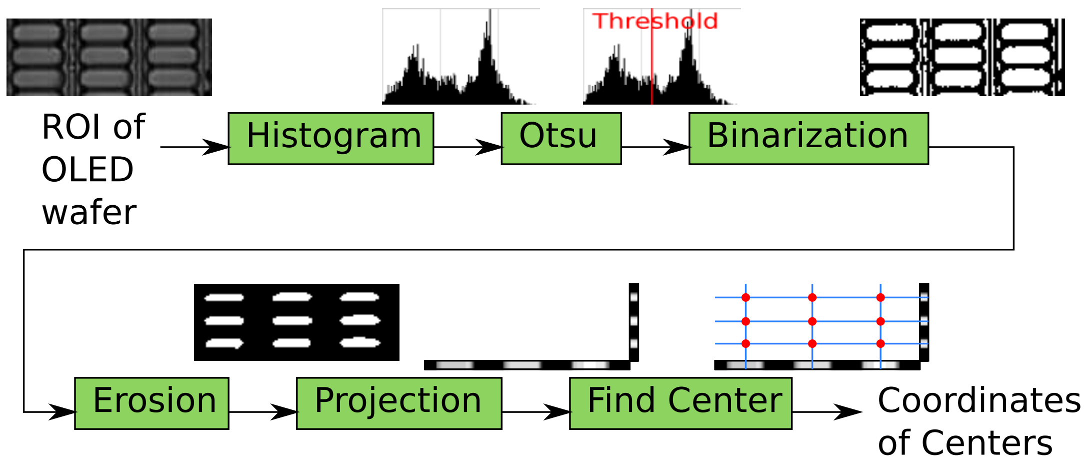


Figure 5.1: The vision pipeline for the fast focus on structures application

5.2 Manual Implementation

The handwritten RTL implementation is explained next as it is the reference for the HLS implementation. The sections are separated into a section explaining the input specification, a section discussing the resource utilization and a final section about the latency of the design.

5.2.1 Input Specification

The manual RTL implementation is based on a reference C++ description of the application which is also the starting point for the high level synthesis implementations. The histogram and erosion kernels discussed in chapter 4 are taken from this reference implementation. Next to the data parallel erosion kernel and the histogram implementation the following modules are part of the design:

- Interface: A FIFO interface compatible to the FSL_V20 FSL bus protocol and a PLB interface.
- FSM: A finite state machine controlling the kernels.
- Grey image copy: Copies the input pixels to a block RAM for later use. Step is executed together with histogram.
- Pixel sum: Calculates the sum of all pixels for use in the OTSU step. Step is executed together with histogram.
- OTSU: A computational intensive kernel calculating the optimal threshold.
- Binarization: Converts 8-bit input pixels to a binary value based on the optimal threshold. The output bits are placed in 120 bit wide vectors to support the parallel execution of erosion.
- Projection: Performs data-parallel horizontal and vertical summations over the binarized image.
- Estimated center: Searches for OLED centers in the projection vectors.

The reference design is written in Verilog and the parallel nature of the language makes it easy to separate all the kernels and introduce a top level FSM to control the design. It should already be noted that such high modularity is not achievable with HLS. Depending on the tool used, each loop will have its own FSM generated by the tool (AutoESL) or will not include an FSM at all (Synphony) and the design is controlled by control signals in the data-path. To enable parallel execution of the *histogram*, *image copy* and *pixel sum* in HLS, loop merging will need to applied disabling the possibility of analysis of separate kernels. Therefore, the logic distribution in the HLS implementation discussions differs from the manual distribution.

5.2.2 Resource Utilization

Figures 5.2 and 5.3 present the logic distribution across the application kernels. The distribution is based on an image resolution of 120x45 and a 3x3 window operation in the erosion kernel. From the seven block RAMs, four are used for image storage. An image copy kernel copies incoming pixels from the FSL to the image block RAMs, requiring two block RAMs for the 5400 8-bit pixels. Binarization prepares 120-bit vectors also requiring 2 block RAMs because of the large bandwidth required in the erosion kernel. The histogram is stored in a separate block RAM and prior to the OTSU kernel, a CH/CIA kernel computes the cumulative histogram and cumulative intensive area and stores it in 2 block RAMs. Note that this is not a fully optimal implementation as OTSU could use a cumulative value of both the CH and CIA, removing the need for the two block RAMs. After synthesis the CH and histogram block RAMs are merged into a single BRAM.

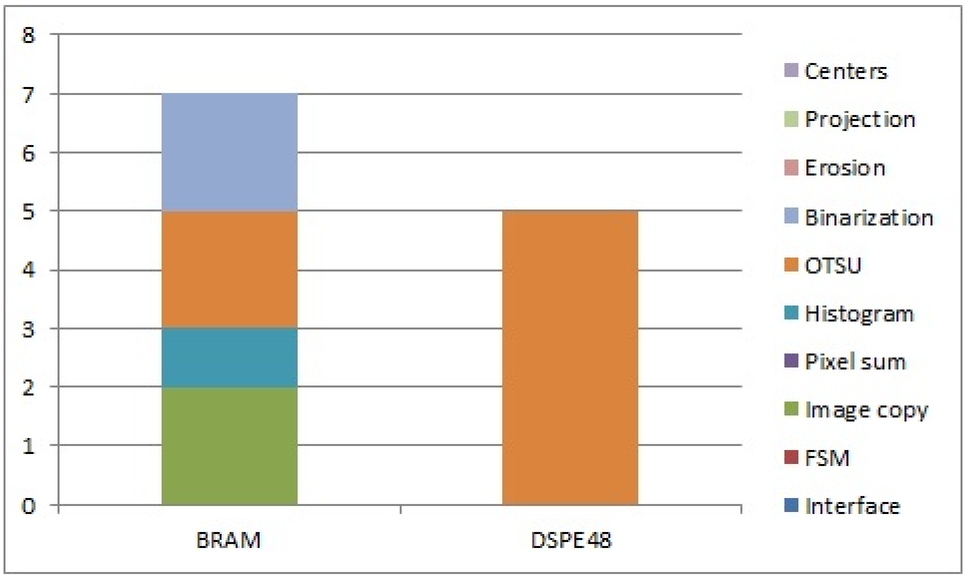


Figure 5.2: BRAM and DSP utilization by manual FFoS implementation

The only kernel requiring DSPE48s is the OTSU kernel, which can be separated in a CH/CIA part and the main threshold calculation. The cumulative histogram computation requires a 19x8 multiplier, the threshold calculation uses two 19x13 bit multipliers and one 21x21 multiplier. Here too, the implementation is not fully optimized. The DSPE48s multipliers support up to 25x19 multipliers, meaning an extra DSPE48 is used for the 21x21 multiplier while this can be avoided by implementing two bits of the multiplication in logic.

Flip flop utilization is dominated by the OTSU kernel, with as main bottleneck the two 19/13 bit dividers in the design. The projection kernel also requires a significant amount of flip flops due to its parallel implementation. LUT utilization is dominated by the same two kernels, the OTSU kernel as a result of the two dividers and the projection kernel due to the massive data parallelism requiring many adders to accumulate a 120-bit line in a single cycle and bitwise adding two 120-bit lines in a single cycle.

A complete overview is given in table 5.1, presenting a summary of the FFoS application after

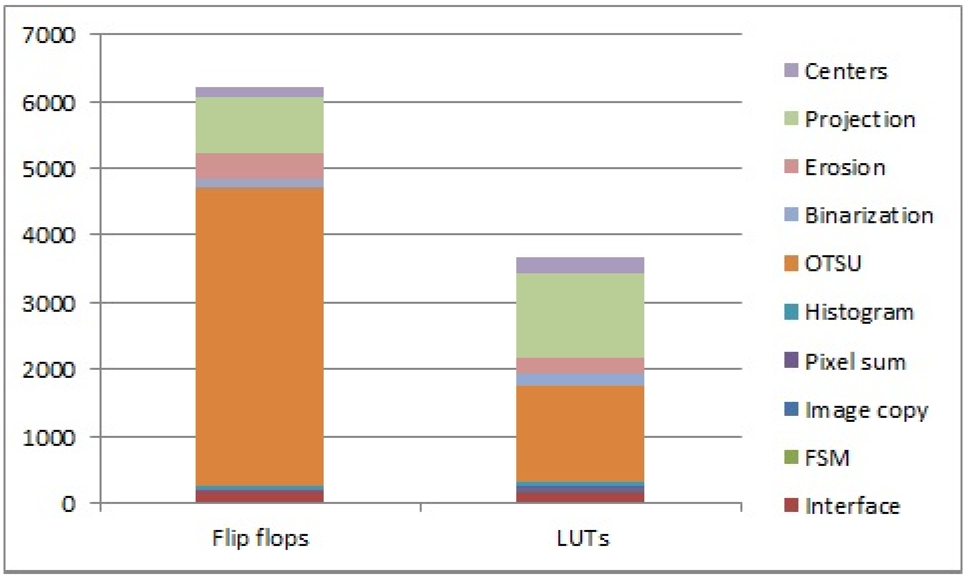


Figure 5.3: Flipflop and LUT utilization by manual FFoS implementation

logic synthesis. This summary table will be gradually updated for each HLS implementation. Note that after logic synthesis of the complete design (instead of modules synthesized separately), more optimizations might be available, or more logic for signal routing might be necessary and the result can slightly differ from the total resource utilization shown in figures 5.2 and 5.3.

Table 5.1: Summary performance results FFoS implementation

Implementation	Flip flops	LUTs	BRAM	DSP	Latency	Clock period	Design time
Manual	6131	4042	6	5	11895	7.556	30 days

5.2.3 Latency

Figure 5.4 shows a latency breakdown for the FFoS implementation. The histogram generation, image copy and sum of pixel values are executed in parallel. The implementation leaves room for several optimizations, such as merging CH/CIA and Sigma calculation, using dual ported block RAM for the image storage to half the number of cycles required for binarization and the use of streaming to overlap kernel execution. The erosion kernel, using a 3x3 window, is executed twice to achieve the result of a 5x5 window while keeping the resource utilization reduced.

5.3 AutoESL Implementation

The AutoESL implementation is explained next. The sections are separated into a section explaining the input specification, and two sections discussing the resource utilization and latency of the design. The design is made to match the manual design as close as possible with regard to the architecture of each kernel. A later section deals with possible modifications to improve performance issues highlighted in this section.

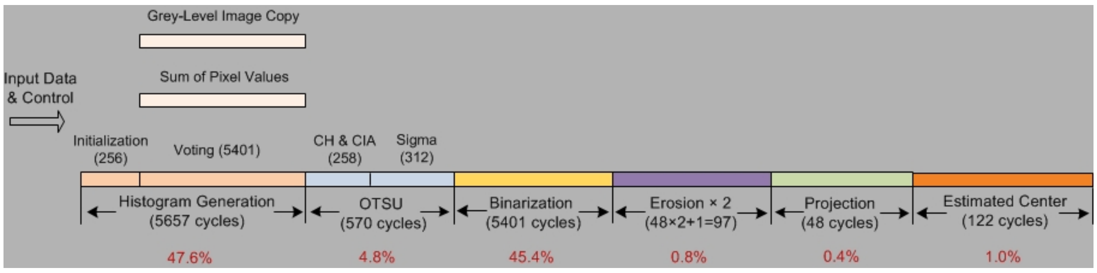


Figure 5.4: Latency distribution for the manual FFoS implementation

5.3.1 Input Specification

As explained in section 5.2, creating modularity in HLS is more complicated. In the case of AutoESL, histogram, image copy and pixel sum are all operations written in a separate function each containing a loop iteration over the number of pixels to process. To enable the parallel execution of these kernels, AutoESL requires the loop bodies to be in a single loop requiring the user to merge all loops and implement the 3 kernels as a single function. AutoESL will not automatically extract inter-task parallelism, and modifications by the user are required to enable parallelism. Next to this parallelization issue, AutoESL generates an FSM for each loop instead of one top-level FSM.

The input specification is written in C and involves a main function which calls all intermediate kernels. The interface is specified with pragmas to inform AutoESL that the input and output of the design should be synthesized in such a way that it is compatible with FSL. With regard to figure 5.4, each block in the horizontal direction is implemented as an outer loop possibly containing inner loops.

5.3.2 Resource Utilization

Figures 5.5 and 5.6 present the logic distribution across the application kernels. The architecture of the manual kernel implementations is mimicked as close as possible, implementing the same amount of parallelism and expensive functional units such as dividers and multipliers.

From the nine block RAMs, six are used for image storage. The read-modify-write operation in erosion, as explained in section 4.2.2, requires the block RAM to be dual ported and disables to possibility to use the same block RAM for the dual execution of erosion. This implementation approach for erosion thus requires four block RAMs, compared to two block RAMs in the reference design. Note that it is important that the designer selects the proper block RAM description to be generated in order to limit the number of synthesized block RAMs. Selecting the wrong block RAM properties results in more latency due to a limitation on the ports or the introduction of more block RAMs than expected. Logic synthesis of the complete application can reduce the required number of block RAM due to merging.

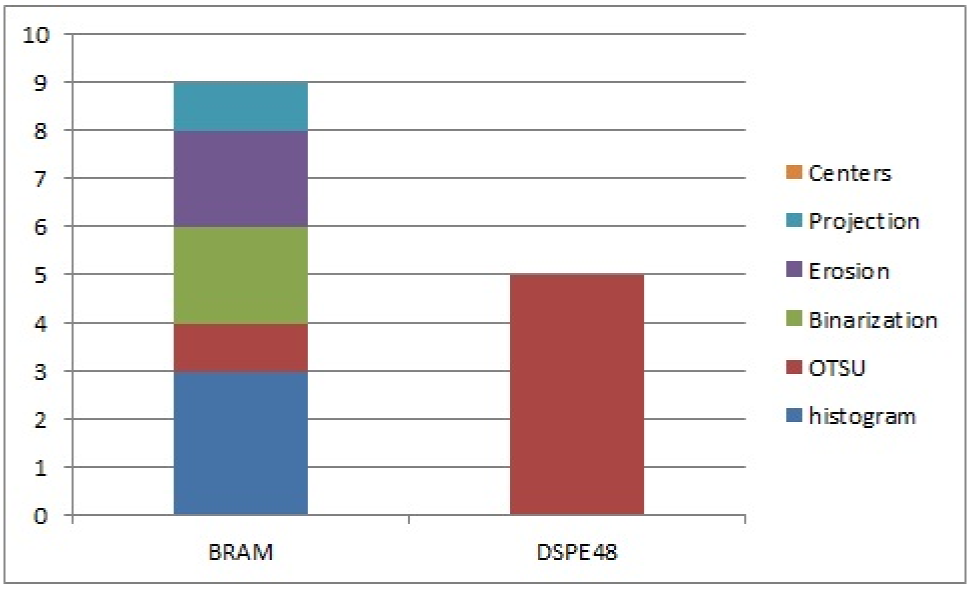


Figure 5.5: BRAM and DSP utilization by AutoESL FFoS implementation

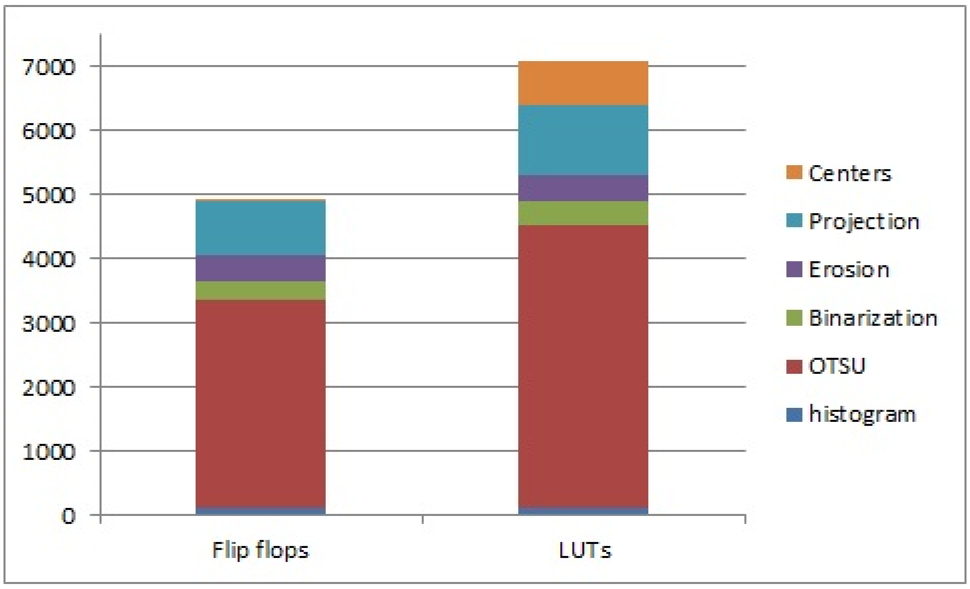


Figure 5.6: Flipflop and LUT utilization by AutoESL FFoS implementation

Again, the only kernel requiring DSPE48s is the OTSU kernel, which can be separated in a CH/CIA part and the main threshold calculation. The cumulative histogram computation requires a 19x9 multiplier, one bit more compared to the manual implementation because the second operand is based on a loop iterator which requires an extra bit to prevent overflow during the exit condition check. The sigma (threshold) calculation is optimized using bit-casting to use two 19x13 multipliers and one 21x21 multiplier. Since the arbitrary precision library from AutoESL allows the access of specific bits and supports bit-casting, the 21x21 multiplier can be reduced to occupy just one DSPE48.

Flip flop utilization is again dominated by the OTSU kernel, although requiring less flip flops compared to the manual design. The two dividers in AutoESL, also implemented as 19/13 bit, require much more LUTs compared to the dividers in the manual which are generated by CoreGen. A complete overview is given in table 5.2, presenting a summary of the complete FFoS application after logic synthesis. To achieve a logic distribution graph, each kernel needs to be synthesized separately requiring the option in AutoESL to not inline the kernels in the top function and thus limit resource sharing. The overall trend is visible that less flip flops but much more LUTs are required. Section 5.5 discusses these issues in more detail and section 5.6 will present some solutions to this issue. It can also be observed that indeed the number of required block RAMs reduces by one because of more room for optimizations when synthesizing the complete design.

Table 5.2: Summary performance results FFoS implementation

Implementation	**Flip flops**	**LUTs**	**BRAM**	**DSP**	**Latency**	**Clock period**	**Design time**
Manual	6131	4042	6	5	11895	7.556	30 days
AutoESL	4967	7187	8	5	12073	7.832	5 days

5.3.3 Latency

Kernel latency in AutoESL consists of three parts. First, the actual latency of the loop itself is based on the number of loop iterations, pipeline initiation interval and schedule length. Secondly each loop which is not pipelined, usually the outer loop when dealing with a 2-nested loop, requires one loop entry and one loop exit cycle adding two cycles to the kernel latency. Finally, when a kernel (function) is not inlined in the top-level design, two extra cycles are required for function entry and exit. This can be compared to the caller/callee principle in CPU programs where some overhead is encountered to store information such that the program returns to correct point in the caller once the callee is done executing.

Figure 5.7 shows a latency breakdown for the FFoS implementation. The breakdown includes 2 cycles for each kernel as a result of the loop entry and exit. In total, four kernels in the final implementation are not inlined to either enable the re-use of that hardware block (executing the same erosion hardware block twice) or to prevent resource duplication because the tool inefficiently shares resources. This means the final latency includes 8 more cycles for function entry and exit for the non-inlined functions. The observations done in section 5.2.3 with regard to possible optimizations remain valid and will be further explored in section 5.6.

A latency issue is shown in the binarization kernel, requiring 181 cycles more compared to the manual implementation. This issue is explained using the input specification shown in listing 5.1. A dual nested loop is shown, with the inner loop specified as pipelined with an initiation interval of 1. The result of this input specification is a 2-stage pipeline for the inner loop which is in turn controlled by the outer loop. Being a 2-stage pipeline, the inner loop has an iteration count of 120 and will require 121 cycles to complete. In the case the specification would be a perfectly nested loop, the loops would automatically be flattened for a total latency of (45*120)+1 cycles. However, the loop is now imperfect as the outer loop also contains a loop body. Even worse, the loop body of the outer loop involves a store to a wide block RAM to support enough bandwidth to the erosion kernel. As seen in section 4.2.2, this requires a read-modify-write operation using 3 clock cycles. As a result the latency for this binarization kernel is (45*121)+(45*3) cycles, a total of 5580 when ignoring the loop entry/exit cycles. As suggested before, a possible solution to improve this latency is to unroll the inner loop twice and use a dual ported block RAM as input.

```
for(y=0;y<HEIGHT;y++){
    for(x=0;x<WIDTH;x++){
    #pragma AP PIPELINE II=1
        tmp.set_bit(x, (img[y*WIDTH+x]>=threshold));
    }
    bin_img[y] = tmp;
}
```

Listing 5.1: Binarization source code with data preparation

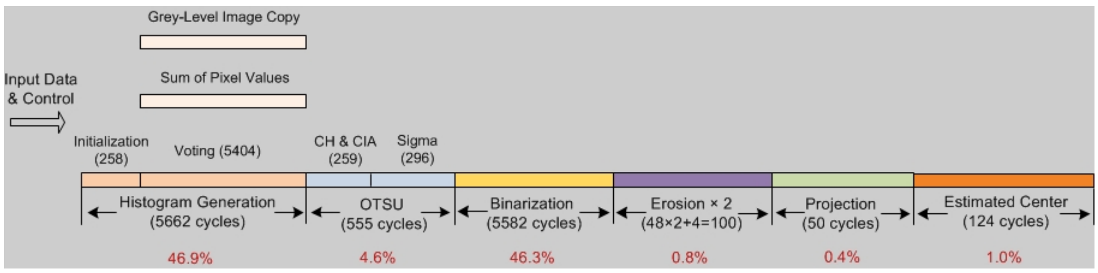


Figure 5.7: Latency distribution for the AutoESL FFoS implementation

5.4 Synphony Implementation

The Synphony implementation is explained next. The sections are separated into a section explaining the input specification, and two sections discussing the resource utilization and latency of the design. The design is made to match the manual design as close as possible with regard to the architecture of each kernel. A later section deals with possible modifications to improve performance issues highlighted in this section.

5.4.1 Input Specification

Synphony has an identical problem to AutoESL with regard to design hierarchy. Each loop is implemented as a hardware block, enabling a designer to separately analyze functions. Interface and control (FSM) however is at top-level containing all hardware blocks and can not be synthesized separately. Although Synphony is able to automatically schedule kernels (PAs) in parallel if there are no data dependencies, histogram, image copy and pixel sum are merged into a single loop to enable parallel execution since the stream interface disables automatic parallelization of the PAs. Note that connecting a stream interface to multiple parallel execution modules is possible at RTL level, but coding stream accesses in C++ limits the number of stream accesses per input to one and thus requiring the kernels to be implemented in the same loop. The interface again differs slightly from the reference as FSL is not supported in Synphony. The data input and output are streams however with identical bit-widths.

The input specification is written in C and optimized to approach the same kernel latencies as the manual and AutoESL designs. The structure of the input specification is identical, but Synphony requires a different way of describing parallelism and has limits on bit-casting which reduces the performance of the OTSU kernel. Wide vectors are described using the *packed data* construct as discussed in section 4.2.3. Because of lack of time and the proper SystemC package, bit-casting is not applied in the OTSU kernel, resulting in dividers and multipliers which could potentially be optimized in area. Synphony does not support standard interfaces such as FSL and requires the user to describe a stream function in combination with pragmas to synthesize the input and output of the design as streaming. Finally, since Synphony is able to extract inter-task parallelism and schedules PAs simultaneously whenever there are no data-dependencies, the estimated center kernel is left as two separate loops each iterating over a projection vector to find the X and Y location of the centers.

5.4.2 Resource Utilization

Figures 5.8 and 5.9 present the logic distribution across the application kernels. Note that these results are from separate kernel synthesis and do not include top-level control and interface. For synthesis the tool Synplify from Synopsys has been used, as the generated dividers can not be synthesized by Xilinx ISE. The implementation with regard to amount of parallelism and number of expensive functional units such as dividers and multipliers is identical to the AutoESL and manual implementation.

In total, nine block RAMs are utilized. Although erosion is not implemented using a read-modify-write operation, extra block RAMs are still used for this kernel. The input and ouput for erosion can be the same block RAMs, however Synphony implements the storage as 4 block RAMs with shorter word width as the AutoESL and manual implementation. The AutoESL block RAMs are implemented as 72-bit wide and the Synphony block RAMs in this case are implemented as 36 bit word size, requiring twice the amount of block RAMs for the same binarized image. Although Synphony provides control over the number of read and write ports, this had no influence on the logic synthesis results. After logic synthesis of the complete application, no more block RAMs are merged and the number of utilized RAMs remains 9, indicating a throughput oriented architecture template.

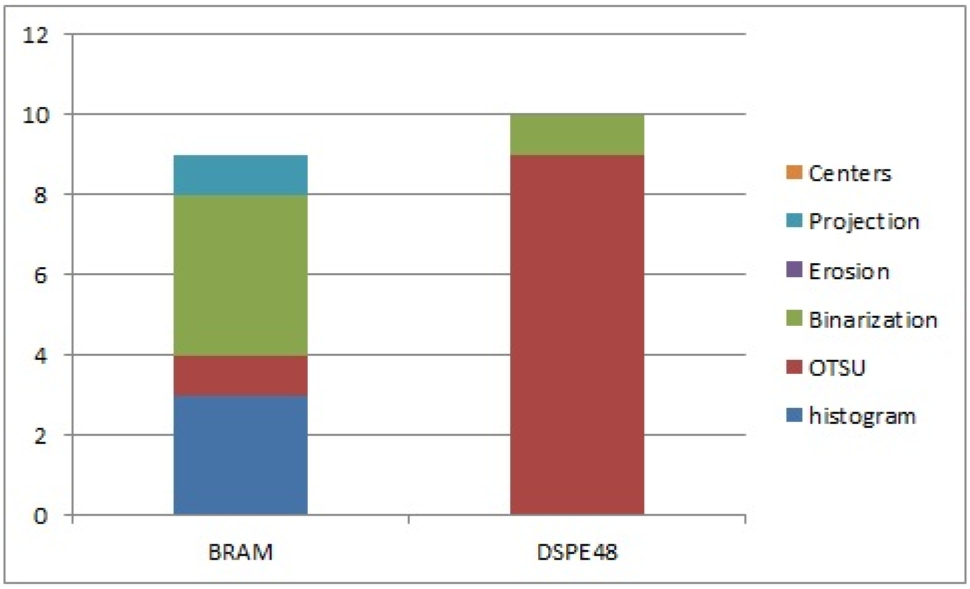


Figure 5.8: BRAM and DSP utilization by Synphony FFoS implementation

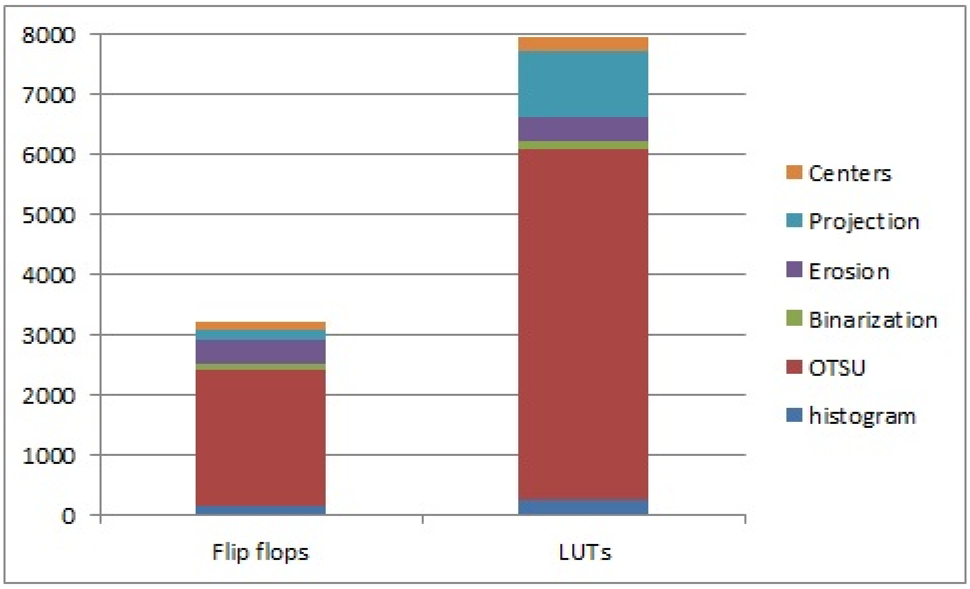


Figure 5.9: Flipflop and LUT utilization by Synphony FFoS implementation

A factor two more DSPE48s are used by this implementation, not limited to the OTSU kernel compared to the manual and AutoESL design. Binarization now requires a DSP, used for the address calculation to access the image. The image index is calculated as $y*WIDTH+x$, with $WIDTH$ a statically defined value. Although this statement could be calculated with a simple adder, Synphony uses a multiplier to calculate this index when the loops are not perfectly nested even though the compiler can theoretically optimize the index calculation. Note that AutoESL is able to extract this information for in-perfectly nested loops. The lack of bit optimizations in the threshold calculation results in the usage of extra DSPs. Using the correct SystemC library in Synphony can prevent this issue.

The computationally intensive OTSU kernel again dominates the flip flop utilization. The large differences between this design and the AutoESL design are contributed by the divider implementations in both tools. Section 5.5 will elaborate further on this as it is an issue observed in both HLS tools. The second largest kernel in LUT utilization is the projection kernel, which requires many parallel adders. The figure shows a large decrease in flip flop utilization, but this is due to the architecture template in which large vectorized data (such as used in the projection kernel) is implemented in the top-level design and can not be observed at kernel level.

A complete overview is given in table 5.3, presenting a summary of the FFoS application after logic synthesis. To achieve a logic distribution graph, each kernel needs to be synthesized separately ignoring top-level control logic and top-level instantiations of register based memories. The overall trend is visible that even less flip flops are required but much more LUTs are required compared to both manual and AutoESL designs. Section 5.5 discusses these issues in more detail and section 5.6 will present some solutions to solve them.

Table 5.3: Summary performance results FFoS implementation

Implementation	**Flip flops**	**LUTs**	**BRAM**	**DSP**	**Latency**	**Clock period**	**Design time**
Manual	6131	4042	6	5	11895	7.556	30 days
AutoESL	4967	7187	8	5	12069	7.832	5 days
Synphony	4727	9646	9	10	11886	19.197	5 days

5.4.3 Latency

Synphony does not require loop and function entry/exit cycles and the latency is simply the latency of all kernels combined. Figure 5.10 shows a latency breakdown for this FFoS implementation. Histogram includes copying the image from the stream interface and calculating the sum of pixels. The automatic choice by Synphony to use multi-cycle multipliers, which can be overridden by the user when required, results in small latency differences in the OTSU implementation. The binarization kernel does not suffer from a read-modify-write operation and can be more effectively pipelined even though it uses a DSP for the multiplication in the address calculation. Erosion and projection both operate on complete binarized image lines and each takes 47 cycles to complete. The estimated center kernel is separated into two processing arrays executing in parallel, one for searching through the vertical and one for searching through the horizontal projection vectors. The latency for the estimated center kernel is dominated by the search through the vertical projection vector which is 120 elements big.

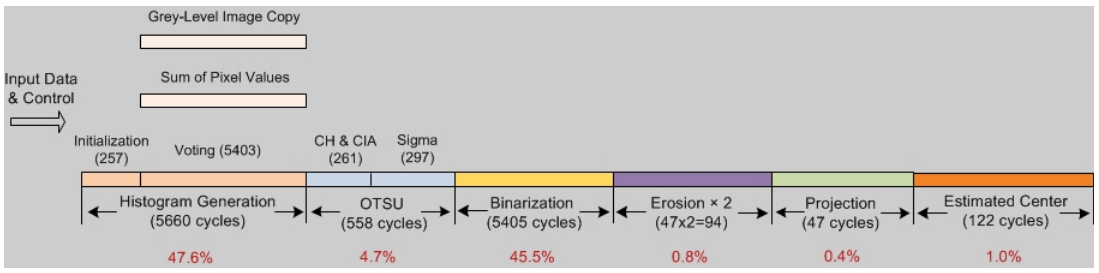


Figure 5.10: Latency distribution for the AutoESL FFoS implementation

5.5 Discussion

In the previous sections, two HLS implementations have been created using the HLS tools AutoESL and Synphony C Compiler. The designs have been kept as identical as possible to the manual implementation. This section summarizes and elaborates on some performance issues encountered in the HLS designs.

5.5.1 Flip flops and LUTs

Previous sections have shown it is not always possible to extract accurate information from synthesis of separate kernels. Reasons for this vary from the RTL generation style of Synphony and the optimization pragma in AutoESL to enable or disable function inlining. Besides this issue, table 5.3 has shown clear differences in utilized area. The most expensive operation throughout the application is the division.

The OTSU algorithm is implemented using two dividers, and since there are many different divider implementations possible with each it's own trade-off figure 5.11 presents resource utilizations for several divider implementations. Note that lack of bit-width optimizations in Synphony results in the use of a 32/32 bit divider in the actual design. Latency of all dividers is 34 cycles. Although Synphony allows the user to specify a latency for the divider, the number of pipeline stages for the actual division remains 8 resulting in the use of less flip flops but a very high minimum clock period because of a longer combinatorial path.

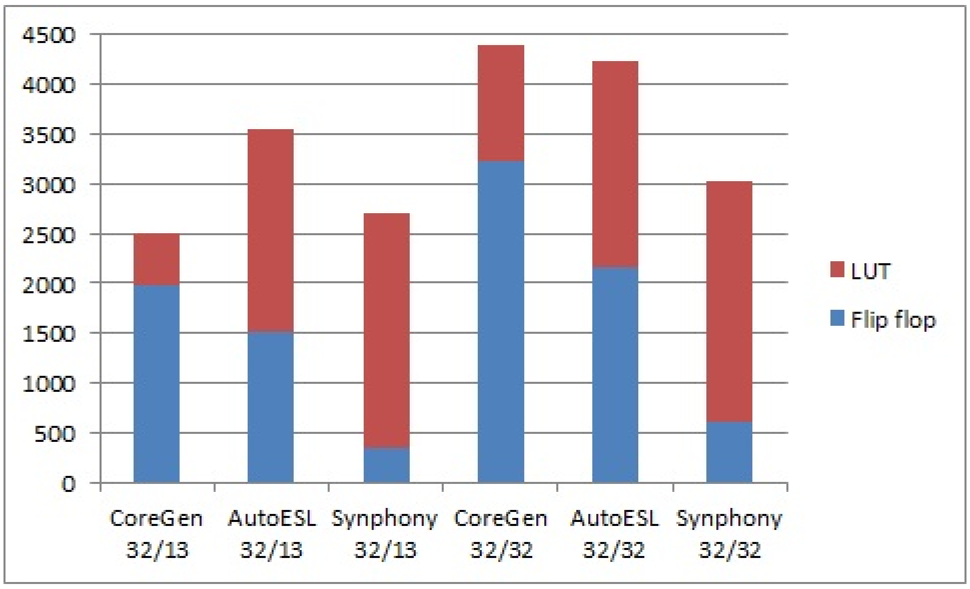
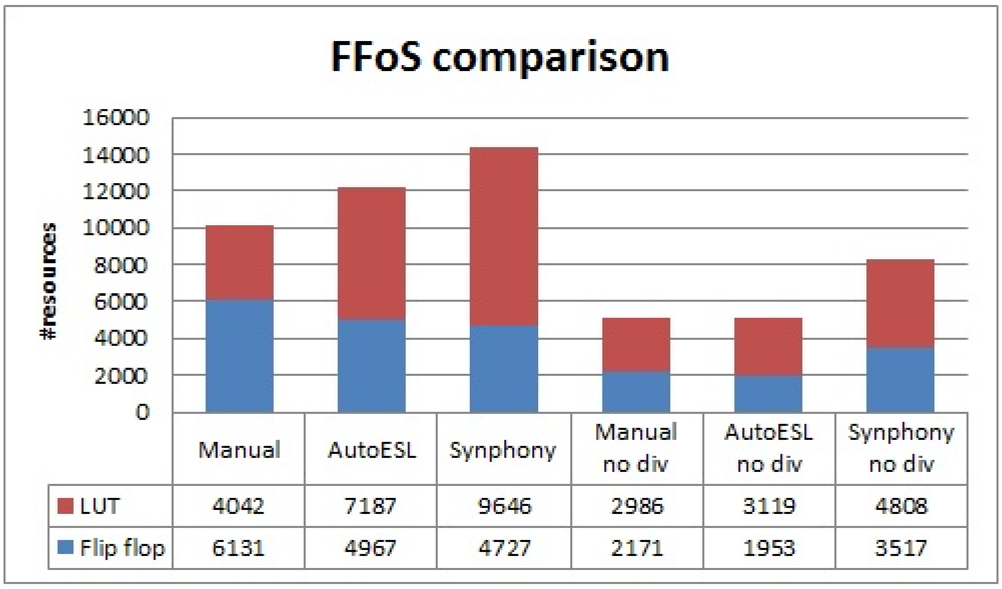


Figure 5.11: Resource utilization for divider implementations

Because of the large differences in the divider component, figure 5.12 presents the flip flop and LUT utilization with the divider resources subtracted. The manual and AutoESL implementations now converge with regard to resource utilization. With AutoESL, different levels of resource sharing can be quickly explored to find the best solution by inlining functions into the top-level design and selecting between three different levels of effort to share resources. Of course, too much resource sharing will result in an increase of resources because of the multiplexer costs and inlining functions which are executed multiple times will result in hardware duplication. In this case, ignoring divider costs and taking extra interface cost for the reference design into account, AutoESL achieves a design requiring around 5% less flip flops and 4% more LUTs. The Synphony design requires 62% more flip flops and 61% more LUTs.



	Manual	AutoESL	Synphony	Manual no div	AutoESL no div	Synphony no div
🟥 LUT	4042	7187	9646	2986	3119	4808
🟦 Flip flop	6131	4967	4727	2171	1953	3517

Figure 5.12: Resource utilization for FFoS with and without divider resources

The Synphony implementation still requires more area, which is a result of several factors. First, the area overhead resulting from the PPA architecture has a significant effect on the resource utilization as less resource sharing is applied and each processing array has logic included to support stalling even though stalling is not necessary. The separated estimated center kernel, which is implemented in two processing arrays, results in extra area for an extra processing array. To solve this, a minor source transformation can merge the loops processing the horizontal and vertical projection vectors at the cost of a comparator. The comparator prevents the index of the horizontal projection vector to go out of bound as it is smaller than the vertical projection vector. Next to this, each call to the erosion kernel results in a separate processing array for the erosion kernel, meaning this implementation will have two copies of the erosion hardware block. This can be avoided by building the erosion kernel separately as a TCAB design and instantiate the TCAB in the FFoS application. This requires more effort though compared to disabling inlining on the erosion function in AutoESL. And finally, the most computationally intensive kernel OTSU has a longer pipeline depth and lacks SystemC bit-casting resulting in more and wider pipeline registers.

5.5.2 DSPs

The number of DSPs used in the manual and AutoESL implementation are equal and can both be reduced by one by limiting the second operand of the 21x21 bit multiplier. The same number of DSPs is achievable with Synphony but is not implemented because of lack of time and the improper SystemC package. Note that the register and LUT count will also decrease in the Synphony design when variables are correctly cast to the correct bit-width.

5.5.3 BRAM

Both HLS designs use more block RAMs. In AutoESL, the erosion kernel requires multiple block RAMs to separate input from output to be able to pipeline the design with an initiation interval of 1. Having the same block RAMs assigned both for input as for output would slow down the design as the read-modify-write operation for the output would require more than two ports. Although Synphony is able to use the same block RAMs for the binarization and erosion kernels it implements block RAMs with smaller word size due to limited porting, still requiring the same amount of block RAMs for the binarization and erosion implementation. Logic synthesis by Xilinx ISE of the AutoESL implementation shares one more block RAM to achieve 8 block RAM utilization instead of 9, whereas less block RAMs are shared in the Synphony implementation.

5.5.4 Latency

Besides the slightly slower binarization implementation in AutoESL, latencies between different implementations are just off by a few cycles. Differences are caused by loop/function entry and exit cycles in AutoESL and differences in pipeline depths between all implementations. The depth of the pipeline in HLS designs can not be set manually and is dependent on the loop body in the C description and the component libraries from the HLS tool in question. The depth is mainly dependent on number of block RAM accesses (on the same RAM) and the functional units which are required in the data-path. Although in AutoESL the preparation of the 120-bit words in the binarization kernel results in a slightly slower design, the problem can be solved by either reducing the width of the word size or unrolling the loop which iterates over the width of the image. The second optimization can be made with almost no area cost, processing two input pixels at the same time requires a dual ported block RAM and some logic to access the data and compare the pixel value. This optimization has no influence on kernels further in the vision pipeline, whereas limiting the degree of parallelism does.

5.5.5 Clock Period

The minimum clock period of the AutoESL design remains within 4% of the reference. The 8-stage divider in Synphony to operate on 32-bit wide operands result in an unacceptably high minimum clock period. Feedback on this issue is given to Synopsys but so far no cause and/or possible solutions is received.

5.6 Alternative Implementations

Issues encountered so far are the costly divider implementations, the CH/CIA computation which can be done more efficiently, the bad performance of binarization in AutoESL due to in-perfect nested loops in combination with the read-modify-write operation and the separate PAs for the finding of the centers in the projection vectors using Synphony. In section 5.6.1 an alternative HLS implementation is presented to solve these issues. Section 5.6.2 introduces the notion of streaming implementations, using streams to improve throughput limiting the degree of data parallelism while increasing instruction level parallelism (or inter-task parallelism).

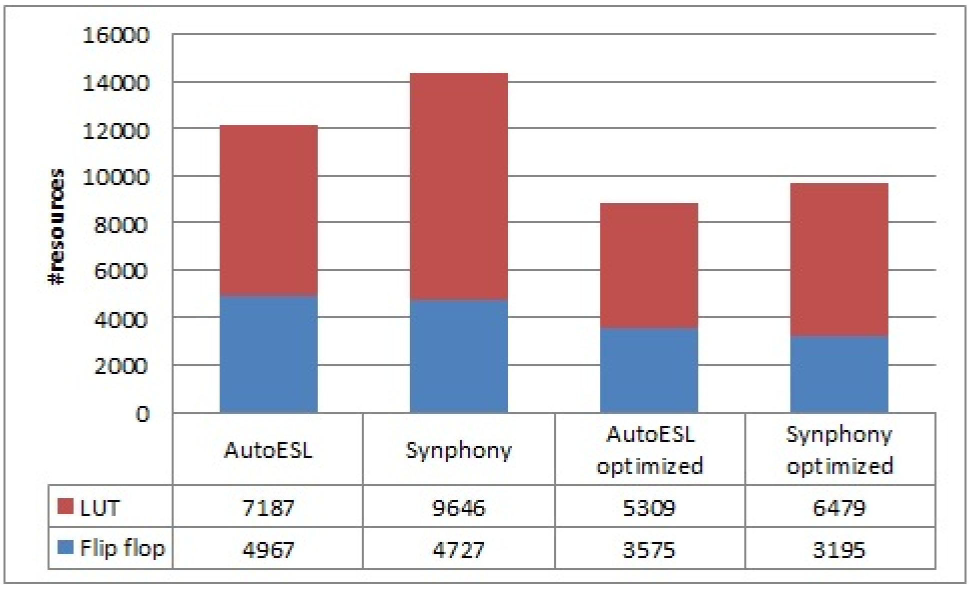
5.6.1 Resolving Highlighted Issues

Several issues were highlighted in the previous HLS implementations. This section shortly describes the performance gain which can be achieved with simple source modifications to resolve several of the mentioned issues. For both AutoESL and Synphony the same changes are made to observe the performance gain. With regard to the reference implementation, the following changes are made:

- The number of dividers is limited to one, trading decreased area for increased latency.
- The CH/CIA block RAMs are removed by merging the cumulative CH/CIA calculations inside the Sigma (threshold) computation.
- The binarization loop is unrolled twice to exploit the advantage of a dual ported block RAM. It is expected that this will improve the latency at a small area cost.
- In Synphony, the estimated center loops are merged to be implemented in a single processing array. The loops are already merged in the AutoESL implementation.

Note that the first two optimizations will influence each other. While the latency increases because the number of functional units is limited in the threshold calculation, the latency can expected to remain stable because the second optimization removes 256 loop iterations which are merged in the threshold calculation. The latency for the OTSU kernel will depend on the pipeline-ability of the threshold calculation with only one divider in the datapath. Although binarization does not perform bad in the Synphony implementation, the image block RAM is made dual ported and the binarization kernel will process two pixels at the same time to keep both HLS design comparable. Note that the optimizations made will have great influence on the performance results but the effort needed to apply the changes involve only minor code changes or even simple code movement such as the loop merging in estimated center.

Figure 5.13 presents the reduced area in flip flops and LUTs compared to previous HLS designs. An average reduction of 30% is observed with regard to flip flops and LUTs which is mainly contributed by the reduced number of dividers. The threshold kernel itself however includes more registers compared to the previous threshold kernel as now the CH/CIA values need to forwarded through the pipeline stages, whereas previously they were already stored in block RAM. By moving the CH/CIA computation into the threshold calculation, each implementation requires one less block RAM.



	AutoESL	Synphony	AutoESL optimized	Synphony optimized
🟥 LUT	7187	9646	5309	6479
🟦 Flip flop	4967	4727	3575	3195

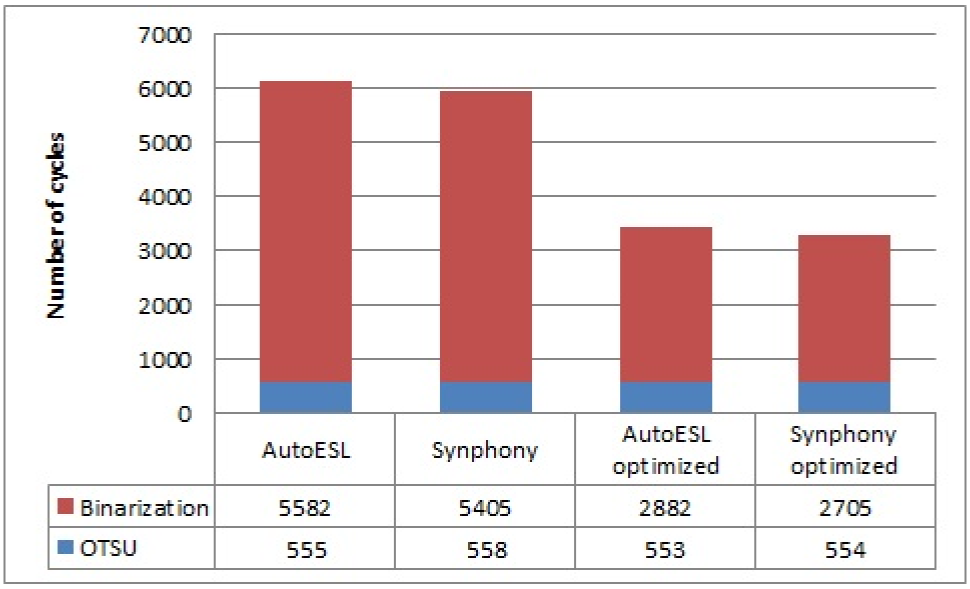
Figure 5.13: Comparison of reference HLS implementation and optimized implementations

The binarization and OTSU latencies are visualized in figure 5.14. As expected, the binarization latency is nearly halved because the kernel processes two pixels per loop iteration and thus the number of iterations is halved. The dual ported block RAM supports enough bandwidth to efficiently pipeline the binarization kernel. More surprisingly is the small reduction in OTSU latency. Both threshold calculation pipelines now have an initiation interval of 2 because the number of dividers is reduced. The reduction in latency is because of the decreased pipelineability is absorbed by the merging of CH/CIA in the threshold pipeline. This way, the AutoESL design gains 2 cycles caused by the reduction of a loop entry and exit cycle. The Synphony design improves by 4 cycles, caused by the removal of the CH/CIA loop which contained a 4 cycle multiplier which is now merged into the threshold calculation. Note that latency of other kernels is not affected by the changes applied in this implementation.

Using minor source modifications and tool specific architectural parameters, area has been reduced significantly compared to the previous HLS designs. Next to area reduction, there is a small decrease in latency as well with as main contributor the binarization kernel. The increased processing time for threshold calculation could be hidden by moving the CH/CIA computation inside the threshold computation, simultaneously reducing latency by removing the cycles required for CH/CIA computation beforehand and increasing latency by reducing the threshold calculation pipelineability. In the end, there is a gain of just a few cycles for the OTSU optimization but the main advantage of the optimization is the significant reduction in area without sacrificing latency. The speedup of the binarization kernel is almost free, as the operation performed on the input pixels is simply a comparison. The extra area required only involves a comparator and block RAM related logic to address and access the data on the second port.

5.6.2 Streaming

Previous implementations of the FFoS application did not contain inter-task parallelism by applying streaming communication between kernels. This section shortly describes a streaming implementation made for AutoESL and Synphony. Because many vision algorithms are simple hardware pipelines where kernels operate on image data, often these kernels can be implemented in such a way that each kernel can produce and consume a data element each cycle to enable a succeeding kernel to start execution as soon as it receives the data on its input. In the case of FFoS, there are 3 kernels which can be implemented in a streaming fashion. The binarization,



	AutoESL	Synphony	AutoESL optimized	Synphony optimized
🟥 Binarization	5582	5405	2882	2705
🟦 OTSU	555	558	553	554

Figure 5.14: Binarization and OTSU reduced latencies

erosion and projection kernel can all be implemented to fetch one pixel per cycle and produce one pixel per cycle. Such an implementation allows the three kernels to run in parallel to increase throughput without requiring massive data parallelism to speed up the design. However, a combination of data parallelism and streaming can also be implemented in which the FIFOs connected between the kernels contain wider words to support the kernel to operate on multiple pixels at the same time.

The next sections present different aspects of two streaming designs. One design streams 8-bit pixels and the other design streams complete lines (120 bit wide FIFOs).

Input Description

Enabling the possibility of streaming requires a bit more effort to transform the code in such a way that the streaming kernels consumption and production rates match. The erosion kernel is re-written to support a 5x5 window instead of a 3x3 window. In the previous AutoESL design the erosion kernel, based on the 3x3 window, could be implemented as a separate hardware block enabling re-use of that hardware block. In a streaming implementation, a 5x5 window operation implemented as two executions of 3x3 window operations would require two instantiations of the 3x3 hardware block. The new erosion description is re-written to use line buffers and a window buffer, similar as shown in figure 5.15.

Both AutoESL and Synphony can implement a streaming design in which binarization, erosion and projection are connected with FIFOs using 8-bit elements as communication tokens. The input descriptions for AutoESL and Synphony are similar, besides some tool specific notations for expressing the use of FIFOs and resource selection. It is also possible to replace the erosion and projection kernel descriptions which should operate in streaming fashion with the parallel descriptions of these kernels. Although theoretically possible, Synphony is the only tool able to correctly synthesize this design. Such a design is not yet implemented in AutoESL, but seems feasible. The architecture of the remaining kernels is identical to the implementation discussed in section 5.6.1. The parallel description of erosion is similar to the one discussed in section 4.2.3, only extended to support a 5x5 window by introducing two extra line buffers.

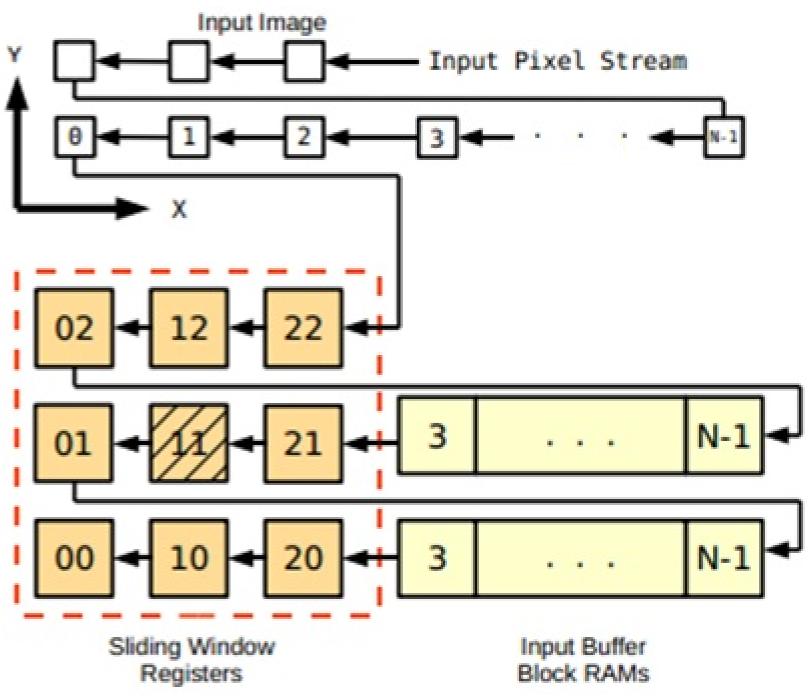
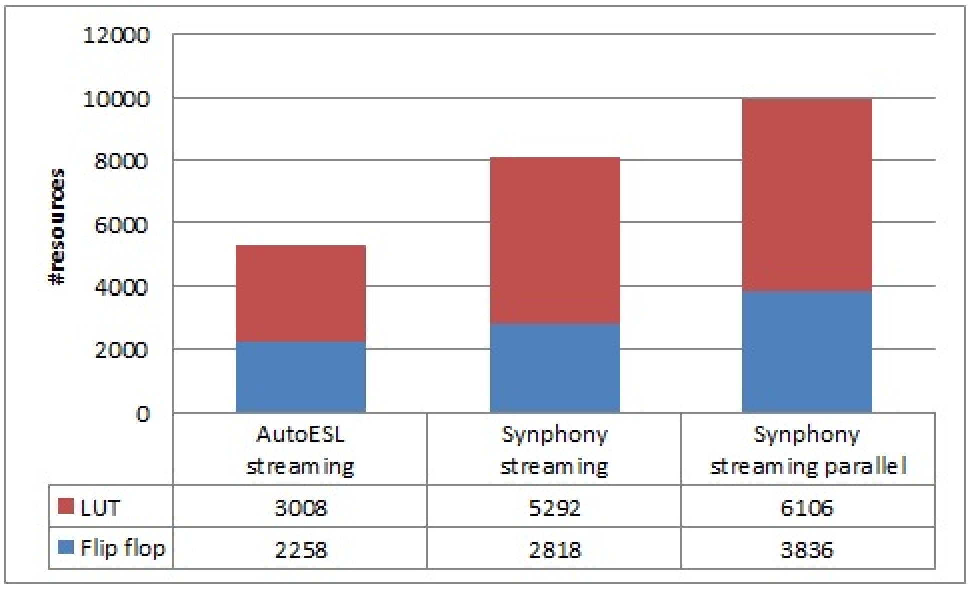


Figure 5.15: Architecture to support streaming in window based operations

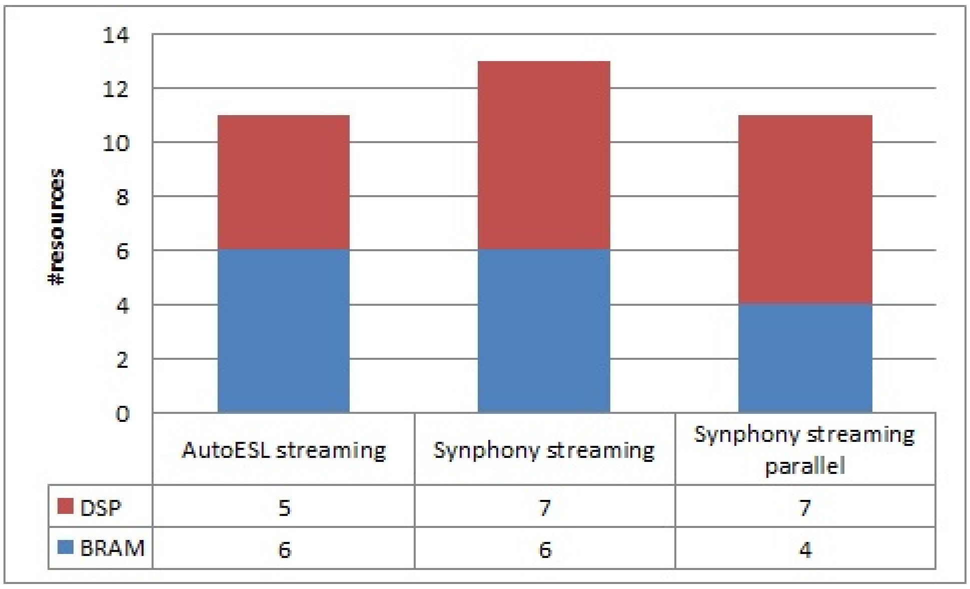
Resource Utilization

Figures 5.16 and 5.17 show the resource utilization after logic synthesis for the 8-bit streaming implementation in AutoESL and Synphony and the 120-bit streaming implementation in Synphony which has a parallel implementation of erosion and projection. All designs include one divider to limit the amount of resources.



	AutoESL streaming	Synphony streaming	Synphony streaming parallel
🟥 LUT	3008	5292	6106
🟦 Flip flop	2258	2818	3836

Figure 5.16: Comparison of different streaming implementations: flip flops and LUTs



	AutoESL streaming	Synphony streaming	Synphony streaming parallel
🟥 DSP	5	7	7
🟦 BRAM	6	6	4

Figure 5.17: Comparison of different streaming implementations: DSP and BRAM

Again it is shown that AutoESL has the most efficient implementation of the two HLS tools, even though the divider in Synphony uses 30% less flip flops and only 18% more LUTs. Although the intermediate binarized image storage before and after erosion are now replaced by FIFOs, block RAMs in the 8-bit streaming implementation are now used for line buffering. Depending on the constraints set, these line buffers can also be implemented in either flip flops or distributed RAM. The streaming implementation with parallel kernels uses less block RAMs as the line buffers can no longer be implemented in block RAM, hence the increase in flip flops.

Latency

To highlight the principle, figures 5.18 through 5.20 show the latency distribution for the streaming implementations. The streaming process denotes the overlapped execution of binarization, erosion and projection. The 8-bit streaming AutoESL performs slightly worse because of an imperfect nested loop in the projection implementation similar to the binarization latency issue discussed in section 5.3. The imperfect nested loop causes the inner loop pipeline to stall, causing the other kernels to stall as well. An imperfect loop is scheduled more efficiently in Synphony. In both pipelines of kernels (the stream process), the bottleneck limiting throughput is the erosion implementation which needs to prefetch several lines of the binarized image before it can start with producing outputs. The streaming 120-bit implementation has a better throughput, only now it is limited by the limited amount of parallelism in the binarization kernel as the input data there is too wide to heavily parallelize. The erosion and projection kernels each take 47 cycles in this implementation, and binarization processes two pixels at the same time to limit the bottleneck. Observe that each purple and green block represent the erosion and projection kernel in figure 5.20. The erosion and projection kernels each iteration process a line, and then block until the binarization kernel is finished with the production of a line. It depends on the constraints if such an implementation is an acceptable approach, as producer and consumer rates differ a lot due to the large difference in number of communication tokens.

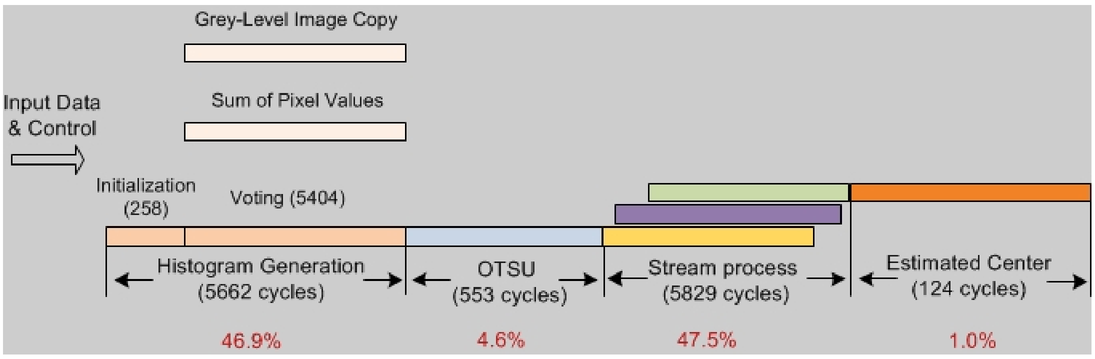


Figure 5.18: Latency distribution streaming 8-bit in AutoESL

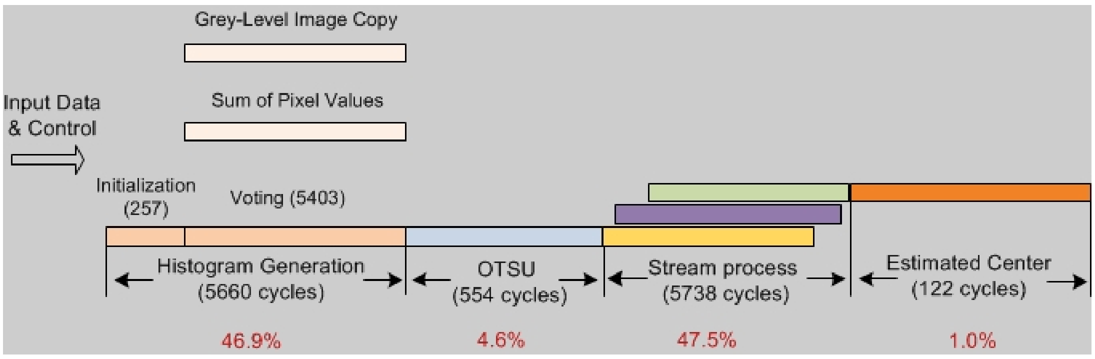


Figure 5.19: Latency distribution streaming 8-bit in Synphony

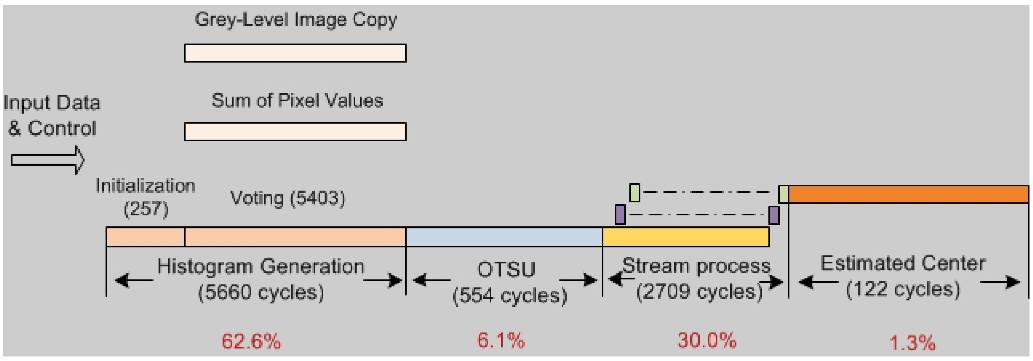


Figure 5.20: Latency distribution streaming 120-bit in Synphony

5.7 Conclusion

In previous sections, several implementations for the FFoS application have been created. The first implementation mimicked the manually written design as close as possible, including as much expensive functional units such as dividers and multipliers and exploiting as much parallelism as in the manual design. Several bottlenecks were encountered such as the divider cost both in area for

Synphony and AutoESL and in clock period for the Synphony implementation and the capability of pipelining for improved throughput in AutoESL when dealing with imperfect nested loops.

A new solution was presented to increase performance by tackling these bottlenecks. Using only minor code modifications, a different implementation resulted in less resource utilization with even a minor latency gain. Next to this, several streaming implementations based on the optimized AutoESL and Synphony implementations have been created, exploiting both data parallelism inside kernels as well as parallelism across kernels by allowing the overlapping of kernel execution. The performance results for all implementations, relative to the reference manual implementation, are shown in figure 5.21.

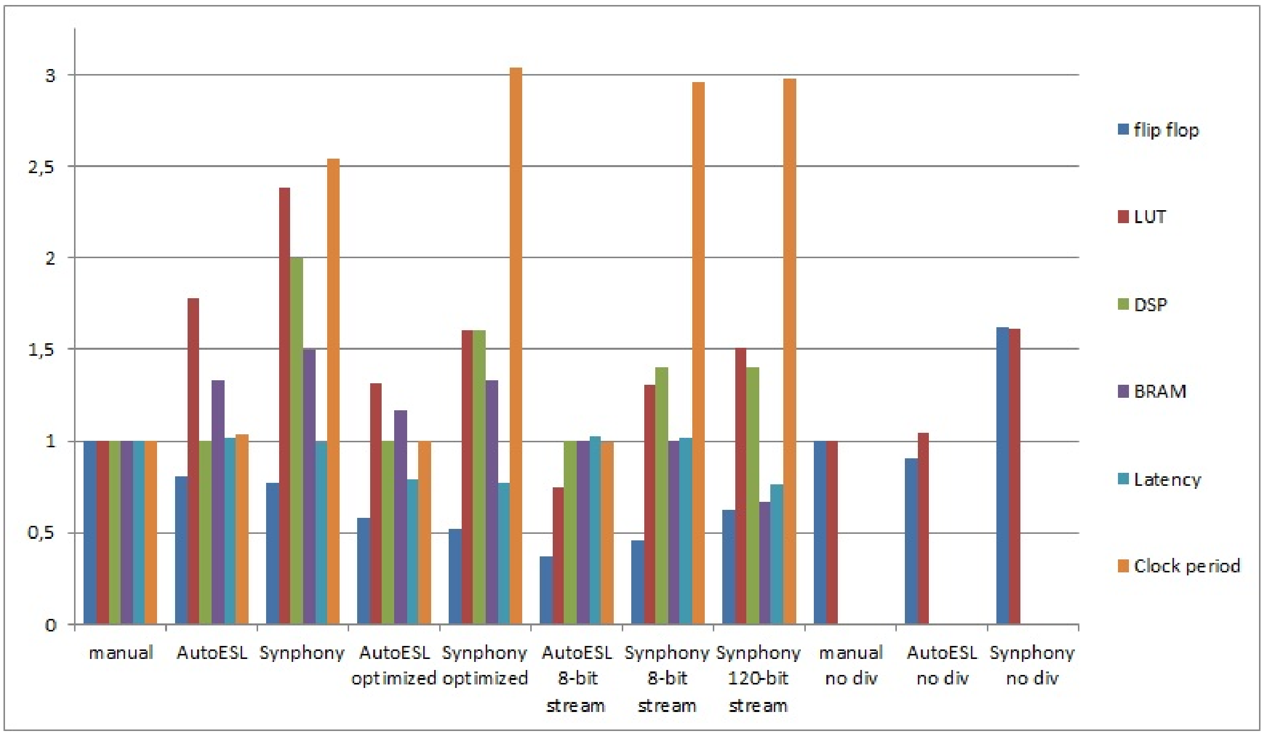


Figure 5.21: Different FFoS implementations and their performance metrics

The key observations which can be made from this graph are:

- The *no div* implementations only compare flip flop and LUT utilization. There are no changes in DSP, BRAM and clock period results compared to the implementation mimicking the manual reference. Flip flop and LUT utilization in the AutoESL design stays within 5% of the reference.
- The clock period in Synphony is extremely high compared to the reference. This is caused by the divider implementation, and this performance might increase depending on the feedback of Synopsys.
- The *optimized* implementations are faster due to two times speedup in the binarization kernel.
- The AutoESL *8-bit streaming* implementation achieves approximately the same latency with less resources due to its streaming implementation of the binarization, erosion and projection pipeline. This implementation contains more task parallelism and less data parallelism.

- DSP usage in Synphony is due to the lack of optimized bit-widths which can be resolved.

Based on the kernel comparison and the application comparison, the following conclusions can be made:

- AutoESL better suits the vision application domain which are often simple hardware pipelines with vision operations executing after each other. Synphony's architecture template supports multiple application domains by improving throughput with the introduction of stall domains. The resource utilization for Synphony is therefore, in each experiment conducted, higher compared to AutoESL while there is little gain in latency.
- Synphony does support design hierarchy and IP re-use, although it requires more effort compared to AutoESL in which a simple pragma can be used to re-use certain IP blocks. Without effort, Synphony will duplicate all hardware related to a function which is called multiple times from the top-level. Note however that Synphony is capable of instantiating TCAB IP blocks in different designs, whereas this is not possible in AutoESL.
- Both tools are capable of accurately mimicking datapaths which are usual in handwritten RTL designs.
- There are still issues to resolve, including costly functional units such as dividers (both tools).
- **To achieve good QoR, source code transformations based on application and platform knowledge are required**

Just as with RTL, there are many optimization possibilities and different implementations with different trade-offs. Depending on the design constraints, the designer can use a set of optimizations and coding styles to achieve the aimed-for datapath. Without clear goal, and without knowledge of the underlying hardware, a designer could end up in an endless optimization effort. With a clear goal and knowledge of underlying hardware, the designer can quickly modify sequential C code to generate predictable and efficient hardware using HLS. The tools explored support a large set of C/C++, allowing the use of specifications usually used to verify correctness of an algorithm (with minor changes for HLS tool compatibility). The case-study has shown that, at least in the case of AutoESL, performance gain is possible and the design time is significantly shorter. In the rest of this report, AutoESL will be used as HLS tool of choice for the following reasons:

- AutoESL appears to better suit the vision application domain. Note that this conclusion is only based on the FFoS application in combination with results from related work. It could still be that Synphony outperforms AutoESL for other vision application, although this is not yet shown.
- Synphony so far has no floating point support and appears to run in beta mode when dealing with large vectorized data.
- Even although the divider implementation is a bottleneck in area consumption, it has been shown that simple modifications at C/C++ level and the use of HLS tool architectural settings can optimize the design.
- Better code readability enables manual changes to the generated RTL itself, and a better analysis.

Chapter 6

Design Space Exploration

One of the benefits of HLS is the possibility to perform fast design space exploration (DSE), which is more time consuming at RTL level as it often requires a complete rewrite of the specification. High level synthesis supports source to source transformations at a higher abstraction level and often uses synthesis constraints to generate datapaths with specific properties. This chapter presents a quick overview of the DSE capabilities of AutoESL, based on the histogram and erosion kernels. Sections 6.1 and 6.2 show DSE results on the histogram and erosion kernel respectively. The goal is not to search through the entire design space, as it is very large, but to explore a range of possible datapaths which are common in RTL design to observe the effectiveness and limitations of DSE using HLS. Each DSE starts with a naive C description as it would be usually coded when targeting a CPU, followed by optimizations with regard to the interface bandwidth and degree of instruction parallelism (optimizations to improve pipelineability) and data parallelism (loop unrolling). Finally, section 6.3 gives a short summary of the results.

6.1 Histogram Design Space Exploration

We treat the histogram design as a single IP block, calculating the histogram bins and notifying the next IP with a done signal when the computation is done. Initialization of the block RAM is ignored by declaring the histogram array as a static variable meaning it well be implemented as set-to-zero on reset. Several designs are created, with as parameters the interface for the input pixels (external block RAM or FIFO), and the unroll factor to determine how many pixels to process at the same time. The unrolling itself has two different implementation options, either packing the input pixels into wider vectors exploiting the 32-bit FSL interface and block RAM widths, or sequentially loading new 8-bit input pixels from the interface. When using an FSL interface, the data-width of each input can be extended to 32 bits supporting up to 4 pixels. If a block RAM is used for the input pixels it is assumed to be external and therefore not included in the results. The advantage of a block RAM is that we can improve bandwidth beyond loading 4 pixels per cycle, by either using wider block RAMs (the target device supports up to 9 pixels per cycle per read port without extra block RAM cost) or distributing the pixels over multiple block RAMs or distributed RAM. Furthermore, block RAMs can be dual ported, potentially improving unroll performance compared to using a FIFO which can only support one read per cycle unless multiple links are used.

The DSE starts with two baseline implementations, shown in listing 6.1 and 6.2 and referred to as base-1 and base-2 respectively. The naive base-1 implementation will result in a slow design caused by a RaW hazard on the histogram bins and base-2 solves this issue by comparing the read and write addresses. The following two sections will highlight interesting observations based on an FSL and block RAM interface respectively. The designs are analyzed based on flip flop and LUT utilization, block RAM utilization for internal histogram bin storage is ignored, but it should be noted that no more than 4 block RAMs are used for partitioning of histogram data.

```
for(x = 0; x < WIDTH*HEIGHT; x++){
        bins[ image[x]] ++;
    }
}
```

Listing 6.1: Base-1 description of the histogram kernel

```
for(x = 0; x<WIDTH*HEIGHT+1; x++){
    index = image[x];
    if(old_index == index){
        accu = accu + 1;
    }else{
        bins[old_index] = accu;
        accu = bins[index] + 1;
    }
    old_index = index;
}
```

Listing 6.2: Base-2 description of the histogram kernel

6.1.1 FSL Interface

As histogramming can usually performed at the start of an image processing pipeline, connecting it directly to a MicroBlaze using a Fast Simplex Link is a common approach. The FSL is 32 bits wide and supports up to four 8-bit pixels as communication elements, enabling a parallelization factor of 4 in both the base-1 and base-2 implementation. Figure 6.1 shows the explored design points in latency and flip flop trade-off, figure 6.2 shows latency and LUT trade-offs.

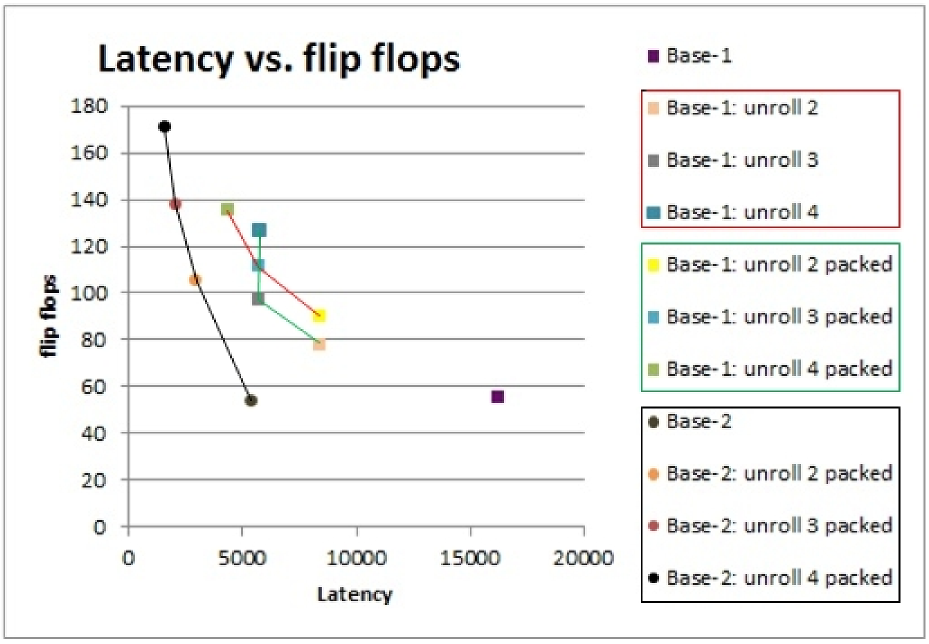


Figure 6.1: Latency versus flip flop utilization for the histogram kernel

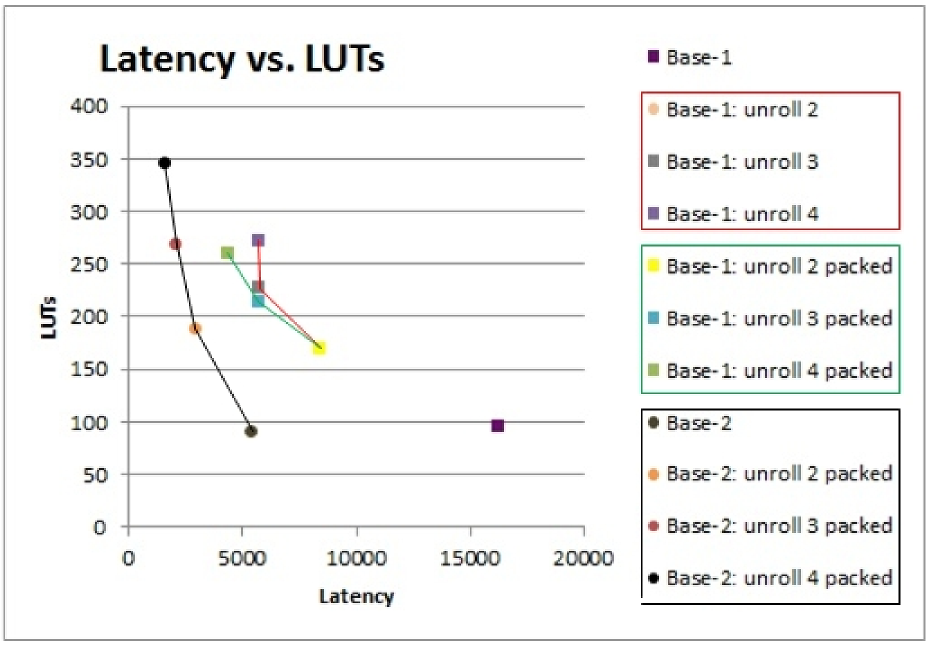


Figure 6.2: Latency versus LUT utilization for the histogram kernel

The names corresponding to the design points relate to the implementation. A manually unrolled implementation is referred to as *unrolled* with the given unroll factor and has an 8-bit FSL interface. Manually unrolled implementations with packed data on the FSL (extending the number of pixels communicated per FSL token) are referred to as *unrolled packed* with the given unroll factor. As a reference, parameterized versions of the input descriptions for these implementations are given in Appendix A. As an example, the base-1 description presents parameterized manual unrolling without data packing whereas the base-2 description shows the approach with data packing. These parameterized descriptions can be used for automatic DSE, however they need manual modifications if the parallelization factor is one (no data parallelism) to prevent histogram bin merging. Data packing has the advantage of requiring less reads from the interface, whereas the absence of data packing requires more reads from the interface. With manual unrolling the RaW latency can be hidden by increasing pipeline depth and overlapping multiple histogram operations at the same time. Colors are used to group design points to certain implementation styles, the lines between design points however do not represent intermediate results but are used for visualization only.

In both figures, 4 design points are Pareto optimal, each corresponding to the same four base-2 implementations. Several interesting observations can be made. The first observation is the large jump in area when either the base-1 or base-2 implementation is unrolled. This jump is caused by the extra loop required to combine the separated histogram data into a single histogram block RAM. Once the unroll factor increases beyond 2, the area increase is less as the hardware to combine histogram data is already there (only a multiplexer and adder are required to add another sub-histogram).

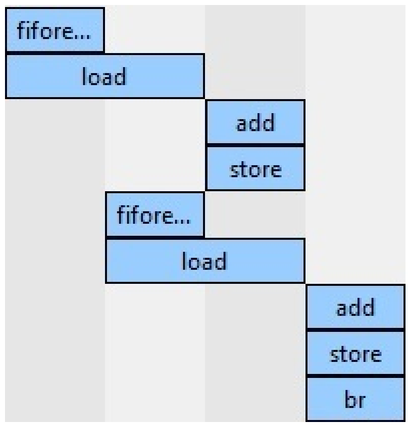


Figure 6.3: Pipeline initiation interval dominated by block RAM read-modify-write

The absence of an unrolled implementation without data-packing in the base-2 implementation can be explained by the maximum bandwidth available. The base-2 implementation already processes 1 pixel per cycle, and reading more pixels per cycle from a FIFO is not possible without duplicating the number of FSLs. In the base-1 implementation, there are 3 pipeline stages and the initiation interval is equal to the pipeline depth, which is the read-modify-write operation on the histogram block RAM requiring 3 cycles. This enables latency improvement when unrolling the loop without the need to pack data on the interface, up to unroll factor 3 the latency bottleneck is the 3 cycles required for the histogram bin read-modify-write. By unrolling, AutoESL increases the pipeline depth increasing the pipeline delay but improving instruction parallelism. When unrolled three times, the bottleneck is both the RaW hazard as well as the input bandwidth. If the unroll factor becomes 4, the bottleneck is no longer the read-modify-write operation but latency is limited by the input bandwidth. Example schedules in figures 6.3 through 6.5 highlight this issue, clearly showing the 3-cycle load-add-store operation and the 4-cycle input reads in the implementation with an unroll factor of 4. The 4 input reads cause a worse pipelineability of the histogram loop. Table 6.1 presents this observation in summary. In the table, δ represents the overhead required for the combining of separated histogram bins, a result of the unrolled implementation.

The *computational bandwidth* is defined as $\frac{\textit{pixels processed per iteration}}{\textit{initiation interval of the pipeline}}$.

Whereas the initiation interval in packed implementations remains equal with increasing unroll factor, it increases in the base-1 unrolled implementation without packing. This indicates the limit in input bandwidth. The base-2 implementation has a static initiation interval of one, resulting in a computational bandwidth equal to the unroll factor.

Table 6.1: Summary performance results FFoS implementation

Implementation	**Input bandwidth**	**Computational bandwidth**	**Overall bandwidth**
Base-1	1	1/3	1/3
Base-1 unroll 2	1	2/3	2/3 + δ
Base-1 unroll 3	1	3/3	3/3 + δ
Base-1 unroll 4	1	4/4	4/4 + δ
Base-1 unroll 2 packed	2	2/3	2/3 + δ
Base-1 unroll 3 packed	3	3/3	3/3 + δ
Base-1 unroll 4 packed	4	4/3	4/3 + δ
Base-2	1	1	1
Base-2 unroll 2 packed	2	2	2 + δ
Base-2 unroll 3 packed	3	3	3 + δ
Base-2 unroll 4 packed	4	4	4 + δ

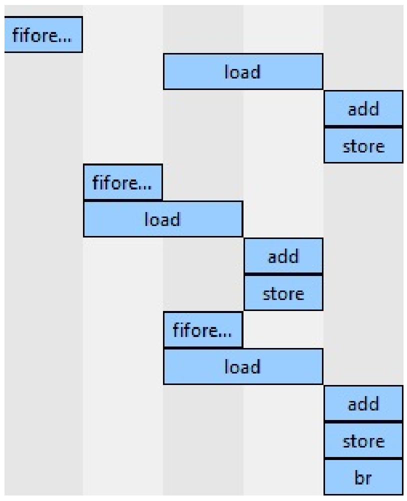


Figure 6.4: Pipeline initiation interval turning point

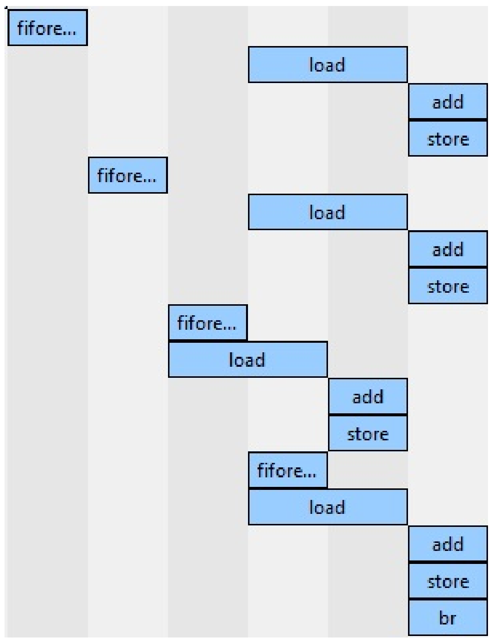


Figure 6.5: Pipeline initiation interval dominated by FIFO access

Although the base-1 implementations are not optimal, design constraints might require the use of one of these implementations. It is then important to note the flipflop-LUT trade-off occurring between packed and non-packed implementations. Packed implementations require more flip flops compared to non-packed implementations with the same unroll factor. When data is packed on the FSL, registers are used to store the unpacked data, resulting in more flip flop utilization compared to unrolled implementations with an 8-bit FSL input. As a result, in packed unrolled implementations, less LUTs are required to route input pixels to the correct histogram block RAM address, whereas in unrolled implementation without packing more LUTs are required to route/multiplex the inputs to the correct block RAM address lines. The increasing pipeline initiation interval occurring once the unroll factor exceeds 3 in non-packed implementations (caused by limited input bandwidth) is no longer an issue in packed implementations as only one input read per cycle is required. Therefore, unrolling 4 times in the packed implementation still results in a latency improvement.

Counter-intuitively, the base-2 reference implementation requires less area compared to the base-1 reference implementation. This is a result of the bad pipeline-ability in the base-1 implementation, resulting in multiple copies of the loop iterator being needed in the datapath. After unrolling, the base-2 implementation requires more flip flops though caused by the need of storage for older indexes to compare read and write addresses. The base-2 implementation is faster and less expensive compared to the base-1 implementation and only requires a change code which an experienced HLS user could implement in minutes. As can be expected, the latency gain becomes increasingly smaller with increased unroll factor. The base-2 implementations are the Pareto points of the explored designs, but is limited by an unroll factor of 4 because of the 32-bit width of the FSL. Note that each unrolled design includes 256 overhead cycles for the combining of the manually unrolled histogram bins which are in multiple block RAMs after manually unrolling the process loop. In the next section, a block RAM is used as interface for the input pixels and will move the latency bottleneck from the width of the interface to the overhead of 256 cycles caused by unrolling.

6.1.2 BRAM Interface

When the image is already stored in external block RAM, the unroll factor is no longer limited to 4 as is the case when using a single FSL. As a side-effect, the manually unrolling of the base-2 implementation by a factor of 2 is now possible (with latency performance gain) by using a dual ported block RAM to increase input bandwidth and prevent the pipeline from stalling. Figure 6.6 shows the explored design points in latency and flip flop trade-off, figure 6.7 shows latency and LUT trade-offs.

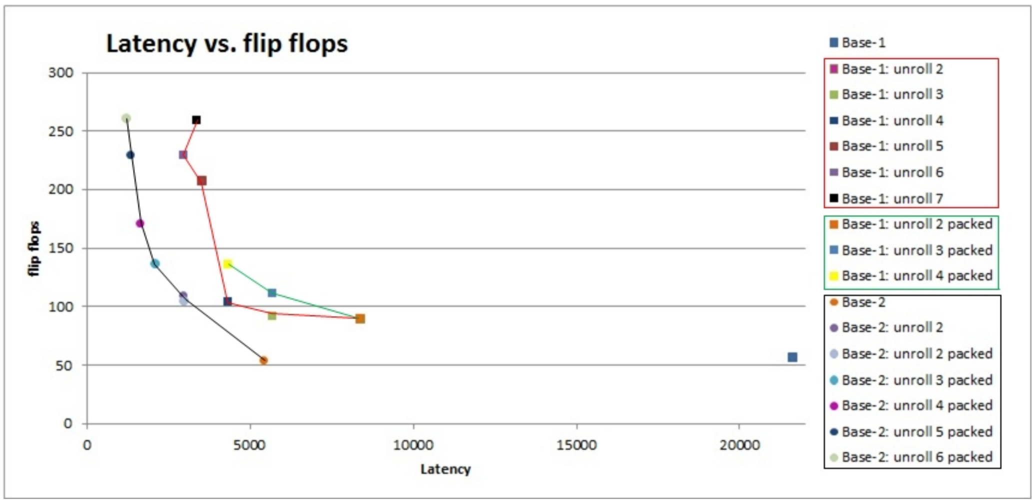


Figure 6.6: Latency vs. flip flop utilization for the histogram kernel

The Pareto optimal points are again resulting from the base-2 implementation. There are more Pareto points with this base implementation as it is now possible to continue the loop unrolling beyond the factor 4 limit when using an FSL. Note however that with increased data word size in block RAMs, the data will be eventually be distributed across multiple block RAMs to facilitate the required throughput. The base-1 implementation requires an unroll factor of 3 to get close to the latency of the base-2 reference implementation, resulting in an area overhead compared to the base-2 reference. For more parallel base-1 implementations even more area would be required,

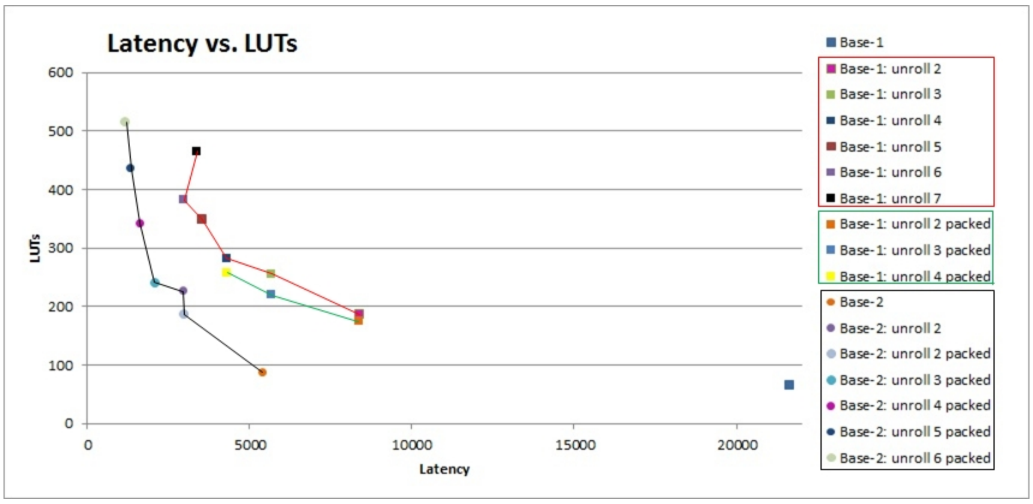


Figure 6.7: Latency vs. LUT utilization for the histogram kernel

making base-2 the optimal solution, allowing the user to constraint the unroll factor according to the design constraints.

Again it can be observed that in the base-1 implementations, unrolling without packing requires more LUTs and less flip flops compared to unrolling with packing. As mentioned, the unroll factor can be increased beyond 4. The base-2 design points clearly show that resource utilization keeps increasing significantly compared to latency decrease when scaling up the unroll factor, and the latency will gradually go to the 256 cycle limit needed for the combination of the partitioned histogram block RAM.

A similar design issue occurs in the base-1 unrolled implementations. As with the FSL link the bottleneck occurred at an unroll factor of 3, caused by the limited input bandwidth, the bottleneck now occurs at an unroll factor of 6 resulting from an increased bandwidth by the use of a *dual* ported block RAM. Once the parallelism goes beyond factor 6, the latency bottleneck moves from the 3-cycle read-modify-write operations to the cycles required to supply the pipeline with data.

6.2 Erosion Design Space Exploration

The erosion operation is assumed to require a 5x5 window to achieve the correct results. This window size can be achieved by either implementing the 5x5 window operation directly, or by splitting up the window operation into smaller sized window operations, executing a smaller erosion kernel multiple times. The DSE is performed on both a 5x5 implementation as a 3x3 implementation executed twice, exploring a few basic optimizations such as pipelining, streaming and unrolling. The input data is assumed to be in external block RAM as the initial input description requires pixels to be fetched from the input multiple times due to the shifting window.

Listing 6.3 presents the naive reference erosion description based on a 3x3 window. The 5x5 description simply extends the 3x3 rectangle to a 5x5 rectangle and increases the border size by one to prevent out-of-memory accesses.

From the reference design, both 3x3 and 5x5 can be expected to produce bad QoR caused by a large amount of reads on the same input array which is implemented in block RAM. The optimizations applied involve source modifications to let the erosion kernel fetch and produce one pixel per cycle, enable interleaved execution of two 3x3 kernels consuming and producing one pixel per cycle, apply unrolling to improve throughput by exploiting dual ported block RAM and implement fine-grained data parallelism to process image lines instead of image pixels. For reference, example input descriptions for a pixel-streaming and a data-parallel driven design can be found in Appendix B.

```
for(y = 0; y<HEIGHT; y++) {
   for(x=0; x<WIDTH; x++) {
      if( (i < 1) || (i>=HEIGHT-1) )
         output[i*WIDTH + j] = 0;
      else if ( (j < 1) || (j>=WIDTH-1) )
         output[i*WIDTH + j] = 0;
      // 3x3 rectangle
      else
         output[i*WIDTH + j] = ( img[(y-1)*WIDTH+x-1] &
                                 img[(y-1)*WIDTH+x  ] &
                                 img[(y-1)*WIDTH+x+1] &
                                 img[(y  )*WIDTH+x-1] &
                                 img[(y  )*WIDTH+x] &
                                 img[(y  )*WIDTH+x+1] &
                                 img[(y+1)*WIDTH+x-1] &
                                 img[(y+1)*WIDTH+x] &
                                 img[(y+1)*WIDTH+x+1] ) ? 1 : 0;
   }
}
```

Listing 6.3: Naive description of the erosion kernel using a 3x3 window

Figure 6.8 shows the explored design points in latency and flip flop trade-off, figure 6.9 shows latency and LUT trade-offs. For the 3x3 window based implementation, the erosion hardware block is only duplicated when applying streaming to speed up the dual execution of the kernel. There is a clear correspondence between the LUT and flipflop design points, except for the two parallel implementations. In the parallel implementation, the 5x5 implementation uses significantly more flip flops which are used for line buffering. The LUT utilization in the 3x3 implementation is slightly less because of the reduced window size, requiring less *and* gates. Note however that the latency for the 5x5 implementation is twice as small as for the 3x3 design, since the smaller implementation needs to execute twice to achieve the same result. The hardware block duplication in the 3x3 streaming implementation results in a less optimal design point compared to the 5x5 throughput optimized design which read and writes one pixel per cycle. Note that more implementations are possible, for example by applying packing to create multiple granularities of parallelism. Packing is not implemented due to time constraints, but better performance is achievable by applying packing in the *3x3 throughput optimized unroll 2* implementation.

6.3 Discussion

The design space exploration on both kernels took only one day to perform, including the gathering of the results and coding the input descriptions. The parameterized input description for the histogram kernel has shown it is possible to automatically perform DSE on specific kernels by the use of scripts. Even if HLS is not accepted in the design flow, it could still be used to do fast proof of concept or just to produce an initial design which can serve as reference for an RTL designer. The DSE on the histogram and erosion kernel has shown it is both fast and easy to produce different trade-offs in the design process.

This DSE has shown that AutoESL can produce a wide range of possible datapaths, including highly parallel and streaming design which are common in FPGA design. The results for the two kernels show that resource utilization and latency can be easily related to the input coding style and knowledge about hardware specifics such as block RAMs and FSLs.

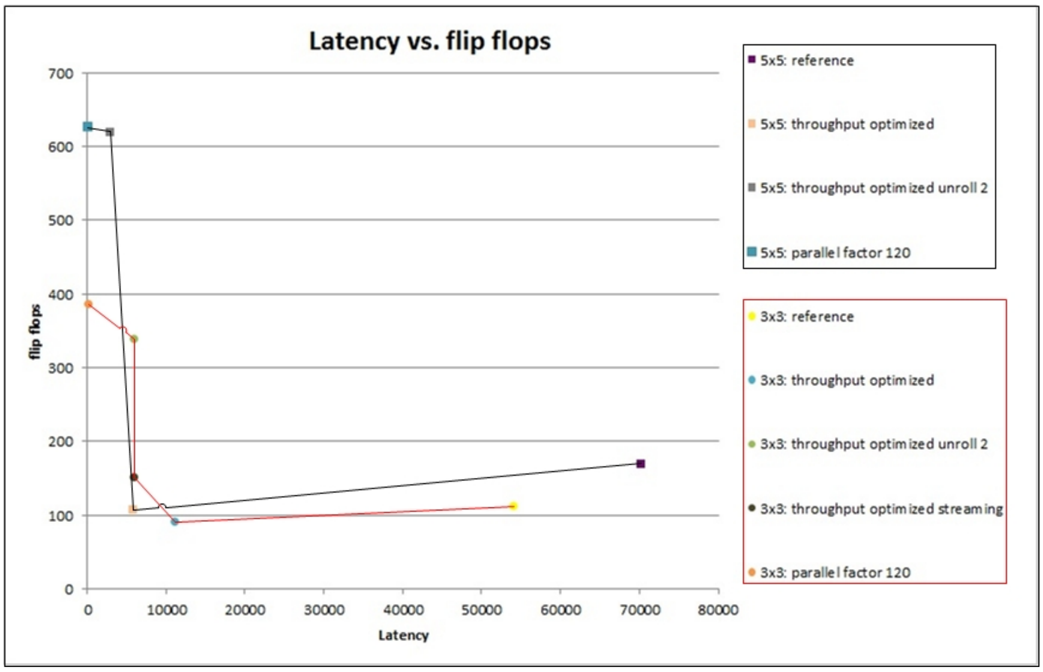


Figure 6.8: Latency vs. flip flop utilization for the erosion kernel

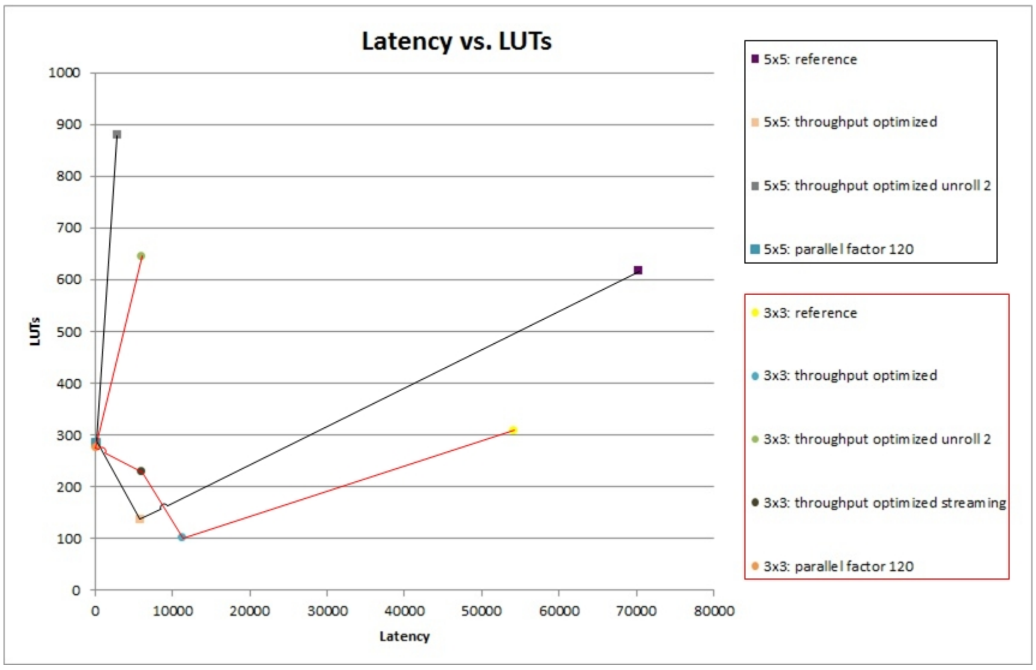


Figure 6.9: Latency vs. LUT utilization for the erosion kernel

Chapter 7

Algorithmic Skeletons for High Level Synthesis

During the design space exploration of the histogram and erosion kernel in chapter 6, several examples of optimized C-code were presented. These code optimizations resulted in higher quality of results when performing high level synthesis, compared to naive C descriptions. For the histogram kernel, it is also shown that the optimized C-code could be easily parameterized to support different degrees of parallelism without requiring any code modifications. The optimized C-code is written in a way to optimize the memory accesses and is independent of the operation which needs to be performed on the data.

This chapter will use and elaborate on these observations and will use the algorithm (FFoS) from the case study in chapter 5 as a leading example towards the introduction of algorithmic skeletons. Goal is to search for parameterizable code structures, the algorithmic skeletons, which can automatically be instantiated for good quality of results in HLS. The question which arises is whether or not these code structures exist and if they can be linked to the algorithm classification as can be found in [45]. This algorithm classification allows fine-grained classification of algorithms while using a limited vocabulary, enabled by the use of parameters and modularity. The focus on the classification is for automated use in compilers and tools, making it a suitable candidate to be included in the high level synthesis design trajectory.

Section 7.1 shortly introduces the existing Bones source-to-source compiler, written by Cedric Nugteren at the Embedded Systems Group. This tool will be used to investigate the usability and applicability of algorithmic skeletons for HLS. Section 7.2 introduces the algorithm classification, on which Bones is based, with regard to the FFoS application. In section 7.3 certain code structures will be related to the algorithm classification for the FFoS example. In section 7.4 several example skeletons for the FFoS algorithm are implemented using the Bones source-to-source compiler targeting AutoESL high level synthesis. Finally, section 7.5 includes a discussion on the findings presented in this chapter.

7.1 Source-to-Source Compiler

The Bones source-to-source compiler will be used to transform highly sequential C code to instantiations of parameterizable skeletons for higher HLS performance and faster design space exploration. Bones is an open-source tool developed by Cedric Nugteren and is written in the Ruby language. The compiler is based on CAST, which parses the input C-code into an abstract syntax tree (AST). Transformations are applied to the AST and is eventually transformed back to C-code by CAST. The tool is open source and supports and encourages users to modify or extend the skeleton library or target library. This work is a first attempt to extend the tool with an FPGA as target platform. Input to the tool is C-code annotated with class information, the output is parallelized target code. Figure 7.1[1] presents a high level overview of Bones. The original code and skeleton code are transformed to an abstract syntax tree. Based on a user supplied transformation list, containing basic transformations such as variable renaming, the skeletons are instantiated. Class specific skeletons can be added by using a number of parameters which are instantiated by Bones, enabling the user to code a specific structure at skeleton level while Bones instantiates the loop body (possibly with basic transformations such as variable renaming).

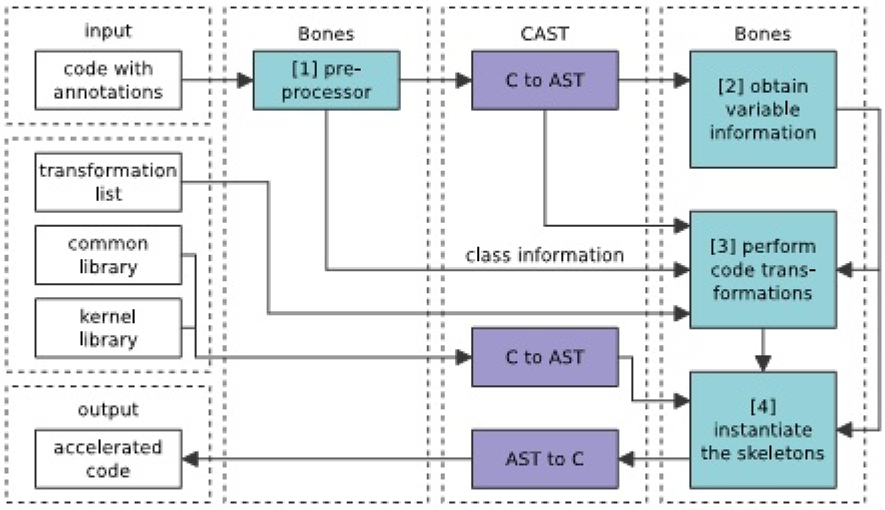


Figure 7.1: High-level overview of the Bones source-to-source compiler

Current targets for code generation include NVIDIA GPUs, AMD GPUs and x86 CPUs using skeletons based CUDA, OpenCL and OpenMP respectively. Bones relies on a new algorithmic classification [45] and generates code based on a skeleton library containing parameterizable skeletons based on the classification. Using a transformation list in combination with the parameterizable skeletons, classified kernels in the input code are transformed and instantiated, replacing the original input description with accelerated versions of the classified kernels.

Figure 7.2 shows the final design flow for the use of algorithmic skeletons in combination with high level synthesis. Bones instantiates skeletons based on manually specified algorithm classes in the algorithm description. Based on parameters in the algorithm class and parameterized skeletons an accelerated algorithm description is generated which results in predictable and understandable high level synthesis results which can be passed down to logic synthesis tools. This design flow can be further extended by automatically analyzing the algorithm specification and inserting the algorithm classes, with the possibility of automatic optimization steps such as kernel fusion.

[1]parse.ele.tue.nl/tools/bones/bones0.9manual.pdf

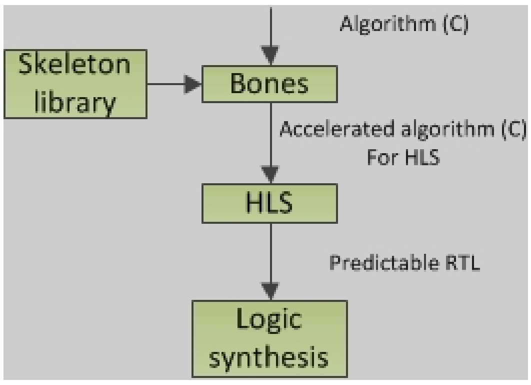


Figure 7.2: Complete design flow using algorithmic skeletons and HLS

7.2 Algorithm Classification

To introduce the algorithm classification used as reference throughout this work, figure **??** shows the same vision pipeline used for the HLS case study in chapter 5. The work of Cedric Nugteren captures the memory access behavior in the classes. To explain this sort of classification and give an intuitive feel of the classification without requiring further reading, we give four small examples based on the FFoS application. The used vocabulary is in research and may not be up to date, the examples in this case serve as an educational example to highlight the main idea behind the classification.

Table 7.1 summarizes the classification for the four kernels discussed in the following sections.

Table 7.1: Algorithm classification for the Fast Focus on Structures application

Kernel	**Classification**
Histogram	Element to shared
Binarization	Element to element
Erosion	Neighborhood to element
Projection	Tile to element

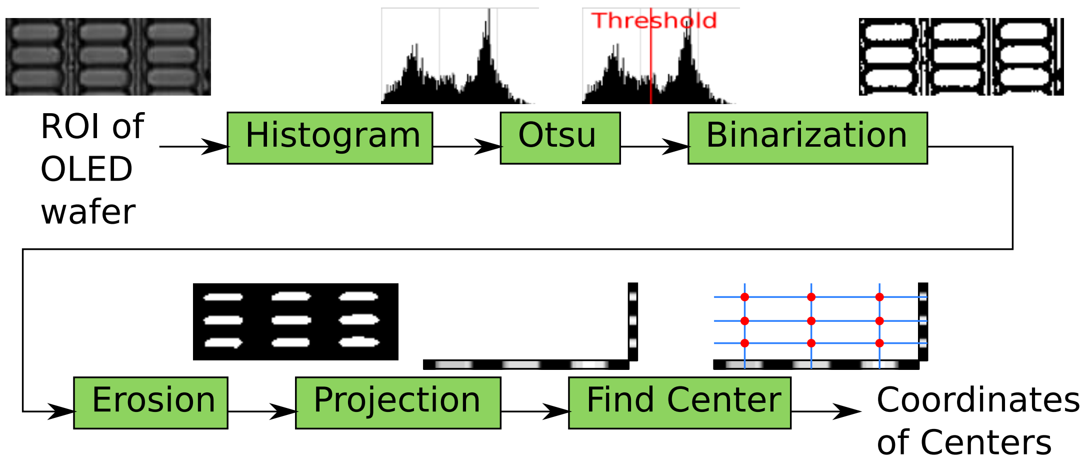


Figure 7.3: The vision pipeline for the fast focus on structures application

7.2.1 Histogram

This kernel reads image data and depending on the pixel values increments a certain histogram bin. Listing 7.1 shows a code example implementing the histogram kernel behavior. The classification in this case implies that the input is two-dimensional of size *HEIGHT * WIDTH* and is processed per element, meaning that each iteration a *different* element is required as input to produce an output. The output in this case is an array containing 256 element, but it is being written in a random order and is therefore of the type *shared*.

The term *shared* in this case implies that multiple iterations of the loop can write to the same location of the output array and thus the output location is shared across the iterations. Note that this classification already enables reasoning about parallelism. The term *shared output* already indicates that parallelizing the computation is not a simple matter of distributing the computations across multiple processing units as multiple of those units might be writing to the same memory location.

```
HEIGHT x WIDTH|element -> 256|shared
for(y = 0; y < HEIGHT; y++){
    for(x = 0;x < WIDTH; x++){
        bins[ image[y][x] ]++;
    }
}
```

Listing 7.1: Histogram behavior and classification

7.2.2 Binarization

This kernel reads image data and depending on the pixel value and the threshold value the output is either one or zero, reducing the required storage for the image. Listing 7.2 shows a code example implementing the binarization behavior. The classification describes the memory access behavior, in this case each input element corresponds to one output element, classifying the kernel as an *element to element* algorithm. In this case the operation, a data comparison, is straightforward but since no notion of operation is captured by the classification this operation can be anything as long as each input element eventually corresponds to one output element in the loop body.

Since each iteration requires a single input element, which is not re-used in other iterations, and writes a single output element which is not accessed in other iterations the computation can be easily parallelized and the degree of parallelism will be limited by the available bandwidth on the target platform. In this case, each output element is only one bit wide, whereas the input bit-width is dependent on the number of bits per pixel and will be the limiting factor with regard to bandwidth.

```
HEIGHT x WIDTH|element -> HEIGHT x WIDTH|element
for(y = 0; y < HEIGHT; y++){
    for(x = 0;x < WIDTH; x++){
        out[y][x] = (in[y][x] > threshold) ? 1 : 0;
    }
}
```

Listing 7.2: binarization behavior and classification

7.2.3 Erosion

This kernel removes noise from an image, based on a particular window size which slides over the input image. Listing 7.3 shows a code example implementing the erosion behavior. The classification describes a *neighborhood* algorithm, requiring a 3x3 window (9 elements) from a 2-dimensional input of size *HEIGHT * WIDTH*. The classification indicates that for each 3x3 input window, a single output element is written. Again, there is no notion of the operation in the classification. It is only identified as a kernel which requires 3x3 input elements, which overlap across loop iterations as indicated by the keyword *neighborhood*. Each output element is based on 9 input elements from the neighborhood window and each output index ia only accessed once across all iterations.

Parallelization of this kernel is more complicated as the sliding window has overlapping elements across loop iterations, meaning multiple iterations requires the same input elements to write an output element. This classification indicates that the input will be the performance bottleneck, and care has to be taken to efficiently manage the loading of the input elements.

```
HEIGHT x WIDTH|neighborhood (3x3) -> HEIGHT x WIDTH|element
for(y=0;y<height;y++){
    for(x=0;x<width;x++){
        if( y<1 || y>= height-1 || x<1 || x>=width-1){
            condition = 0;
        }
        else{
            condition = 1;
            for(a=-1;a<=1;a++){
                condition = condition & in[y-1][x+a]
                                      & in[y  ][x+a]
                                      & in[y+1][x+a];
            }
        }
        out[y][x] = condition;
    }
}
```

Listing 7.3: Erosion behavior and classification

7.2.4 Projection

The projection kernel maps the two dimensional image to a vector. In this case, as shown in listing 7.4, horizontal projection is performed by horizontally adding the input element and storing them in a vector. The classification includes the keyword *tile*, indicating that for each output element a tile of 120 elements is required. The *tile* classification differs from the *neighborhood* classification in the sense that *tile* operations do not have overlapping elements across loop iterations. This means that each output element is based on 120 elements (the tile) which are not needed anymore after the production of that particular output element. In total, *HEIGHT* elements are produced, requiring the same amount of tiles to be fetched from memory.

The classification again describes the memory access pattern for this kernel. If for example the inner loop needs to be fully parallelized, the classification informs the designer that it is necessary to read 120 elements from memory in parallel.

```
HEIGHT x WIDTH|tile (1x120) -> HEIGHT|element
for(y=0;y<height;y++){
    acc = 0;
    for(x=0;x<width;x++){
        acc += in[y][x];
    }
    out[i] = acc;
}
```

Listing 7.4: Projection behavior and classification

7.2.5 Discussion

The previous sections have shown some simple examples of algorithms and the way they are classified. It should be noted that these classifications might be obvious in the examples given, but can be hard to extract manually when the algorithm is more complex. The projection kernel for example can be a merged algorithm which calculates both the horizontal and the vertical projection in the same loops, resulting in a mixture of classes. These small examples however already show the usefulness of the classification, as the designer can extract the main latency bottleneck when targeting the code to high level synthesis from the classification alone without requiring knowledge about the operation(s) in the loop body. In fact, the kernel comparison in chapter 4 and the case study in chapter 5 have shown that the most time consuming code transformations are related to the memory access patterns and memory hierarchy. Memory optimizations in HLS input specifications are the most common way of optimizing the quality of results. This indicates that it is useful to have a classification based on the memory access patterns, but does not yet confirm that we do not need knowledge about the actual operation to be performed. The following section will discuss the code transformations applied to several kernel in the FFoS algorithm and will relate them to the algorithm classification as was presented in this section.

7.3 Code Structures in FFoS

Throughout this work, several code optimizations for the kernels of interest have been shown. This section will summarize these optimizations in relation to the algorithm classification. Section 7.4 will discuss the integration of these structures in the Bones source-to-source compiler, highlighting the advantages, usability and limitations of the algorithm classification and Bones tool for high level synthesis.

7.3.1 Element to Shared

The *element to shared* example in FFoS is the histogram kernel, for which several implementations have been discussed in section 6.1. Listing A2 in section A presents the parameterized source code for the optimal design points explored during the design space exploration. The basic code structure involves address comparisons to remove to the randomness in the block RAM access to improve latency. The random array access on the shared vector results in a RaW hazard, causing high level synthesis to produce a badly pipelined datapath, removing the randomness in the memory accesses enabled high level synthesis tools to produce better quality of results.

The code structure is completely based on the memory access pattern and assumes the operation is performed on a single input, i.e. the operation does not require data from any other memory locations.

7.3.2 Element to Element

The *element to element* class is a fairly simple class as there is no data re-use across loop iterations or data dependent memory accesses such as in the *element to shared* class. Listing 7.5 shows a code example of the basic code structure for AutoESL. By supplying an unroll factor it is possible to create different degrees of parallelism. The operation performed in the loop body can be any operation as long as the code adheres to the classification, consuming one input element and producing one output element per loop iteration. Since AutoESL supplies the user with pragmas for automatic memory optimizations, the loop body can be directly inserted in this code structure. Note that compared to the *element to shared* example previously discussed, the loop body for the algorithm specification can remain unchanged, which will be an important issue when the structures are integrated into Bones.

```
#pragma AP array_reshape variable=in factor=UNROLL_FACTOR dim=2
#pragma AP array_reshape variable=out factor=UNROLL_FACTOR dim=2
for(y = 0; y < HEIGHT; y++){
    for(x = 0;x < WIDTH; x++){
    #pragma AP pipeline
    #pragma AP unroll factor= UNROLL_FACTOR
      <loop body>
    }
}
```

Listing 7.5: Code structure for an element to element class algorithm

7.3.3 Neighborhood to Element

The *neighborhood to element* example in FFoS is the erosion kernel, for which several implementations have been discussed in section 6.2. The listings in section B show two example of possible code structures. The first code, listing B1, solves the input memory performance bottleneck by sequentially loading input elements and storing them in internal row and window buffers to prevent re-used elements to be loaded multiple times from the same memory. The interesting notion of this structure is that the sizes of row and window buffers, number of loop iterations and conditions of the conditional statements can all be extracted from the classification. The loop body of the naive description in listing 7.3 is the same as in the code structure in listing B1, except for the renaming of the input memory, which is now an internal local memory. This indicates that the code structure for a neighborhood classification is operation independent.

Listing B2 shows a different structure, one that operates on the image data in parallel. Although the operation is still the *and* operation, it is important to notice that the loop body is now completely different from the naive description. This indicates that some knowledge about the operation needs to be available if the code structure needs to be instantiated automatically, as the original loop body can no longer be copied into the code structure. Note that this is identical in the case of the *element to shared* code structure.

7.3.4 Tile to Element

The *tile to element* class is in a sense similar to the *element to element* class. In high level synthesis, tile to element algorithms can be efficiently pipelined due to the memory access pattern. If parallel behavior is required, care needs to be taken provide enough bandwidth on the input (and possibly output) memory. Listing 7.6 shows example code for a parameterizable horizontal projection kernel. In this example there is a significant difference with the element to element class. Because a tile is needed to produce an output, the loops are no longer perfectly nested loops. This means that the loop body is not just found in the most inner loop, but actually starts after the first loop. Because the pragmas are inserted inside the inner loop, the loop body is modified and we no longer have a code structure in which the functionality of the naive implementation can be inserted without modifications.

```
#pragma AP array_reshape variable=in factor=UNROLL_FACTOR dim=2
for(y=0;y<height;y++){
    acc = 0;
    for(x=0;x<width;x++){
    #pragma AP pipeline
    #pragma AP unroll factor=UNROLL_FACTOR
        acc += in[y][x];
    }
    out[i] = acc;
}
```

Listing 7.6: Projection behavior and classification

7.3.5 Discussion

Previous sections have shown a few examples of code structures for better quality of results in high level synthesis compared to naive input descriptions. It is shown that mainly the *element to shared* and *neighborhood to element* classes require significant code modifications for optimized performance. The *element to element* and *tile to element* classes appeared to be very similar, however the tile to element class required code modifications inside the loop body whereas the modifications in the element to element class required modifications before the occurrence of the loop body. A similar issue was encountered in the element to shared and neighborhood to shared class, where in some cases the loop body in the naive input description needs to be modified to fit in the optimized code structure. This is an important observation with regard to the introduction of algorithmic skeletons for HLS in combination with the Bones tool. The following sections will elaborate on the integration of these skeletons in the Bones tool to target HLS design for FPGA, showing the limitations caused by these observations.

7.4 Algorithmic Skeletons for HLS in Bones

Several skeleton classes have been implemented in the Bones tool to target HLS using AutoESL. Some issues will be addressed in this section regarding the use of Bones for skeleton instantiation for HLS and the usability and feasability of algorithmic skeletons for HLS.

7.4.1 Element to Shared

The skeleton of the element to shared class is similar to the implementation shown in listing A2 in appendix A, only there are some minor implementation differences. In the skeleton implementation, the choice is made to support loop unrolling to duplicate the loop body and process multiple pixels at the same time. As a result, the user is responsible for the creation of a memory throughput which matches the degree of parallelism required. The parallelization factor is for now only selectable in the skeleton itself, but should eventually be tuned by a parameter inserted with the input specification. Overhead compared to a manually written design occurs when no parallelism is required and user selects an unroll factor of 1. An extra loop with the number of iterations equal to the size of the histogram memory is now required to copy internal histogram data to the output of the skeleton. This can easily be overcome by smarter code generation based on analysis of the parallelization parameter, since the loop is not required to be generated when the parameter is set to 1.

Since the loop body is completely rewritten compared to the naive implementation, the operation for the histogram kernel is hard-coded in the skeleton implementation. Reason for this is the current operation of Bones, which does not perform loop body transformations besides some basic transformations such as variable renaming. This essentially makes the element to shared skeleton a histogram template function for now. To make the skeleton generic, supporting every operation, a way needs to be found to instantiate the operation in the skeleton as well. This can be done by feeding the operation as an argument to the Bones tool, but this will limit the number and complexity of the operations as it is not possible to capture every imaginable operation in a parameter. Furthermore, the replacement of native C data-types by arbitrary precision data-types is required to optimize resource utilization on FPGA as AutoESL is not always able to reduce bit-widths automatically.

Summarized, the observations done mean that for HLS, a source-to-source compiler is required which is able to extract operation information and somehow instantiate this operation in the skeleton of the considered class.

Next to this histogram template skeleton, a skeleton for the *element to one-shared* class has been made. This class operates on N elements to produce a single output element. Examples of such kernels are a calculation of the sum of all pixel values in an image and the search for a maximum or minimum. With the addition of a parallelization parameter all degrees of parallelism can be generated for kernels like the sum-of-pixels. The maximum or minimum kernel can not be fully parallelized with this skeleton as AutoESL is not able to extract the commutative property of the operation and requires significant loop body changes to achieve the same result as loop unrolling does for addition. Listing 7.7 presents a simplified example of the skeleton implementation. Parameters instantiated by bones are surrounded by brackets. The *code1* parameter is replaced with the loop body by Bones.

```
#pragma AP resource variable=<in_1_name> core=RAM_2P
#pragma AP array_partition variable=<in_1_name>
                         cyclic factor=UNROLL_FACTOR dim=1
for(k=0; k<<in_1_dim>; k++){
   #pragma AP PIPELINE II=1
   #pragma AP UNROLL factor=UNROLL_FACTOR
   <code1> //i.e sum = sum + k*in[k]
}
```

Listing 7.7: Element to one-shared skeleton

The reason why maximum does not fit this skeleton description is explained by use of listing 7.8. In this example, the maximum of 8 input elements is searched in a single clock cycle. Writing this kernel as a loop and applying loop unrolling in AutoESL results in a chain of comparisons executed in multiple clock cycles, not gaining the maximum possible performance. Again, such an implementation requires loop body transformations.

```
max1 = (in[0]>in[1]) ? in[0] : in[1];
max2 = (in[2]>in[3]) ? in[2] : in[3];
max3 = (in[4]>in[5]) ? in[4] : in[5];
max4 = (in[6]>in[7]) ? in[6] : in[7];

max5 = (max4 > max3) ? max4 : max3;
max6 = (max2 > max1) ? max2 : max1;

max = (max6 > max5) ? max6 : max4;
```

Listing 7.8: Optimized code for maximum kernel

7.4.2 Element to Element

The element to element class is the simplest class in HLS as it requires almost no changes. Parallelization of such easy kernels is supported by AutoESL by use of pragmas. Again, parameterization of the skeleton classification is required to automatically parallelize with any degree. Currently, the skeleton implementation requires the user to select the proper unroll factor in the skeleton itself. With no parallelism required, an unroll factor of 1 selected, the throughput is improved by automatically selecting the pipelining optimization in AutoESL. Since the loop body of the input description requires no change, all operations are supported.

7.4.3 Neighborhood to Element

For the neighborhood to element class, a skeleton is implemented which optimizes throughput on a pixel-by-pixel basis. Listing B1 in appendix B has been fully parameterized, but again does not optimize for reduced bit-widths and fully parameterizable parallelism. Similar to the histogram template, data type replacement and operation extraction is required to support automatic parallelization in a way such as shown in listing B2 in appendix B.

The skeleton implementation requires the user to handle border cases of the image in the loop body, thus requiring the loops to iterate over the complete image.

7.4.4 Tile to Element

The tile to element skeleton has limited performance due to issues similar to the issues encountered in the histogram implementation and erosion implementation. To support more latency/area trade-off points, loop body modifications are necessary, such as pragma insertion in the inner loop to enable a higher throughput at low area cost and unroll parameterization and partitioning in the inner loop to enable the use of multiple degrees of parallelism. Currently, the skeleton only supports complete parallelization of the inner loop, thus loading the tile in a single cycle for improved latency but at large area cost.

7.5 Discussion

In this chapter it is shown that the algorithm classification is suitable for HLS in the image processing domain. In HLS, most code optimizations are based on the memory access patterns to optimize throughput and the classification has shown particular code optimizations for several classifications such as the *neighborhood to element*, *element to shared*, *element to element* and *tile to element* classes. However, several limitations have been found which will be summarized next.

The programming style for HLS is different from programming GPUs and CPUs. In GPUs and CPUs, the computation is distributed across hardware processing units which are already there waiting to be utilized. This makes index calculations after loop and array flattening acceptable, whereas it would be costly in FPGA design because of the required area utilization for divider and modulo operations. However, by applying loop flattening the skeletons can be made generic to fit any N-dimensional data structure. In this work, skeletons are based on two-dimensional data structures whenever the input or output of the algorithm is based on an image, which should therefore be represented as two-dimensional in the input specification. AutoESL has problems with dealing with flattened data structures, caused by its automatic bit-width optimization. Once an array is flattened and a pointer is used to access the array, AutoESL looses track of the array dimensions and generates incorrect address line bit-widths without the use of explicit bit-casting.

To exploit parallelism or remove memory bandwidth bottlenecks on an FPGA, the loop body often needs to be fully rewritten to exploit arbitrary precision data type usage, in order to keep the resource utilization at an acceptable level. The complete rewrite based on operation and data-types is not included yet in the Bones tool, as so far it supports basic transformations such as variable renaming to, for example, exploit shared memory on the GPU (a transformation also used in neighborhood to element skeleton for HLS).

Although some skeletons have been implemented, the full range of datapaths which could potentially be generated is not exploited yet. To achieve this, an approach more focused towards HLS is needed. To prevent the skeletons from becoming function templates (such as the element to shared skeleton), there is a need for the following:

- Data type renaming to enable fine grained data parallelism.
- A notion of the operation next to the memory access pattern to be able to produce generic skeletons supporting any operation.
- A parameter describing the degree of parallelism required. For example an unroll factor parameter to instantiate multiple copies of the generated data-path. With this, a smarter tool is required to only instantiate useful code based on the parallelization parameter.
- A smarter tool which can apply source transformations on the loop body depending on the classification.

The first and third item can be easily implemented in Bones, but the second and last items pose the largest bottleneck for the introduction of algorithmic skeletons for high level synthesis. AutoESL supports all the necessary arbitrary precision functionality to recode naive C implementations to C descriptions generating efficient datapaths (with any degree of parallelism). However, since no notion of operation exists in the algorithm classification, automatically instantiating skeletons is infeasible so far. The data bit-widths and arbitrary precision functionality required is dependent on the operation and this operation can be anything from a simple *bit-wise and* to a complicated datapath containing multipliers and dividers. Furthermore, experiments have shown unexpected HLS behavior when using flattened data structures (used to keep skeletons generic) and optimizations such as arbitrary precision have an influence on all the occurences of the variable (thus are not limited to the classified kernel).

With a tool able to perform the requires code transformations, table 7.2 presents the lines of code required at the different specification levels, justifying the use of algorithmic skeletons for HLS. In the table, *C* is the lines of code required as input for the tool that instantiates the skeletons, *C'* is the lines of code after code transformations for high quality of results in HLS and *Verilog* is the lines of code for a manual RTL design. The lines of code in *C'* and Verilog are related to similar designs and it is shown that design time can be significantly reduced by an average of a factor 3 (assuming LOC as design time metric) in the case of the four example kernels.

Table 7.2: Total lines of code

Kernel	**C**	**C'**	**Verilog**
Histogram	3	20	120
Binarization	3	7	160
Erosion	9	18	160
Projection	6	10	240

Chapter 8

Conclusion and Future Work

This chapter contains a short overview of this work, summarizing results and observations, and suggestions for future work. The conclusion is presented is section 8.1, suggestions for future work can be found in section 8.2.

8.1 Conclusion

Two small image processing kernels are ported to FPGA using two HLS tools. Both tools were able to mimic the datapath of handwritten RTL designs serving as reference. The latency constraint set by the reference designs was met using both tools. AutoESL appeared to have the best performance for these image processing kernels. The architecture template in Synphony infers components which are often not necessary in the image processing domain, as it is mainly the stall domain logic which causes area overhead in Synphony. AutoESL uses more area for the histogram kernel, which is a result of the absence of fine grained interface control when specifying FSL as the interface in AutoESL. The handwritten design is optimized to use an interface control bit to start or stop kernel execution, whereas in AutoESL HLS kernel execution is controlled by a loop and the accompanying hardware. Design time for such small kernels differ from one to two hours in HLS to one to two days in RTL.

The Fast Focus on Structures application was successfully ported to FPGA using both HLS tools. The absence of floating point in Synphony can, in this case, be overcome by conversion of a floating point kernel to an integer kernel. Note that this can not be done for all image processing kernels, for example due to accuracy constraints, and the lack of floating point in Synphony can be a big issue. Large fluctuations in the cost of dividers generated by CoreGen, AutoESL and Synphony resulted in a difficult comparison. When ignoring divider cost in the best comparable designs, differing only slightly in interface cost, AutoESL uses 5% less flip flops and 4% more LUTs whereas Synphony required 62% more flip flops and 61% more LUTs. Latencies for both HLS designs are within 1% of the reference. It is also shown that small modifications to the input specification, requiring only minutes of work, have large positive effects on the performance results. With just a few small changes, both latency and area cost were reduced. Several designs were explored within a day, where a limitation in AutoESL was encountered regarding wide vectorization in combination with the use of FIFOs. This indicates that the tool is still under development and can not yet generate as fine grained designs as can be achieved in handwritten RTL. When experienced with the tool, design time was reduced by a factor 6 compared to the handwritten RTL design.

Following the performance analysis, two design space explorations on image processing kernels are performed. The results have shown that using parameterizable input descriptions, a large scope of designs can be explored within a day. Resource utilization and execution time could be related to the input specification, and generated schedules by AutoESL visualize performance bottlenecks to supply the designer with the necessary information for design optimization.

During the kernel comparison, application mapping and design space exploration, several code structures were identified which could be related to an existing algorithm classification. An attempt was done to automate the code optimization process using the Bones source-to-source compiler. During this attempt, several issues were encountered. The main issue is the HLS tool specific input specifications which are required by AutoESL and the need for extensive loop body transformations. The Bones tool so far only performs basic loop body transformations such as variable renaming, which have been successfully used in several skeleton implementations for HLS. However, the code structures identified also require more complicated loop body transformations. The skeleton implementation for HLS differs in so many ways from skeletons implemented in Bones for GPU and CPU, that it might be more effective to introduce a separate tool to perform the automated tasks. Furthermore, it is debatable whether or not automatic code transformation is worth the effort as many limitations were encountered. Optimized code instantiation at kernel level results in the need for code transformations at application level, such as data type conversion and changes in memory sizes and layout. Loop and array flattening to support generic skeletons supporting multiple dimensions results in area overhead for address calculation by the use of dividers and modulo operations and bit-width optimizations issues in AutoESL.

To summarize, the results of the high level synthesis analysis look promising regarding the application which was used and the covered scope of generated datapaths. Although several performance issues still exist, HLS seems to have matured enough to be included in the design process. If performance issues are encountered, it is always possible to fall back to handwritten RTL and the generated RTL by HLS can even be used as reference as it is readable and visualized by a schedule viewer. The use of automated code transformation is debatable as extensive loop body transformations are required and have effects on required transformations in the entire application scope. Furthermore, optimized C descriptions heavily depend on tool specific libraries and coding styles imposed by the HLS tool. It is shown however that design time decreases even further when using algorithmic skeletons, and the designer can ignore the underlying hardware to achieve good quality of results.

8.2 Future Work

Next to this work, several other papers exist in which good quality of HLS results is observed (refer to section 2.3). However, since some issues still exist in HLS, such as divider cost and vectorization issues in AutoESL, a different approach could be research to a combination of HLS design and manual design. An application in which a performance bottleneck is observed might be replaced by a design mixture containing both HLS design and handwritten RTL. This is similar to a C program in which a portion of code is accelerated by the use of inlined assembly. The question is how much effort is required to separate HLS and manual design and how to efficiently combine them.

More work needs to be done on the feasibility and usability of automatic source transformations using algorithmic skeletons for HLS. More applications need to be analyzed in order to see if other code structures for high quality of results exist and if they match the algorithm classification. The results so far indicate that tool specific skeleton libraries are required and that extensive loop body transformations are required to achieve good quality of results. A good approach would be to design a separate tool for HLS skeleton instantiation, starting with automatic code transformations based on the code structures explored in this work and gradually extend to tool to support more complicated applications.

A different design trajectory with regard to skeleton instantiation can also be explored, being the hardware skeleton trajectory. Using this approach, skeletons will not be HLS tool dependent but most likely be logic synthesis tool dependent. The question is if higher or similar quality of results can be achieved compared to HLS and handwritten RTL and if design time can be reduced compared to manual code transformations for HLS.

Appendix A

Parameterized Histogram Descriptions for DSE

```
void histogram(ap_uint<8*UNROLL_FACTOR> * image, ap_uint<16> * pi){
    ap_uint<13> j;
    ap_uint<UNROLL_FACTOR+1> a;
    static ap_uint<16> pi_tmp[UNROLL_FACTOR][256];
    #pragma AP array_partition variable=pi_tmp complete dim=1
    #pragma AP resource variable=pi_tmp core=RAM_2P
    static ap_uint<16> tmp;

    for(j = 0; j<(WIDTH*HEIGHT); j+=UNROLL_FACTOR){
    #pragma AP PIPELINE
        for(a=0;a<UNROLL_FACTOR;a++){
        #pragma AP UNROLL
            pi_tmp[a][image[j+a]]++;
        }
    }
    for(j=0;j<256;j++){
    #pragma AP PIPELINE
        for(a=0;a<UNROLL_FACTOR;a++){
        #pragma AP UNROLL
            tmp += pi_tmp[a][j];
        }
    pi[j] = tmp;
    }
}
```

Listing A1: Base-1 description of the histogram kernel

```
void histogram(ap_uint<8*UNROLL_FACTOR> * image, ap_uint<16> * pi){
    ap_uint<13> j;
    unsigned int a;
    static ap_uint<16> pi_tmp[UNROLL_FACTOR][256];
    #pragma AP resource variable=pi_tmp core=RAM_2P
    ap_uint<8*UNROLL_FACTOR> packed;
    #pragma AP array_partition variable=pi_tmp complete dim=1
    ap_uint<8> old_index[UNROLL_FACTOR];
    #pragma AP array_partition variable=old_index complete dim=1
    ap_uint<8> index[UNROLL_FACTOR];
    #pragma AP array_partition variable=index complete dim=1
    ap_uint<16> accu[UNROLL_FACTOR];
    #pragma AP array_partition variable=accu complete dim=1

    for(j=0;j<UNROLL_FACTOR;j++){
    #pragma AP UNROLL
        accu[j] = 0;
        old_index[j] = 255;
    }
    for(j=0;j<(5400/UNROLL_FACTOR)+1;j++){
    #pragma AP PIPELINE
    #pragma AP dependence variable=pi_tmp intra RAW false
        packed = image[j];
        for(a=0;a<UNROLL_FACTOR;a++){
        #pragma AP UNROLL
            index[a] = packed.range(a+(a*7)+7,a*8);
        }
        for(a=0;a<UNROLL_FACTOR;a++){
        #pragma AP UNROLL
            if(old_index[a] == index[a]){
                accu[a] = accu[a] + 1;
            }
            else{
                pi_tmp[a][old_index[a]] = accu[a];
                accu[a] = pi_tmp[a][index[a]] + 1;
            }
                old_index[a] = index[a];
        }
    }
    ap_uint<16> tmp = 0;
    for(j=0;j<256;j++){
    #pragma AP PIPELINE
        for(a=0;a<UNROLL_FACTOR;a++){
        #pragma AP UNROLL
            tmp += pi_tmp[a][j];
        }
        pi[j] = tmp;
    }

}
```
Listing A2: Base-2 description of the histogram kernel

Appendix B

Erosion Descriptions for DSE

.

```
void erosion(ap_uint<1> * img, ap_uint<1> * out){
   unsigned int i, j;
   ap_uint<1> tile_buffer[3][3];
   #pragma AP ARRAY_PARTITION variable=tile_buffer dim=0 complete
   ap_uint<1> row1[120];
   ap_uint<1> row2[120];
   ap_uint<1> row3[120];
   ap_uint<1> temp_data;
   ap_uint<1> output;

   for(i = 0; i < (HEIGHT+1); i++) {
      for(j = 0; j < (WIDTH+1); j++) {
      #pragma AP PIPELINE
         output = 0;
         if(i<HEIGHT && j <WIDTH){
            row1[j] = row2[j];
         row2[j] = row3[j];
            temp_data = data[i*WIDTH+j];
         row3[j] = temp_data;
         tile_buffer[0][0] = tile_buffer[0][1];
         tile_buffer[0][1] = tile_buffer[0][2];
         tile_buffer[1][0] = tile_buffer[1][1];
         tile_buffer[1][1] = tile_buffer[1][2];
         tile_buffer[2][0] = tile_buffer[2][1];
         tile_buffer[2][1] = tile_buffer[2][2];
         tile_buffer[0][2] = row1[j];
         tile_buffer[1][2] = row2[j];
         tile_buffer[2][2] = row3[j];
         }
         if(i>0 && j>0){
            if( (i >= 2 || i < HEIGHT) && (j >= 2 || j<WIDTH) ){
            output= ( tile_buffer[0][0] & tile_buffer[0][1] &
                      tile_buffer[0][2] &  tile_buffer[1][0] &
                      tile_buffer[1][1] & tile_buffer[1][2] &
                      tile_buffer[2][0] & tile_buffer[2][1] &
                      tile_buffer[2][2] ) ? 1 : 0;
         }
```

```
            else{
                output = 0;
            }
                pbuf_erode[(i-1)*WIDTH+j-1] = output;
            }
        }
    }
}
```

Listing B1: Pixel-streaming erosion description based on a 3x3 window

```
void erosion(ap_uint<120> * img, ap_uint<120> * out){
    unsigned int i, j;
    static ap_uint<120> row1;
    static ap_uint<120> row2;
    static ap_uint<120> row3;
    static ap_uint<120> tmp;
    static ap_uint<120> out;

    for(i=0;i<HEIGHT+1;i++){
    #pragma AP PIPELINE
        row1 = row2;
        row2 = row3;
        if(i<HEIGHT){
            row3 = data[i];
        }
        else{
            row3 = 0;
        }
        tmp = row1 & row2 & row3;
        if(i>0){
            for(j=1;j<WIDTH-1;j++){
            #pragma AP UNROLL
                output[i-1].set_bit(j,(tmp[j-1] & tmp[j] & tmp[j+1]));
            }
        }
    }
}
```

Listing B2: Parallel erosion description based on a 3x3 window

Bibliography

[1] John Hennessy and David Patterson. *Computer Architecture - A Quantitative Approach.* Morgan Kaufmann, 2003.

[2] Reiner Hartenstein. Coarse grain reconfigurable architecture (embedded tutorial). In *Proceedings of the 2001 Asia and South Pacific Design Automation Conference*, ASP-DAC '01, pages 564–570, New York, NY, USA, 2001. ACM.

[3] Katherine Compton and Scott Hauck. Reconfigurable computing: a survey of systems and software. *ACM Comput. Surv.*, 34:171–210, June 2002.

[4] João M. P. Cardoso, Pedro C. Diniz, and Markus Weinhardt. Compiling for reconfigurable computing: A survey. *ACM Comput. Surv.*, June 2010.

[5] K. Wakabayashi and T. Okamoto. C-based soc design flow and eda tools: an asic and system vendor perspective. *Computer-Aided Design of Integrated Circuits and Systems, IEEE Transactions on*, 19(12):1507 –1522, dec 2000.

[6] P. Coussy, D.D. Gajski, M. Meredith, and A. Takach. An introduction to high-level synthesis. *Design Test of Computers, IEEE*, 26(4):8 –17, july-aug. 2009.

[7] S. Director, A. Parker, D. Siewiorek, and Jr. Thomas, D. A design methodology and computer aids for digital vlsi systems. *Circuits and Systems, IEEE Transactions on*, 28(7):634 – 645, jul 1981.

[8] John Granacki, David Knapp, and Alice Parker. The adam advanced design automation system: overview, planner and natural language interface. In *Proceedings of the 22nd ACM/IEEE Design Automation Conference*, DAC '85, pages 727–730, Piscataway, NJ, USA, 1985. IEEE Press.

[9] P. G. Paulin, J. P. Knight, and E. F. Girczyc. Hal: A multi-paradigm approach to automatic data path synthesis. In *Papers on Twenty-five years of electronic design automation*, 25 years of DAC, pages 587–594, New York, NY, USA, 1988. ACM.

[10] G. Zimmermann. The mimola design system a computer aided digital processor design method. In *Proceedings of the 16th Design Automation Conference*, DAC '79, pages 53–58, Piscataway, NJ, USA, 1979. IEEE Press.

[11] H. Man, J. Rabaey, P. Six, and L. Claesen. Cathedral-ii: A silicon compiler for digital signal processing. *IEEE Des. Test*, 3(6):13–25, November 1986.

[12] G. Martin and G. Smith. High-level synthesis: Past, present, and future. *Design Test of Computers, IEEE*, 26(4):18 –25, july-aug. 2009.

[13] R. A. Bergamaschi, R. A. O'Connor, L. Stok, M. Z. Moricz, S. Prakash, A. Kuehlmann, and D. S. Rao. High-level synthesis in an industrial environment. *IBM J. Res. Dev.*, 39(1-2):131–148, February 1995.

[14] P.E.R. Lippens, J.L. van Meerbergen, A. van der Werf, W.F.J. Verhaegh, B.T. McSweeney, J.O. Huisken, and O.P. McArdle. Phideo: a silicon compiler for high speed algorithms. In *Design Automation. EDAC., Proceedings of the European Conference on*, pages 436 –441, feb 1991.

[15] David W. Knapp. *Behavioral synthesis: digital system design using the synopsys behavioral compiler*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1996.

[16] A. Hemani, B. Karlsson, M. Fredriksson, K. Nordqvist, and B. Fjellborg. Application of high-level synthesis in an industrial project. In *VLSI Design, 1994., Proceedings of the Seventh International Conference on*, pages 5 –10, jan 1994.

[17] Dan Gajski, Todd Austin, and Steve Svoboda. What input-language is the best choice for high level synthesis (hls)? In *Design Automation Conference (DAC), 2010 47th ACM/IEEE*, pages 857 –858, june 2010.

[18] Stephen A. Edwards. The challenges of synthesizing hardware from c-like languages. *IEEE Des. Test*, 23(5):375–386, September 2006.

[19] Altera. Nios ii c-to-hardware acceleration compiler. Website. `http://www.altera.com/devices/processor/nios2/tools/c2h/ni2-c2h.html`.

[20] E. Martin, O. Sentieys, H. Dubois, and J.L. Philippe. Gaut: An architectural synthesis tool for dedicated signal processors. In *Design Automation Conference, 1993, with EURO-VHDL '93. Proceedings EURO-DAC '93. European*, pages 14 –19, sep 1993.

[21] J. Villarreal, A. Park, W. Najjar, and R. Halstead. Designing modular hardware accelerators in c with roccc 2.0. In *Field-Programmable Custom Computing Machines (FCCM), 2010 18th IEEE Annual International Symposium on*, pages 127 –134, may 2010.

[22] Xilinx. Xilinx autoesl high-level synthesis tool. Website. `http://www.xilinx.com/tools/autoesl.htm`.

[23] Cadence. Cadence c to silicon compiler. Website. `http://www.cadence.com/products/sd/silicon_compiler/pages/default.aspx`.

[24] Forte Design Systems. Forte synthesizer. Website. `http://www.forteds.com/products/cynthesizer.asp`.

[25] Mentor Graphics. Catapult c synthesis. Website. `http://www.mentor.com/esl/catapult/overview`.

[26] Synopsys. Synopsys c compiler. Website. `http://www.synopsys.com/Systems/BlockDesign/HLS/Pages/default.aspx`.

[27] S. Muchnick. *Avanced Compiler Design and Implementation*. Morgan Kaufmann, 1997.

[28] K. Rupnow, Yun Liang, Yinan Li, Dongbo Min, Minh Do, and Deming Chen. High level synthesis of stereo matching: Productivity, performance, and software constraints. In *Field-Programmable Technology (FPT), 2011 International Conference on*, pages 1 –8, dec. 2011.

[29] Betul Buyukkurt, Zhi Guo, and Walid Najjar. Impact of loop unrolling on area, throughput and clock frequency in roccc: C to vhdl compiler for fpgas. In Koen Bertels, Jo?o Cardoso, and Stamatis Vassiliadis, editors, *Reconfigurable Computing: Architectures and Applications*, volume 3985 of *Lecture Notes in Computer Science*, pages 401–412. Springer Berlin / Heidelberg, 2006. 10.1007/11802839_48.

[30] Betul Buyukkurt, John Cortes, Jason Villarreal, and Walid A. Najjar. Impact of high-level transformations within the roccc framework. *ACM Trans. Archit. Code Optim.*, 7:17:1–17:36, December 2010.

[31] Michael Fingeroff. *High Level Synthesis Blue Book*. Xlibris, 2010.

[32] Joonseok Park, P.C. Diniz, and K.R. Shesha Shayee. Performance and area modeling of complete fpga designs in the presence of loop transformations. *Computers, IEEE Transactions on*, nov. 2004.

[33] Krste Asanovic, Ras Bodik, Bryan Christopher Catanzaro, Joseph James Gebis, Parry Husbands, Kurt Keutzer, David A. Patterson, William Lester Plishker, John Shalf, Samuel Webb Williams, and Katherine A. Yelick. The landscape of parallel computing research: A view from berkeley. Technical Report UCB/EECS-2006-183, EECS Department, University of California, Berkeley, Dec 2006.

[34] B. Catanzaro, A. Fox, K. Keutzer, D. Patterson, Bor-Yiing Su, M. Snir, K. Olukotun, P. Hanrahan, and H. Chafi. Ubiquitous parallel computing from berkeley, illinois, and stanford. *Micro, IEEE*, 2010.

[35] K. Benkrid, D. Crookes, and A. Benkrid. Towards a general framework for fpga based image processing using hardware skeletons. *Parallel Comput.*, 28:1141–1154, August 2002.

[36] J. Cong, Hui Huang, and Wei Jiang. Pattern-mining for behavioral synthesis. *Computer-Aided Design of Integrated Circuits and Systems, IEEE Transactions on*, 30(6):939 –944, june 2011.

[37] Colella, phillip. de?ning software requirements for scienti?c computing. slide of 2004 presentation included in david pattersons 2005 talk, 2004.

[38] Laura Carrington, Mustafa M. Tikir, Catherine Olschanowsky, Michael Laurenzano, Joshua Peraza, Allan Snavely, and Stephen Poole. An idiom-finding tool for increasing productivity of accelerators. In *Proceedings of the international conference on Supercomputing*, ICS '11, pages 202–212, New York, NY, USA, 2011. ACM.

[39] C. Olschanowsky, A. Snavely, M.R. Meswani, and L. Carrington. Pir: Pmac's idiom recognizer. In *Parallel Processing Workshops (ICPPW), 2010 39th International Conference on*, pages 189 –196, sept. 2010.

[40] W. Caarls, P.P. Jonker, and H. Corporaal. Algorithmic skeletons for stream programming in embedded heterogeneous parallel image processing applications. In *Parallel and Distributed Processing Symposium, 2006. IPDPS 2006. 20th International*, page 9 pp., april 2006.

[41] Wouter Caarls. *Automated Design of Application-Specific Smart Camera Architectures*. PhD thesis, Delft University of Technology, 2008.

[42] D. K. G Campbell. Towards the classification of algorithmic skeletons. Technical report, Technical Report YCS 276, University of York, YEAR = 1996,.

[43] Dhrubajyoti Goswami, Ajit Singh, and Bruno R. Preiss. From design patterns to parallel architectural skeletons. *Journal of Parallel and Distributed Computing*, 62(4):669 – 695, 2002.

[44] C. Nugteren, H. Corporaal, and B. Mesman. Skeleton-based automatic parallelization of image processing algorithms for gpus. In *Embedded Computer Systems (SAMOS), 2011 International Conference on*, pages 25 –32, july 2011.

[45] C. Nugteren and H. Corporaal. A modular and parameterisable classification of algorithms. Website, 2011. `http://www.es.ele.tue.nl/esreports/esr-2011-02.pdf`.

[46] Cedric Nugteren and Henk Corporaal. Introducing 'bones': a parallelizing source-to-source compiler based on algorithmic skeletons. In *Proceedings of the 5th Annual Workshop on General Purpose Processing with Graphics Processing Units*, GPGPU-5, pages 1–10, New York, NY, USA, 2012. ACM.

[47] T.H. Drayer, IV King, W.E., J.G. Tront, R.W. Conners, and P.A. Araman. Using multiple fpga architectures for real-time processing of low-level machine vision functions. In *Industrial Electronics, Control, and Instrumentation, 1995., Proceedings of the 1995 IEEE IECON 21st International Conference on*, volume 2, pages 1284 –1289 vol.2, nov 1995.

[48] A. Shahbahrami. Fpga implementation of parallel histogram computation. Website. `http://ce-serv.et.tudelft.nl/publicationfiles/1477_509_shahbahrami.pdf`.

[49] E. Jamro, M. Wielgosz, and K. Wiatr. Fpga implementaton of strongly parallel histogram equalization. In *Design and Diagnostics of Electronic Circuits and Systems, 2007. DDECS '07. IEEE*, pages 1 –6, april 2007.

[50] D.G. Bailey and C.T. Johnston. Algorithm transformation for fpga implementation. In *Electronic Design, Test and Application, 2010. DELTA '10. Fifth IEEE International Symposium on*, pages 77 –81, jan. 2010.

[51] D.G. Bailey. Invited paper: Adapting algorithms for hardware implementation. In *Computer Vision and Pattern Recognition Workshops (CVPRW), 2011 IEEE Computer Society Conference on*, pages 177 –184, june 2011.

[52] C.T. Johnston and D.G. Bailey. Fpga implementation of a single pass connected components algorithm. In *Electronic Design, Test and Applications, 2008. DELTA 2008. 4th IEEE International Symposium on*, pages 228 –231, jan. 2008.

[53] D.G. Bailey. Image border management for fpga based filters. In *Electronic Design, Test and Application (DELTA), 2011 Sixth IEEE International Symposium on*, pages 144 –149, jan. 2011.

[54] D.G. Bailey. Efficient implementation of greyscale morphological filters. In *Field-Programmable Technology (FPT), 2010 International Conference on*, pages 421 –424, dec. 2010.

[55] K.T. Gribbon, D.G. Bailey, and C.T. Johnston. Design patterns for image processing algorithm development on fpgas. In *TENCON 2005 2005 IEEE Region 10*, pages 1 –6, nov. 2005.

[56] K. Benkrid and D. Crookes. From application descriptions to hardware in seconds: a logic-based approach to bridging the gap. *Very Large Scale Integration (VLSI) Systems, IEEE Transactions on*, april 2004.

[57] A. Papakonstantinou, K. Gururaj, J.A. Stratton, D. Chen, J. Cong, and W.-M.W. Hwu. Fcuda: Enabling efficient compilation of cuda kernels onto fpgas. In *Application Specific Processors, 2009. SASP '09. IEEE 7th Symposium on*, pages 35 –42, july 2009.

[58] A. Papakonstantinou, Yun Liang, J.A. Stratton, K. Gururaj, Deming Chen, W.-M.W. Hwu, and J. Cong. Multilevel granularity parallelism synthesis on fpgas. In *Field-Programmable Custom Computing Machines (FCCM), 2011 IEEE 19th Annual International Symposium on*, pages 178 –185, may 2011.

[59] S. Ahuja, Wei Zhang, A. Lakshminarayana, and S.K. Shukla. A methodology for power aware high-level synthesis of co-processors from software algorithms. In *VLSI Design, 2010. VLSID '10. 23rd International Conference on*, pages 282 –287, jan. 2010.

[60] Deming Chen, Jason Cong, and Yiping Fan. Low-power high-level synthesis for fpga architectures. In *Proceedings of the 2003 international symposium on Low power electronics and design*, ISLPED '03, pages 134–139, New York, NY, USA, 2003. ACM.

[61] M. Meribout and M. Motomura. Efficient metrics and high-level synthesis for dynamically reconfigurable logic. *Very Large Scale Integration (VLSI) Systems, IEEE Transactions on*, 12(6):603 –621, june 2004.

[62] S.M. Logesh, D.S.H. Ram, and M.C. Bhuvaneswari. Multi-objective optimization of power, area and delay during high-level synthesis of dfg's; a genetic algorithm approach. In *Electronics Computer Technology (ICECT), 2011 3rd International Conference on*, volume 1, pages 108 –112, april 2011.

[63] Jason Cong, Bin Liu, Guojie Luo, and Raghu Prabhakar. Towards layout-friendly high-level synthesis. In *Proceedings of the 2012 ACM international symposium on International Symposium on Physical Design*, ISPD '12, pages 165–172, New York, NY, USA, 2012. ACM.

[64] J. Cong, Bin Liu, S. Neuendorffer, J. Noguera, K. Vissers, and Zhiru Zhang. High-level synthesis for fpgas: From prototyping to deployment. *Computer-Aided Design of Integrated Circuits and Systems, IEEE Transactions on*, 30(4):473 –491, april 2011.

[65] Juanjo Noguera, Stephen Neuendorffer, Sven Van Haastregt, and Kees Vissers. Implementation of sphere decoder for mimo-ofdm on fpgas using high-level synthesis tools. *Analog integrated circuits and signal processing*, 69(2-3):119–129, 2011.

[66] Yinan Li Yun Liang, Kyle Rupnow. High-level synthesis: Productivity, performance, and software constraints. Website. `http://www.hindawi.com/journals/jece/2012/649057/`.

[67] A. Plesco. Program transformations and memory architecture optimizations for high-level synthesis of hardware accelerators. Thesis. `http://tel.archives-ouvertes.fr/docs/00/54/43/49/PDF/alexandru.plesco_these.pdf`.

[68] Berkely Design Technology Inc. An independent evaluation of the autoesl autopilot high-level synthesis tool. Website. `http://www.bdti.com/MyBDTI/pubs/AutoPilot.pdf`.

[69] Yuko Hara, Hiroyuki Tomiyama, Shinya Honda, and Hiroaki Takada. Proposal and quantitative analysis of the chstone benchmark program suite for practical c-based high-level synthesis. *Journal of Information Processing*, 17:242–254, 2009.

[70] Jason Cong and Wolfgang Rosenstiel. The last byte: The hls tipping point. *Design Test of Computers, IEEE*, 26(4):104, july-aug. 2009.

[71] Kathryn S. McKinley, Steve Carr, and Chau-Wen Tseng. Improving data locality with loop transformations. *ACM Trans. Program. Lang. Syst.*, 18(4):424–453, July 1996.

[72] M. Kandemir, J. Ramanujam, and A. Choudhary. Improving cache locality by a combination of loop and data transformations. *Computers, IEEE Transactions on*, 48(2):159 –167, feb 1999.

[73] Cedric Bastoul, Albert Cohen, Sylvain Girbal, Saurabh Sharma, and Olivier Temam. Putting polyhedral loop transformations to work. In Lawrence Rauchwerger, editor, *Languages and Compilers for Parallel Computing*, volume 2958 of *Lecture Notes in Computer Science*, pages 209–225. Springer Berlin / Heidelberg, 2004.

[74] Fabien Quilleré and Sanjay Rajopadhye. Optimizing memory usage in the polyhedral model. *ACM Trans. Program. Lang. Syst.*, 22(5):773–815, September 2000.

[75] Cedric Bastoul. Code generation in the polyhedral model is easier than you think. In *Proceedings of the 13th International Conference on Parallel Architectures and Compilation Techniques*, PACT '04, pages 7–16, Washington, DC, USA, 2004. IEEE Computer Society.

[76] Martin Palkovic, Francky Catthoor, and Henk Corporaal. Trade-offs in loop transformations. *ACM Trans. Des. Autom. Electron. Syst.*, 14(2):22:1–22:30, April 2009.

[77] Preeti Ranjan Panda, Nikil D. Dutt, and Alexandru Nicolau. Exploiting off-chip memory access modes in high-level synthesis. In *Proceedings of the 1997 IEEE/ACM international conference on Computer-aided design*, ICCAD '97, pages 333–340, Washington, DC, USA, 1997. IEEE Computer Society.

[78] Yi Zou Jason Cong, Peng Zhang. Optimizing memory hierarchy allocation with loop transformations for high-level synthesis. In *Design Automation Conference (DAC), 2012*, june 2012.

[79] Adam Morawiec Philippe Coussy. *High-Level Synthesis: From Algorithm to Digital Circuit.* Springer, 2008.

[80] Xilinx. In *AutoESL User Guide UG867 2011.4.2*, January 2012.

[81] A threshold selection method from gray-level histograms. *Systems, Man and Cybernetics, IEEE Transactions on*, 9(1):62 –66, jan. 1979.