UNIVERSITY OF CALIFORNIA, SAN DIEGO

Templates and Patterns: Augmenting High-Level Synthesis for Domain-Specific Computing

A dissertation submitted in partial satisfaction of the
requirements for the degree of Doctor of Philosophy

in

Computer Science

by

Janarbek Matai

Committee in charge:

Ryan Kastner, Chair
Rajesh Gupta
Ali Irturk
Bhaskar D. Rao
Michael B. Taylor

2015

The Dissertation of Janarbek Matai is approved and is acceptable in quality and form for publication on microfilm and electronically:

Chair

University of California, San Diego

2015

DEDICATION

To mom and dad.

EPIGRAPH

I wanted a perfect ending. Now I've learned, the hard way, that some poems don't rhyme, and some stories don't have a clear beginning, middle, and end. Life is about not knowing, having to change, taking the moment and making the best of it, without knowing what's going to happen next.

Gilda Radner

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

ACKNOWLEDGEMENTS

First and foremost I wish to thank my advisor Professor Ryan Kastner. This thesis would not have been possible without his support over these years. I am grateful for the friendly research environment provided by him, and for encouraging me to do research during the initial years of my PhD studies. I am also grateful to Ryan for providing me with the opportunity to get involved in the curriculum development process for the MAS WES program at UCSD where I learned many technical and life lessons.

I would like to thank Dr. Ali Irturk for his assistance during the initial stages of my PhD. His comments have greatly improved the quality of my research. I also would like to thank Dr. Juanjo Noguera and Dr. Stephen Neuendorffer at Xilinx Research Labs for providing an amazing internship experience during the summer of 2011. I owe special thanks to Dr. Stephen Neuendorffer at Xilinx for answering many of my technical questions even after my internship. I also like to thank Dr. Joo-Young Kim for the amazing intern experience at MSR and providing the chance to participate in the cutting-edge research project.

I also would like to thank my committee members. I thank them for allocating their valuable time for my thesis proposal and final defense from their busy schedule.

I am indebted to many of my colleagues for fun and positive environment. I am especially thankful for Jason Oberg, Dustin Richmond, Alric Althoff and Perry Naughton for their assistance during these years for helping me to polish my paper writing process.

This journey would not have been possible without the support of all my family members located half way around the world. I thank my parents and parents-in-law for their love. I am always thankful to my brothers and sisters for supporting me during these years even they were so far away. Last but not least, I am grateful to my wife for tolerating my ignorance and my daughters for being nice kids during these years when ”Daddy” was a student.

Chapter 2 contains materials as they appear in International Conference on Field-Programmable Technology (FPT), 2012. Matai, Janarbek; Meng, Pingfan; Wu, Lingjuan; Weals, Brad; Kastner, Ryan. The chapter also contains additional materials (mainly larger figures) that were cut from the publication due to space constraints. The dissertation author is the primary investigator and author of this work.

Chapter 3 contains materials from a work that was published in International Symposium on Field-Programmable Custom Computing Machines (FCCM), 2011. Matai, Janarbek; Irturk, Ali; Kastner, Ryan. The chapter also contains additional materials that were omitted from the publication due to space constraints. The dissertation author is the primary investigator and author of this work.

Chapter 4 contains contains material as it appears in the 25th IEEE International Conference on Application-specific Systems, Architectures and Processors (ASAP), 2014. Matai, Janarbek; Kim, Joo-Young; Kastner, Ryan. The dissertation author is the primary investigator and author of this work.

Chapter 5 contains materials printed in First International Workshop on FPGAs for Software Programmers (FSP 2014). Matai, Janarbek; Richmond, Dustin; Lee, Dajung; Kastner, Ryan. This chapter also contains significant amount of new materials not included in the publication. The dissertation author is the primary investigator and author of this work.

Chapter 6 contains materials from our current work. This work is to be submitted to relevant conference in March, 2015. Matai, Janarbek; Lee, Dajung; Alric Althoff; Kastner, Ryan. The dissertation author is the primary investigator and author of this work.

Chapter 7 contains materials currently in progress. Matai, Janarbek; Richmond, Dustin;Lee, Dajung; Zac Blair; Kastner, Ryan. The dissertation author is the primary investigator and author of this work.

VITA

2004	Bachelor of Science, Mongolian University of Science and Technology
2007	Master of Science, Korea Advanced Institute of Science and Technology (former ICU Campus)
2009	Electronics and Telecommunications Research Institute
2015	Doctor of Philosophy, University of California, San Diego

PUBLICATIONS

Janarbek Matai, Dustin Richmond, Dajung Lee, Zac Blair, and Ryan Kastner. "Resolve: Computer Generation of High-Performance Sorting Architectures from High-Level Synthesis." to be submitted within 2015

Janarbek Matai, Dajung Lee, Alric Althoff, and Ryan Kastner. "Composable, Parameterizable Templates for High Level Synthesis." to be submitted.

Quentin Gautier, Alexandria Shearer, Janarbek Matai, Dustin Richmond, Pingfan Meng, and Ryan Kastner. "Real-time 3D Reconstruction for FPGAs: A Case Study for Evaluating the Performance, Area, and Programmability Trade-offs of the Altera OpenCL SDK." In International Conference on Field-Programmable Technology (FPT), 2014.

Janarbek Matai, Dustin Richmond, Dajung Lee, and Ryan Kastner. "Enabling FPGAs for the Masses." In Proceedings of the First International Workshop on FPGAs for Software Programmers (FSP), 2014

Dajung Lee, Janarbek Matai, Brad Weals, and Ryan Kastner. "High throughput channel tracking for jtrs wireless channel emulation." In Proceedings of the 24th International Conference on Field Programmable Logic and Applications (FPL), 2014

Janarbek Matai, Joo-Young Kim, and Ryan Kastner. "Energy Efficient Canonical Huffman Encoding." In Proceedings of the The 25th IEEE International Conference on Application-specific Systems, Architectures and Processors (ASAP), 2014

Motoki Kimura, Janarbek Matai, Matthew Jacobsen, and Ryan Kastner. "A low-power Adaboost-based object detection processor using Haar-like features." In Proceedings of the Third International Conference on Consumer Electronics Berlin (ICCE-Berlin), 2013.

Janarbek Matai, Pingfan Meng, Lingjuan Wu, Brad T. Weals, and Ryan Kastner. ”Designing a hardware in the loop wireless digital channel emulator for software defined radio.” In Proceedings of the International Conference on Field-Programmable Technology, 2012.

Janarbek Matai, Jason Oberg, Ali Irturk, Taemin Kim, and Ryan Kastner. ”Trimmed VLIW: Moving application specific processors towards high level synthesis.” In Proceedings of the Electronic System Level Synthesis Conference (ESLsyn), 2012.

Janarbek Matai, Ali Irturk, and Ryan Kastner. ”Design and implementation of an FPGA-based real-time face recognition system.” In Proceedings of the 19th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM), 2011.

ABSTRACT OF THE DISSERTATION

Templates and Patterns: Augmenting High-Level Synthesis for Domain-Specific Computing

by

Janarbek Matai

Doctor of Philosophy in Computer Science

University of California, San Diego, 2015

Professor Ryan Kastner, Chair

Implementing an application on a FPGA remains a difficult, non-intuitive task that often requires hardware design expertise in a hardware description language (HDL). High-level synthesis (HLS) raises the design abstraction from HDL to languages such as C/C++/Scala/Java. Despite this, in order to get a good quality of result (QoR), a designer must carefully craft the HLS code. In other words, HLS designers must implement the application using an abstract language in a manner that generates an efficient micro-architecture; we call this process writing restructured code. This reduces the benefits of implementing the application at a higher level of abstraction and limits the impact of HLS

by requiring explicit knowledge of the underlying hardware architecture. Developers must know how to write code that reflects low level implementation details of the application at hand as it is interpreted by HLS tools. As a result, FPGA design still largely remains job of either hardware engineers or expert HLS designers. In this work, we aim to take a step towards making HLS tools useful for a broader set of programmers. To do this, we made following contributions: 1) we study methodologies of restructuring software code for HLS tools; 2) we provide examples of designing different kernels in state-of-the art HLS tools; 3) we described a theoretical framework for parameterizable HLS templates for composing restructured HLS code based on design patterns, 4) we present a domain-specific framework that generates efficient FPGA designs using pre-optimized HLS templates and design patterns.

Introduction

Two clear trends have emerged in computing in the past 5-10 years: power consumption and processing massive amounts of data. Power consumption become a major metric in all areas of computing platforms from mobile devices to datacenters. We want our smart phones to last longer and datacenters to consume less power while still achieving massive computation.

Big data has reached into almost all industries to improve the quality of service. Machine learning algorithms process thousands of images to create classifiers, search engines analyse billions of search queries to find relevant data, and weather forecast systems analyse massive amounts of historical data to make better predictions. Massive data processing comes at a price. It requires huge computational power or high-performance computers. Thus, we need high-performance computers with efficient power consumption to run computationally demanding applications. The state of the art high-performance platforms (built using CPUs or GPUs) for computational expensive applications are not energy efficient.

Recently researchers have suggested using specialized hardware accelerators as one of the solutions to the worsening power consumption of computational platforms [127, 40, 111], among others. Specialized hardware is developed on different platforms such as FPGA (Field-Programmable Gate Array) or ASIC (Application-Specific Integrated Circuit). Designing an ASIC is expensive and have low time-to-market. Flexibility and programmability of FPGA makes it a primary platform to develop specialized

hardware. FPGAs are seeing widespread adoption in applications, including wireless communication, image processing, and datacenter (e.g., [111]). Despite the increasing use in these application domains, programming an FPGA largely requires an expert hardware designer. This severely inhibits the widespread use of these devices.

We aim to raise the level of programming abstraction to allow designers that have limited or ideally no understanding of the underlying hardware architecture to implement their application of choice on an FPGA. This has been the goal of numerous academic and industrial projects, many of which fall into the domain of high-level synthesis (HLS). There has been tremendous progress in the development of HLS tools since its introduction over three decades ago; however, there are several existing challenges that still hinder the ability for any designer to program an FPGA using HLS.

First and foremost, designers cannot effectively use existing HLS tools unless they have intimate knowledge of micro-architectural trade-offs. Even the best HLS tools require at least a two stage code transformation to generate efficient hardware. The initial design process converts a "normal" C application description to code that is synthesizable by the HLS tools. This is done by removing unsupported programming constructs, e.g., dynamic memory allocation [137, 57]. Then this synthesizable C code undergoes an additional conversion where it is rewritten to take advantage of specific micro-architectural features; we call this process developing "restructured" code. This must adhere to strict coding styles that are dictated by the HLS tool and the target device [13, 113]. This two stage code transformation process presents several challenges. It requires that the designer understand how to develop code that exploits low-level micro-architectural features using an abstract language. These are often not easily specified using that language, and typically require intimate knowledge of the HLS tool as well as significant hardware design expertise.

In summary, while HLS tools are meant to be used by a larger number of designers

and increase productivity, creating an optimized implementation requires substantial hardware design skills. Thus in order to successfully use today's HLS tools one needs to have: 1) domain knowledge about the application, 2) hardware design expertise, and 3) the ability to translate the domain knowledge into an efficient hardware design.

The goal of this dissertation is to make specialized hardware design process (built on FPGAs) easier so that wide variety of people (scientists and software programmers) can benefit from FPGA's high-performance and low power. Our approach is based on the observation that certain application domains have a number of basic kernels that share similar computational primitives. This indicates that these kernels can and should be generated from highly optimized templatized architectures. Based on this observation, we develop an approach based upon parameterizable templates that can be composed using common design (computational and structural) patterns. These templates describe an efficient hardware architecture for a wide variety of commonly occurring kernels for certain application domains. Common design patterns orchestrate these templates in different ways to build large complex applications.

Patterns take advantage of the fact that the task-level parallelism in many applications is organized in a structured manner. Different kernels of applications designed with templates communicate in a certain pattern. Examples of patterns are functional pipelining (bulk synchronous), Fork/Join, Merge, sliding window, and tiled computations (e.g., tiled matrix multiplication). By using these patterns and templates, we can form new templates (by composing) which can be added to the existing template pool.

In this dissertation, we present a set of reusable and optimized templates for HLS that forms a new template based on specific design (computational and structural) patterns. We propose a domain-specific framework that generates optimized HLS code without requiring any low level hardware details using reusable templates and patterns. Next, we list the main contributions of this research in the order they appear in this

dissertation:

1. Study of several application designs using high-level synthesis. These application designs provided initial guiding principles for efficient FPGA design using HLS. Some of these applications are: High Throughput Wireless Channel Emulator, Face Recognition, and Canonical Huffman Encoding.

2. Building restructured code and coding techniques for a number of application kernels for different application domains; computer vision, linear algebra and data processing (sorting).

3. An experimental study about evaluating usage of pre designed templates to design FPGA hardware.

4. A theoretical treatment of the composibility and parameterization of templates (based on patterns) in order to combine basic templates into more complex ones that can be used to build complex applications based on specific computational patterns.

5. A case study of generating optimized FPGA designs from templates based on a domain-specific framework. Specifically, we presented a framework, Resolve, which generates optimized HLS code from a simple domain-specific language written in python using templates.

6. The development of basic templates across application domains and their open source implementations.

To assist the reader of this thesis, we now give a brief overview of each chapter. Chapter 1 presents a brief overview of high-level synthesis. We first discuss the force-directed scheduling algorithm, then we introduce Vivado HLS, a high-level synthesis

tool used in this dissertation. Finally, we will present a simple "'Hello World"' in Vivado HLS and give an overview of some of the issues in current HLS tools.

Chapter 2, Chapter 3 and Chapter 4 present three application case studies using high-level synthesis to design FPGA hardware. These applications demonstrate issues and promises of the current high-level synthesis design approach. In Chapter 2, we present a design flow of using high-level synthesis for a wireless channel emulator. The goal of this chapter is to introduce the design flow of high-level synthesis for real world application and to demonstrate a variety of results obtained by HLS optimization. Chapter 3 demonstrates design and implementation of face recognition on an FPGA. The goal of this chapter is to introduce designing regular applications with HLS. Chapter 4 presents design and implementation of Canonical Huffman encoding used for datacenter applications. The goal of this chapter is to demonstrate design and implementation of irregular applications using high-level synthesis. Based on experiences gained designing these applications, we present restructured code in following chapter.

In chapter Chapter 5, we formally define restructured code and present examples of restructured code. First, we present a study on the importance of restructuring code to obtain an efficient FPGA designs with good QoR (Quality of Result). We present several case studies of code restructuring in different levels (instruction and task level) for different kernels. We also present a small survey about using HLS to design applications on an FPGA.

Chapter 6 presents a composable template based on the restructured code presented in chapter Chapter 5. The main contribution of this chapter is providing a theoretical framework for the treatment of the composibility and parameterization of templates in order to combine basic templates into more complex ones based on patterns.

Chapter 7 presents a framework that generates sorting architectures automatically by composing existing templates. In this chapter, we cover design and implementation

of sorting architectures. Then we present a domain-specific framework that generates sorting architectures automatically based on sorting templates and patterns.

Chapter 8 presents conclusion and future research directions of this thesis.

Chapter 1

Background

1.1 Introduction

There are many embedded applications such as computer vision/graphics, digital signal processing, wireless communications which benefit from field-programmable gate arrays (FPGAs). An FPGA is a reconfigurable hardware platform which can be configured after manufacturing. Due to their reconfigurable nature, FPGAs are good platform for many of these applications because they provide a good trade-off between performance and power consumption. Traditionally, FPGA systems are programmed using low level Hardware Description Languages (HDL) such as Verilog/VHDL. While programming an FPGA with HDL gives a designer finer granularity control, it has drawbacks such as it requires expertise, it is expensive, and it has long time-to-market. An alternative way of designing an FPGA system is using High-Level Synthesis (HLS) tools. HLS tools allow designers to use high level languages such as C/C++ to design FPGA systems. HLS tools promise to increase productivity (decrease design time) of FPGA design, increase portability/flexibility of design, and to allow more design space exploration.

Due to its promising features, HLS has attracted the interest of both industry and academia. The first HLS tools emerged in the 1970's and targeted application-

specific integrated circuits (ASICs), before FPGA chips were introduced in 1985 [134]. Early works include HLS tools from academia such as HAL [108], Olympus\Herculeus [44, 45, 78] and HLS tools from industry such as Cathedral 2/3 [58], Yorktown Silicon Compiler [29], and Amical [73]. These works contributed largely on basic HLS research and influenced development of other tools. For example, one of the best known HLS scheduling algorithms, Force-Directed Scheduling, was first developed for the HAL system and is still used in several academic and industrial HLS works.

Since FPGA arrived on the market, many HLS tools targeting FPGA have emerged both from academia and industry. Academic efforts include SPARK [63], LegUP [30], ROCCC [129], SA-C [65], Chisel [18], NAPA-C [60], Stream-C [59], Trident [124]. Industry efforts include Vivado HLS [41] (formerly known as AutoESL), Synopsys Synphony C Compiler [9] (former name PICO), Catapult-C (Calypto) [22], Cynthesizer [92], CyberWorkBench [133], C-to-Silicon [19] DIME-C [49], Impulse-C [3], Mitron-C [6], Bluespec System Verilog [102], Synplify DSP [120], Simulink HDL Coder [118], and AccelDSP [68].

Many of these tools are based on $C-like$ language while others are based on MATLAB, Haskell and Scala. The MATLAB based tools include Synplify DSP [120], Simulink HDL Coder [118], and AccelDSP [68]. Bluespec System Verilog [102] and Chisel [18] are based on functional programming languages such as Haskell and Scala.

Most HLS tools accept source code and optimization directives as inputs. Initial input code is usually first modified to target strict HLS requirements. This is because some constructors in the language are not supported by HLS tools such as dynamic memory allocation. This is done by replacing an unsupported part of code with equivalent code that is synthesizable. (e.g., a double pointers in C is translated into a two dimensional arrays.)

Optimization directives tell the HLS tool which part of code to optimize. In

general, most HLS tools have similar optimization directives with slightly different names. One common optimization directive is *pipeline*. This directive is given as a *pragma* in C code and pipelines loops or functions. Next, we start an introduction of basic HLS concepts followed by introduction of the Force-Directed Scheduling algorithm.

1.2 High-Level Synthesis

HLS tools create an architecture through several steps such as scheduling, allocation and binding process that generate different components to create a datapath [**?**], and each of these steps are deeply covered in [93, 54]. The general flow of high-level synthesis is shown in Figure 1.1. Initially, the HLS compiler front-end takes C/C++/Java code and transforms it to a data flow graph (DFG). The DFG is fed into high-level synthesis tool chain that does scheduling, resource allocation and binding. Scheduling is the assignment of each operation to a time slot (identifying start and end time). Resource Allocation is the process of identifying types of hardware components and the number for each type to be included in the final implementation. Binding is the process that allocates hardware components to specific operations. The final step of HLS tools is to generate HDL (Hardware Description Language) in the form of VHDL/Verilog.

In the HLS design process, designers are required to give hints and constraints in order to optimize performance. This is typically done through pragmas or other language features. Essentially, constrains are hints to a scheduler to identify the part of algorithm to optimize. Once it has its inputs (DFG, Constraints), it is the scheduler's job to define the final hardware.

In general, scheduling is most important part of any HLS tool. In next the sections, we cover one of the important HLS scheduling algorithms known as Force-Directed Scheduling.

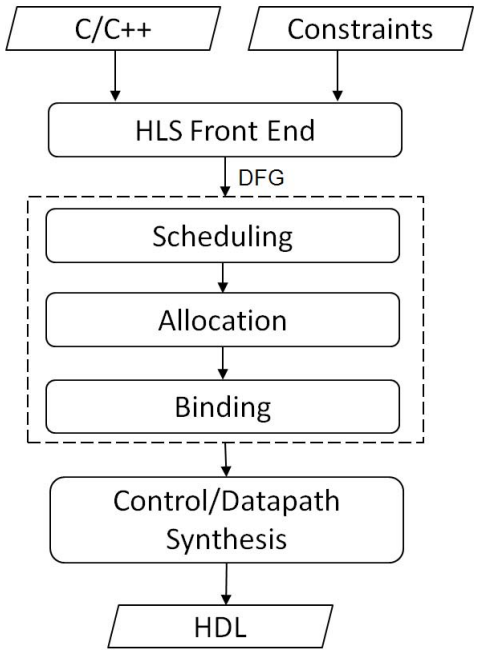


Figure 1.1. High-Level Synthesis tool flow.

1.2.1 Scheduling

Scheduling is an essential component of high-level synthesis. It determines the start and end time of each operation for a given data flow graph [93, 54]. By determining start and end times of individual operations in a given data flow graph, it extracts concurrency of the resulting implementation which affects the performance and area of the final hardware.

One of the most used scheduling algorithms for high-level synthesis is the force-directed scheduling (FDS) by Paulin and Knight. [107]. The FDS algorithm employs a heuristic approach which can be applied to resource constrained scheduling as well as time constrained scheduling. The FDS algorithm is based on a concept called "'*force*"' where operations with least force are scheduled. The FDS algorithm works by first identifying time frame of an operation. The time frame is the time interval where the operation can be scheduled. In FDS, the time frame of operation is determined by ASAP

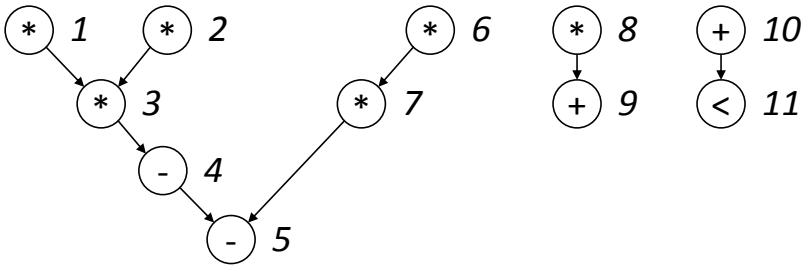


Figure 1.2. An example data flow graph.

(As soon as possible) and ALAP (As late as possible) scheduling algorithms. We show how ASAP and ALAP schedules are used to determine the time frame of a data flow graph in Figure 1.2. In the following, we give examples that will explain ASAP, ALAP and FDS scheduling based on [107, 93]. Figure 1.2, Figure 1.3, Figure 1.4, Figure 1.5, Figure 1.6 are also based on the same example given in [107, 93] for ASAP, ALAP and FDS scheduling.

The ASAP Scheduling Algorithm: The ASAP schedule works by topologically sorting operations in a given data flow graph [93]. Topologically sorted operations are scheduled as soon as all of their dependencies (predecessors) are scheduled. The goal of this scheduling algorithm is to minimize latency (number of clock cycles) under unconstrained resources. (This does not take into account area constraints.) To illustrate, how ASAP scheduling works we use an example data flow graph in Figure 1.2 from [107]. The ASAP schedule of the data flow graph given in Figure 1.2 is show in Figure 1.3. The detailed algorithm is given in Algorithm 1.

The ALAP Scheduling Algorithm: The ALAP scheduling algorithm is a latency constrained scheduling algorithm. This means, the ALAP schedule gets an upper bound λ on the latency input with data flow graph G_s. The ALAP algorithm is given in Algorithm 2. The ALAP schedule of the data flow graph given in Figure 1.2 is shown in Figure 1.4.

Algorithm 1: ASAP Scheduling

```
1 Procedure ASAP()
    Data: G_S(V,E)
    /* V: Vertices, E: Edges                                    */
    Result: t
2   Schedule v_0 by setting t_0 = 1
3   repeat
4       Select a v_i whose predecessors are all scheduled;
5       Schedule v_i by setting t_i = max(t_j) + d_j
6       where j : (v_j, v_i) ∈ E
7   until v_n is scheduled;
```

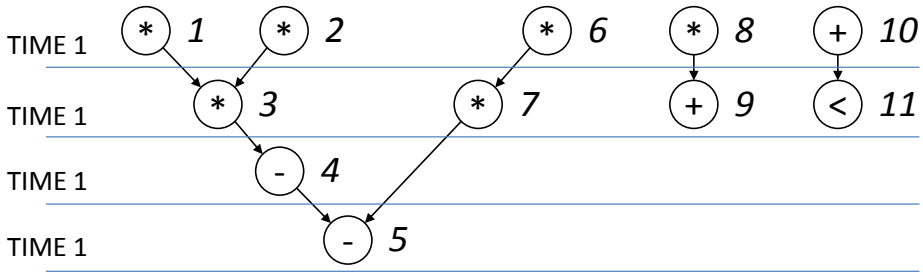


Figure 1.3. ASAP Schedule.

Algorithm 2: ALAP Scheduling

```
1 Procedure ALAP()
    Data: G_S(V,E)
    /* V: Vertices, E: Edges                                    */
    Result: t
2   Schedule v_n by setting t_n = λ + 1
3   repeat
4       Select a v_i successors are all scheduled;
5       Schedule v_i by setting t_i = min(t_j) − d_i
6       where j : (v_i, v_j) ∈ E
7   until v_0 is scheduled;
```

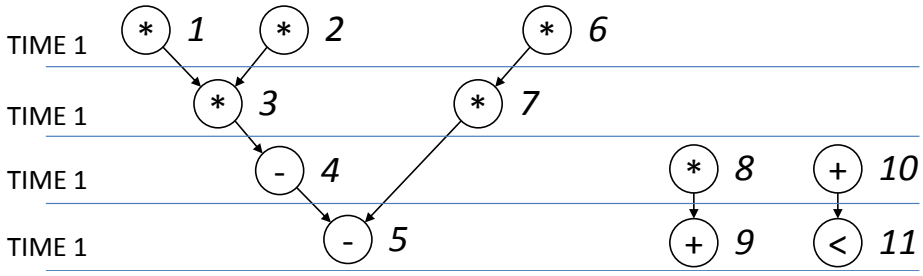


Figure 1.4. ALAP Schedule

Using the ASAP and ALAP scheduling results, time frames of each operation are determined as in Figure 1.5. The next step in FDS algorithm is to calculate the *operation probability* and the *type distributions* for each resource type.

Operation probability is a function that is equal to zero outside of the time frame of an operation and equal to inverse of the time frame within it (inside the time frame). We use $p_i(l)$ to denote the operation probability of an operation i at time l. For example, for operation 6, $p_6(1) = 0.5$, $p_6(2) = 0.5$, $p_6(3) = 0$ and $p_6(1) = 0$. (Operation 6 can only be scheduled in time slot 1 or time slot 2.) Similarly, we can determine $p_1(1) = 1$, $p_1(2) = p_1(3) = p_1(4) = 0$ and $p_2(1) = 1$, $p_2(2) = p_2(3) = p_2(4) = 0$.

The *type distribution* is the sum of probabilities of an operation that can be executed with a specific hardware resource at time l. We use $q_k(l)$ to denote type distribution of resource k at time l. For example, in time step 1 operations $1, 3, 6$ and 8 can be executed with a multiplier. Thus the type distribution of the multiplier at time step 1 is equal to $q_1(1) = p_1 1 + p_2(1) + p_6(1) + p_8(1) = 1 + 1 + 0.5 + 0.3 = 2.8$ (k=1 indicates a multiplier). Distribution graph (DG) is constructed by using type distributions from all time steps for a specific resource. The Figure 1.6 shows a DG of a multiplier and adder calculated using Equation 1.1.

There will be no horizontal bar graph(type distribution) if no operation can be schedule at a specific time. For example, no horizontal bar graph (type distribution) is shown in Figure 1.6 for a multiplier since no multiplier can not be scheduled at time step

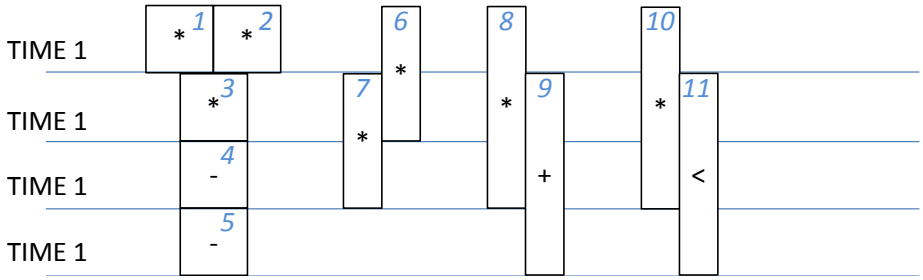


Figure 1.5. Time frames for operations

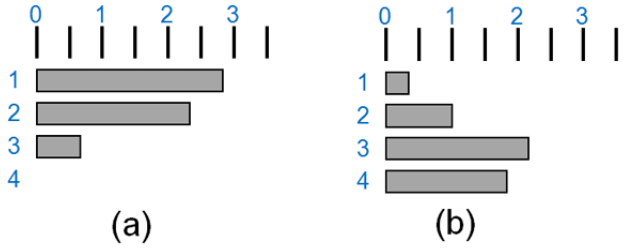


Figure 1.6. Distribution graphs for a multiplier and adder

4.

$$DG(l) = \sum P_i(l) \tag{1.1}$$

The selection of an operation to a specific time step is governed by a concept called "'*force*"'. Each operation of a data flow graph has a "'force"' associated with each time step. The algorithm works by a force calculation for each operation in a specific time step. The operation with the least force is then scheduled.

The force is analogous to Hooke's law $F = Kx$, where K is constant of a *spring*, x is the displacement and F is the force needed. In FDS algorithm, the value of distribution graph is acts like the spring constant K and the operation probability is like spring displacement. In FDS, two kinds of force are calculated for each operation: self-force and successor/predecessor force. The self-force is associated with assigning operations to specific time step and the successor/predecessor force is associated with the change

of operation dependencies. For example, scheduling the operation 6 into time step 1 changes $p_6(1)$ from 0.5 to 0.5 (1-0.5) and $p_6(2)$ from 0.5 to -0.5 (0-0.5). Assuming we schedule operation 6 to time step 1, we calculate self-force as 2.8*(1-0.5)+2.3*(0-0.5)=0.25. Similarly, when operation 6 is scheduled at time step 2, its self-force is $2.8*(0-0.5)+2.3*(1-0.5_{)}=-0.25$.

Assigning operation 6 to time step 2 indirectly implies that operation 7 will be assigned to step 3. In this case, we calculate successor/predecessor force of time step 7. This is added to the self-force of 6. The detailed FDS algorithm is given in Algorithm 3.

Algorithm 3: Force Directed Scheduling

```
1 Procedure ForceDirectedScheduling()
    Data: G_S(V,E)
    /* V: Vertices, E: Edges                                   */
    Result: t
2   repeat
3     Compute time frames
4     Compute operation probabilities
5     Compute DG
6     Compute self-force, succ/pred force
7     Schedule the operation with least force
8   until all operations are scheduled;
```

1.3 Design with HLS

We start this section by introducing of Vivado HLS from Xilinx which is used to illustrate examples in next sections [10].

1.3.1 Vivado High-Level Synthesis

Vivado HLS takes synthesizable C and C++/SystemC code as input and generates RTL. Vivado provides directives to optimize area, latency and throughput. In this way, the system development time is greatly reduced without the need to manually write RTL

code. The high-level synthesis process in Vivado includes three main steps: synthesis, RTL simulation and RTL implementation. In the synthesis step Verilog HDL, VHDL and SystemC code are generated based on the input code, and a report file is generated which includes design latency, resources and a interface summary of the design. In this step, various directives can be added by the designers to optimize the input code to meet the system timing and area requirements. This process is iterative, and eventually RTL code which meets the design requirements is obtained. Testbench wrappers for SystemC code and appropriate simulation scripts for Verilog and VHDL code are generated for simulation. The second step is RTL simulation to verify the behavioural functions of the RTL code. In this step, designers can simulate the SystemC code and Verilog/VHDL code with third party RTL simulators to verify the design. The third step is RTL implementation in which NGC files containing both logical design and constraints are generated for each module. Table 1.1 and in Table 1.2 listed several of these directives. For complete description, we refer [10] for details.

Table 1.1. Vivado HLS directives to optimize throughput and latency.

Directive names	Description
latency	Define a minimum and/or maximum latency for a specified scope.
loop_unroll	Unroll a loop, flatten a loop, or merge loops automatically.
loop_flatten	Flatten a loop
loop_merge	merge loops automatically.
dataflow	Indicate blocks of code (functions or loops) operate concurrently.
pipeline	Pipeline the computation of a function or loop.
array_partition	Distribute the array across several memories.
array_reshape	Combine partitioned arrays into a single array.

The typical design process for Vivado will first optimize throughput and then focus on area optimizations. Area optimizations include bit-width optimization, removing design hierarchy, and limiting the number of operations. Table 1.2 describes common directives used for area optimization.

Table 1.2. Vivado HLS directives to optimize area.

Directive names	Description
allocation	Limit the number of operations in scheduling and binding stages.
array_map	Combine arrays to utilize memory more efficiently.
resource	Specify the function unit used for binding.
inline	Allows optimization to be performed across function hierarchies.

Among above directives, dataflow is one of the important features provided by Vivado HLS. The dataflow is used to implement task level parallelism (both streaming and none-streaming) of functions or loops. For example, if we need to design a system with two blocks working as follows: the first block produces an output after multiplying the input with a constant, and the second block (consumer block) reads an input from the producer and decides if input is odd or even. Task level parallelism between the producer and the consumer block is achieved by using dataflow. This means the first instance of the consumer block can be parallelized with the second instance of producer block. This increases throughput of design by a factor of 2X. When multiple tasks are parallelized as in this case, a memory is created between the producer and the consumer tasks. The memory can be a block RAM or a FIFO. If the tasks are not streaming, then a memory that is large enough to store all the output of producer must be created. if the tasks are streaming, then a FIFO must be created. In this case the size of FIFO must be large enough to store outputs of producer in a given rate. Streaming in Vivado HLS is supported by $hls :: stream < TYPE >$ class. Here $TYPE$ is any primitive types such as int, char or

Initiation Interval (II) as a measurement of throughput in HLS. II is used as a measurement of instruction level parallelism (provided by pipeline) or as task level parallelism (provided by dataflow). Ideally, $II = 1$ is the optimal and $II > 1$ means suboptimal. $II = 1$ means the task (design) can accept new data every clock cycle.

1.4 HLS Hello World

In this section, we provide a brief overview of HLS design process. A HLS design must contain at least two modules. The first module is the top level design. A hardware will be generated for the top level design. The second module is the testbench. The testbench is where application programmers or HLS users write the "'main()"' function. Modern HLS tools (e.g., Vivado HLS) allows designers to use a C testbench to verify the functionality of top level design. Listing 1.1 and Listing 1.2 shows an example a top level design and a testbench. Design in Listing 1.1 has two inputs namely $A[]$ and $B[]$. In the given codes, each of $A[]$ and $B[]$ has size of $SIZE$. Let us assume $SIZE = 8$ for now. Having a design like the one in Listing 1.1 is the starting point of using HLS. Next, we synthesis the top level function (in this case "'HelloWorld"'). The input to the HLS tool is a top level function, a clock period, the FPGA device and optimization directives. Depending on the target clock period, the target FPGA device and the optimization constraints, HLS tool generates different HDL code. For example, synthesizing the code in Listing 1.1 without any optimization constraints produce a sequential hardware as in Figure 1.7. Assuming the loop body takes 3 clock cycles (1 clock cycle for reading data, 1 for addition and 1 for storing the data), the loop will complete in $8 * 3 = 24$ clock cycles.

```
1 void HelloWorld(int A[SIZE], int B[SIZE]) {
2   #pragma HLS UNROLL factor=8
3   for(int i=0;i<SIZE;i++){
4     B[i]=A[i]+3;
5   }
6 }
```

Listing 1.1. HLS Hello World: HLS code for top level design

```
1
2 //Software code for verification
3 void Software_HelloWorld(int A[SIZE], int B[SIZE]) {
4   for(int i=0;i<SIZE;i++){
```

```
5     B[i]=A[i]+3;
6   }
7 }
8
9 int main() {
10   std::vector<int> input(SIZE, 0);
11   std::vector<int> output_hw(SIZE, 0);
12   std::vector<int> output_sw(SIZE, 0);
13   Boolean fail=0;
14
15   for(int i=0;i<SIZE;i++) {
16     A[i] = ,...;
17   }
18
19   //Call Hardware HelloWorld
20   void HelloWorld(input, output_hw);
21
22   //Call Software HelloWorld
23   void Software_HelloWorld(input, output_sw);
24
25   //Functionality Verification
26   for(int i=0;i<SIZE;i++) {
27     if(output_hw[i]!=output_sw[i]) {
28       fail=1;
29     }
30   }
31
32   if(fail==1) {
33     printf("'FAILED\n"');
34   }
35   else {
36     printf("'PASSED\n"');
37   }
38
39   return 0;
40 }
```

Listing 1.2. HLS Hello World: HLS code for testbench

Now let us assume that we want to generate an architecture like in Figure 1.8 (a). In other words, we want to parallelize all additions and finish the computation in 3 clock cycles. This can be achieved by using *partition* and *unroll* directives provided by HLS. Using *partition* we put elements of $A[]$ and $B[]$ into separate memories (registers) in order

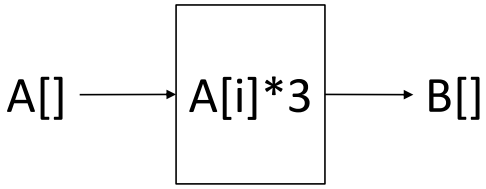


Figure 1.7. HLS Hello World Hardware Architecture

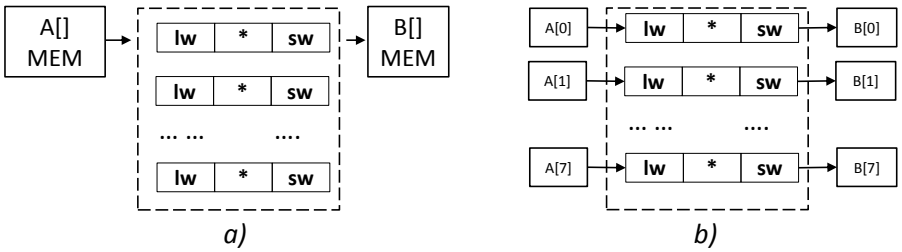


Figure 1.8. Hardware architectures for hello world HLS code. a) Hardware architecture created by Listing 1.1 b) Hardware architecture that we want to create.

to access them at the same clock cycle. Using *unroll* we fully unroll the loop. These optimizations can be seen in Listing 1.3. The design generated by this code finishes in 3 clock cycles. Since we are partitioning input arrays (in this case into individual elements), actual architecture will look like as in Figure 1.8 (b).

```
1 void HelloWorld(int A[SIZE], int B[SIZE]) {
2   #pragma HLS ARRAY_PARTITION variable=A complete dim=1
3   #pragma HLS ARRAY_PARTITION variable=B complete dim=1
4   for(int i=0;i<SIZE;i++){
5   #pragma HLS UNROLL factor=8
6     B[i]=A[i]+3;
7   }
8 }
```

Listing 1.3. Optimized HLS Hello World

1.5 Conclusion

Increasing design cost of accelerated hardware is pushing the design community to develop tools that shorten the time-to-market duration beyond RTL design. High-Level

Synthesis has been one such a tool that provides promise in increasing productivity. In this chapter, we provided quick overview of concepts behind HLS tools and provided basic HLS optimizations with ”Hello World” HLS. We demonstrated basic optimization pragma usage giving an example. While current HLS tools provide handy optimizations (pragmas) to generate optimized hardware, some real world application design flow involves several stages in order to get optimized hardware from HLS. In next chapter, we demonstrate generally accepted design flow of current HLS tool. We will do this by presenting wireless channel emulator design with HLS tool.

Chapter 2

Wireless Digital Channel Emulator

2.1 Introduction

In this chapter, we will introduce generally accepted design flow using high-level synthesis. Since HLS accepts C-like code, many HLS designs start from a legacy code written in high level languages by a domain expert. These languages can be classic programming language such as C/C++, MATLAB. The general flow of using HLS tool is shown in Figure 2.1. This flow has several code transformations.

Current HLS tools require at least a two stage code transformation to generate efficient hardware. The initial design process converts a unsynthesizable legacy application code (Initial software code) to code that is synthesizable (baseline) by the HLS tools. We name the result of synthesizable code baseline. Baseline design is obtained differently based on initial software code. For example, if the initial software code is a legacy C/C++ application, then this is done by removing unsupported programming constructs, e.g., dynamic memory allocation [137]. If the initial legacy code is written in MATLAB, then designer must convert the MATLAB code into synthesizable HLS code. Then this synthesizable C code (baseline) undergoes an additional conversions. In the next code transformation, designers manually modify the code in order to assist HLS tool to generate efficient hardware; we call this process developing "restructured" code.

Restructured code can be done in different way depending on memory access pattern, communication between different modules and complexity of the application. We will cover restructured code in deep in Chapter 5. In software design, designers use primitive types such as int, char, long which have fixed number of bits. In hardware design, we can use arbitrary number of bits for variables and arrays. In the bit-width optimization stage, we set required bit widths for variables and arrays. This stage will reduce area and increase frequency. The final stage (Optimizations or PUP) is where designers use HLS provided pragmas such as pipeline, unroll and partition (PUP).

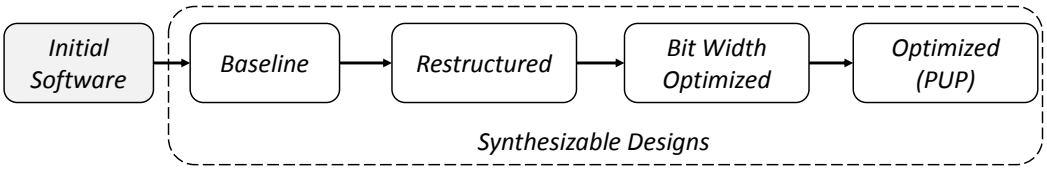


Figure 2.1. Overview of HLS design flow

Next we are going to demonstrate the design and implementation of wireless channel emulator using the aforementioned HLS design flow. For this particular application, we are given a specific throughput to achieve. Next we discuss importance of implementing a wireless channel emulator on an FPGA.

One of the many challenges in the development and evaluation of wireless systems is testing and verification. Ultimately, radio systems need to be field tested to ensure that they satisfy the stated requirements. However, this is expensive, time consuming, and difficult to repeat. Simulation can overcome these problems, but is ultimately limited by fidelity issues and/or excessive run times; thus it is more suited to the early stages of development. Real-time hardware in the loop (HWIL) radio frequency (RF) channel emulation fills the gap left between simulation and field testing. A good RF channel emulator will provide repeatability, high-fidelity, and the opportunity to test complete radio systems in different network scenarios. This is especially important for state-of-the-

art wireless systems which incorporate complex link adaptation algorithms with the goals of improving spectral efficiency, reducing inter-symbol interference (ISI), and increasing resilience against multipath, Doppler shift, and fading.

Software defined radios (SDR) use agile spread spectrum, ultra wideband and other techniques to cognitively utilize the available spectrum. They can operate over different frequency band and have been proposed for next generation military and civilian voice, video and data links. In order to evaluate the effectiveness of SDR networks in responding to adverse link conditions, an appropriate real-time, accurate RF channel emulator is required. A channel emulator simulates the antenna and propagation of the signal, modifying it depending upon a predetermined scenario. It works directly with a variety of existing radios, and thus must handle a wide range of frequencies with rapidly varying channels over a range of distances.

To meet the challenging requirements in testing and evaluating the state-of-the-art wireless systems, we look at a portion of this overall system. We designed a digital wireless channel emulator (DWCE) on an FPGA platform. This is high fidelity and broadband. It provides real-time emulation for radios operating in a frequency range of 2 MHz to 2 GHz. It has a range resolution of 0.25 km. It also provides link losses of up to 130 dB and has a resolution of 0.5 dB.

A major focus of this work is to evaluate the suitability of implementing our DWCE using the AutoESL high-level synthesis (HLS) tool. Our DWCE has to handle wide range of dynamically changing parameters such as Doppler effect fast fading, and multipath. We carefully describe the optimization process (baseline, restructured, bit-width, PUP) that we used to obtain a design that meets the required specification requirements. This was done solely through modification to the code given to the HLS tool. And only through careful optimization of this code, with an understanding of how this code effect the HLS process, were we able to meet the required performance metrics.

We initially started form a legacy MATLAB code which is several hundred lines. Thus it provides an interesting example of a complex, high throughput design using high-level synthesis tools targeting FPGAs. We show how to effectively use the HLS tools to achieve the target design goal.

The specific contributions of this paper include:

1. Designing an FPGA implementation of a wireless channel emulator operating across a 2 MHz - 2 GHz spectrum with a range resolution of 0.25 km and link losses up to 130 dB with 0.3 dB resolution.

2. Studying the effectiveness of the AutoESL HLS tool to design a complex, high throughput application. The emulator has three major modules, *PathDelays*, *PathGains*, and *ChannelFunction* with over 180, 500, 120 lines of code, respectively. Each module has $C \times M^2$ complexity where M is the number of paths in each channel and C is the number of loops in each module.

3. Describing the benefits of different optimization techniques on the area and throughput of the emulator. Starting with a naive implementation (MATLAB), we utilize and analyze a number of optimizations in order to reach to target performance metrics.

The remainder of this chapter is organized as follows: Section 2.2 describes the wireless channel emulator model in detail. In Section 2.3.1, we give detailed description of hardware design and implementation of wireless channel emulator. Section 2.5 presents experimental results. We provide an overview of related work in Section 2.7, and we conclude in Section 2.8.

2.2 Wireless Channel Model

This section provides a description of the wireless channel model and algorithms that our emulator uses.

Figure 2.2 provides a graphical description of the proposed channel emulator. The emulator connects to N radios using analog-to-digital (ADC) and digital-to-analog converters (DAC) in series with an RF module. The radios connect to the RF module in lieu of an antenna. The RF module down converts the radio signal to baseband signal. Then the ADC converts the baseband signal to digital signal as the input to the emulator. The emulator combines all the digital signal inputs and processes them with the wireless channel model (described in the next subsection) to produce the digital outputs for each radio. The digital output signals are converted to an analog signal by the DAC and up converted to radio frequency by the RF module. Finally, each radio receives an RF signal as if it is in a real field testing environment with $N-1$ other radios. We designed the emulator to accept signals within a 2 MHz to 30 MHz range.

The wireless channel can be broken down into a set of receive/transmit paths. For example, an emulator with 8 radio ports has up to 56 channels with the assumption that the receive/transmit paths are non-symmetric. A transmitted RF signal encounters reflection, diffuse scattering, and diffraction due to the ground and the obstacles in the environment. There is also a Doppler effect which is caused by the motion of the transmitter and the receiver. The emulator is responsible for calculating these environment effects in the digital signal domain.

A *single* wireless channel is described in Eq (2.1), where $si_{t-i\Delta\tau}$ are the previous n input complex samples, $w(t)_i$ is a dynamically changing set of complex weights, and so_t is the complex output of the channel at the present sample time. The weights $w(t)_i$ control the delays and the gains of output signals to emulate the environment effects of

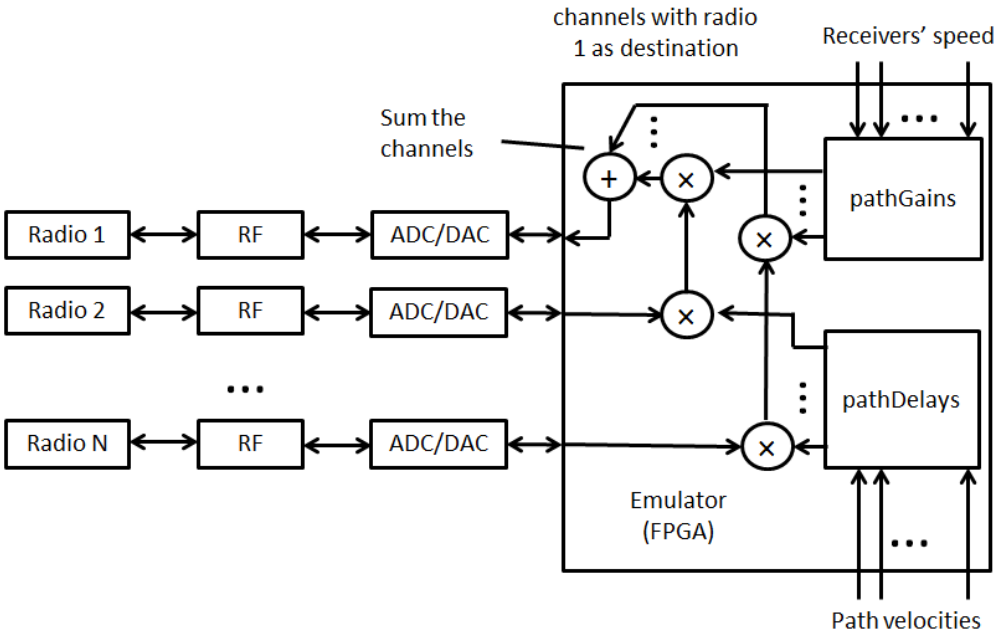


Figure 2.2. System overview of the wireless channel emulator.

the simulated scenario.

$$so_t = \sum_{i=0}^{n} si_{t-i\Delta\tau} w(t)_i \tag{2.1}$$

The methods *PathDelays* and *PathGains* are used in tandem to compute $w(t)_i$. *PathDelays* emulates the real field testing path delay phenomenon by integrating the path velocity. In this model, we use tapped delay line (TDL) model. A TDL model stores input samples in a FIFO. Then, it computes the delay of ADC samples in the FIFO. The output of *PathDelays* is a series of sample numbers from the delay line representing the first ADC sample of a consecutive group for each path and a series of weight vectors for each path. In order to access the samples in the delay line, *PathDelays* calculates the index for each path. Thus, the output of *PathDelays* is *index[N]* and *weight[K][N]* where N is the number of paths and K is predefined constant.

PathGains emulates the simulated scenario's path gain. It implements a statistical

method to model the effect of many independent scatters near the receiver where none of them is dominant. *PathGains* filters a complex white noise source for each path. The result of the filter represents the path gain considering the effect of reflections, obstacles, etc. *PathGains* also computes the Doppler effect by controlling the coefficients of the white noise filter. The result of *PathGains* is a series of complex gains used to multiply the output digital signals of *PathDelays*. The output signal of *PathGains* is equivalent to the signal as it is transmitted across a real fading environment. In our model, we use complex signals. Thus the output of *PathDelays* is the matrix *gains[2][N]*. After getting, *index[N]*, *weight[K][N]*, and *gains[2][N]*, we can calculate real and imaginary part of complex sample output using Eqs (2.4) and (2.5),

$$tapline_{ij} = delayline_{index} \tag{2.2}$$

$$taps_i = \sum_{i,j=0}^{N,K} tapline_{ij} * weight_{ji} \tag{2.3}$$

$$so\_r_i = \sum_{i=0}^{N} gains\_r_i * taps\_r_i - gains\_i_i * taps\_i_i \tag{2.4}$$

$$so\_i_i = \sum_{i=0}^{N} gains\_r_i * taps\_i_i - gains\_i_i * taps\_r_i \tag{2.5}$$

where *gains_r*, *taps_r* and *gains_i*, *taps_i* are the real and imaginary part of the complex signals for each path.

As shown above, each channel between the various radios are modeled as described in Equation (2.1) which is further describe in Equations (2.2), (2.3), (2.4), (2.5). The remainder of the paper focuses on implementing *a single channel emulator*. This can be duplicated to model *N* channels.

2.3 Hardware Design and Optimization

In this section, we introduce the architecture of the emulator and discuss the design process using HLS. We provide an algorithmic description of the emulator, and present how different optimization of AutoESL affects area (BRAM, DSP48E, FF, LUT) and latency. We present four different results of different design optimizations (Baseline, Restructured, Bit-width, and PUP).

2.3.1 Architecture

The block diagram of the system is shown in Figure 2.3. It consists of three modules: *PathDelays*, *PathGains* and *ChannelFunction*.

ChannelFunction accepts an input signal, path velocities, and carrier frequency, and calculates the complex output sample signal. It sends path velocities to *PathDelays* and gets the index of starting signal and weight. The index is used to retrieve the delayed signal samples from the input signal as described in Eq (2.2). The results are used to calculate taps for each path using Eq (2.3). The samples from the input signal, weight and gain from *PathGains* are used to calculate the complex signal (*SampleOut*) as described in Eqs (2.4) and (2.5).

The *PathDelays* module takes as input the path velocities of each path in a channel. It computes the indexes of the input samples and the weights to calculate the expected sample at the specified path distance for all paths. It has three sub blocks as shown in Figure 2.3. Each sub block depends on results from the previous. It outputs vectors *index[N]* and *weight[K][N]*. *PathDelays* and *ChannelFunction* require a throughput of 2-30 MHz in order for the emulator to meet the required operating characteristics.

PathGains generates a random signal, interpolates the signal with a filter, and calculates the gain by multiplying the Doppler effect, interpolation and CORDIC outputs.

PathGains has a throughput requirement of 100kHz-200 kHz to meet the operating characteristics. The result matrix *gain[2][N]* is integrated with every 80 results of *PathDelays*.

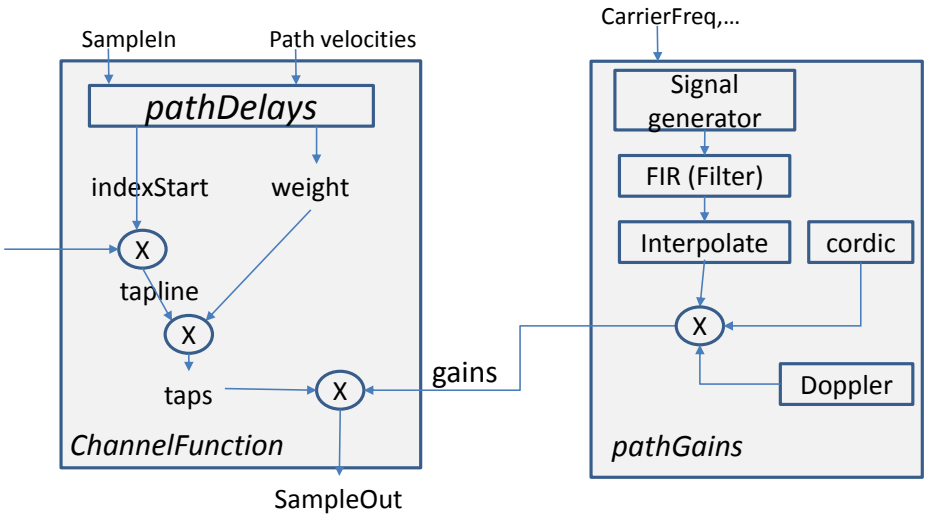


Figure 2.3. A block diagram of the channel emulator, which includes three main functions *ChannelFunction*, *PathGains* and *PathDelays*.

We started with Matlab source code for the three modules. We converted these to synthesizable C code, e.g., limited pointers, no dynamic memory, and writing our own standard C functions (max, min, pow, sin, cos). Then, we implemented the functions with AutoESL. We achieved the target design goal using the following steps: code restructuring, bit-width optimization, and pipelining, unrolling and memory partitioning. These steps are discussed in detail in the following.

2.3.2 Baseline

The baseline architecture uses the synthesizable C code "as is" and implemented it with AutoESL. Additionally, we created testbenches and top modules for each of *PathDelays*, *PathGains* and *ChannelFunction*. The primary goal of this step is to get an initial, functionally correct design with AutoESL. Typically, the area of design is

large; it is not uncommon for this to be millions of times larger than the area of final optimized version. The throughput at this stage can be worse than original software implementation due to large clock period difference between CPU and FPGA. Additionally the manual implementation of standard C functions may incur additional cost since the standard C functions are highly optimized. To wit, the baseline design of *PathDelays*, *PathGains* and *ChannelFunction* are 106, 17, 4.8 times slower than their equivalent software implementations as in Figure 2.10. Again, the goal here is to make sure that the translated synthesizable C code is functionally correct. Once we have this, we perform optimizations that can significantly improve both the area and throughput.

In the following, we briefly describe the baseline implementations of *PathDelays*, *PathGains* and *ChannelFunction*.

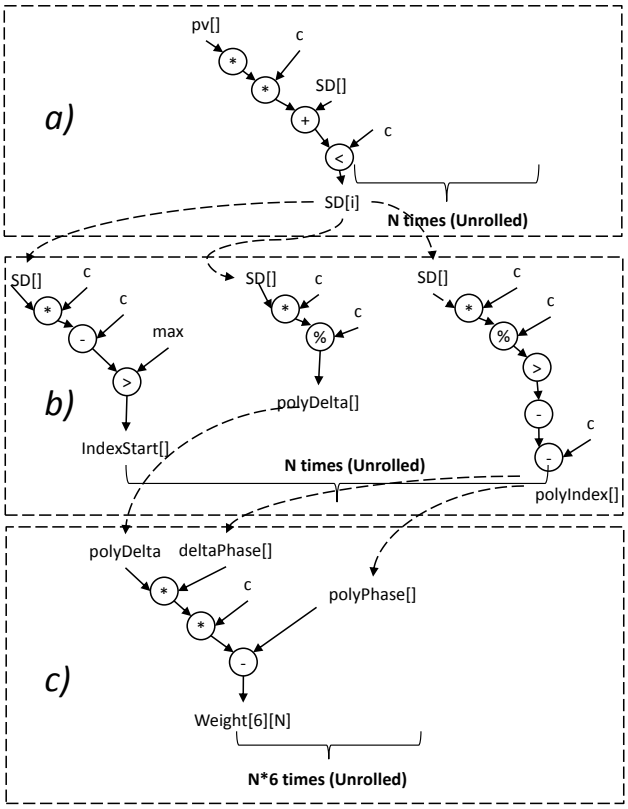


Figure 2.4. The final restructured *PathDelays* code. There are three blocks: (a) sample delay calculation, (b) index calculation, (c) weight calculation.

PathDelays: This module calculates the sample delay by multiplying the path velocity by a constant for each of the *N* paths. Then the maximum of sample delays and pre-computed index is selected. This procedure is shown in Figure 2.4 (a). Then using the previous sample delay, it calculates the *indexStart*, *polyDelta* and *polyIndex* as shown in Figure 2.4 (b). Finally, it calculates the weight for each path as shown in Figure 2.4 (c). The initial *PathDelays* function had seven blocks of code where each block is a loop. The result of each block is used in the subsequent block.

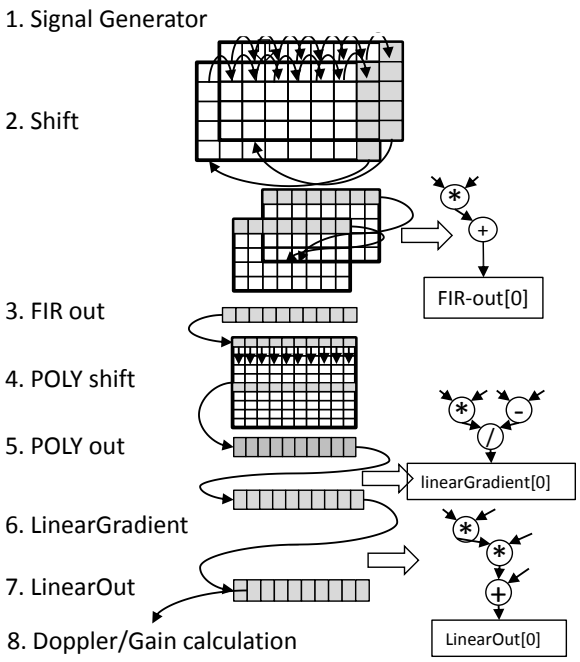


Figure 2.5. The computation involved in the PathGains module.

PathGains: This module has several steps. The first step involves signal genera-

tion through interpolation and Doppler effect calculation. Initially, it generates a random signal based on a linear feedback shift register (LFSR). Then the signal is filtered by FIR and POLY phase shaping filters to perform a linear interpolation. In the signal generation, we generate *NxMxK* random values where N is the number of paths, $M = 256$ (M is fixed and same as number of elements in Jake's spectrum), and $K = 2$ since we need to generate a value for a complex signal (real and imaginary parts). The final interpolation is a matrix of size *1xN*. The interpolation results are used to calculate the gain for each path. Figure 2.5 shows the signal generator, interpolation and Doppler/gain calculation. In the Figure 2.5, Step 1 is signal generation, Steps 2-7 are interpolation and Step 8 is Doppler and gain calculation. To implement the base code for *PathGains* module, we converted 216 lines of Matlab code to 700 lines of C. Then we removed loops that have variable number of bounds. In one case, we used the TRIP_COUNT directive to describe the minimum and maximum range of the loop. We wrote code for the standard C functions. For some functions, we removed them from design since they are use only few times with constant inputs, e.g., pow(2,32) was replaced with constant.

ChannelFunction: This module did not require much optimization.

2.3.3 Code Restructuring

The goal of this stage is to perform optimization on the code itself without using AutoESL directives. In this step, we performed two optimizations; 1) loop merging, unrolling and loop flattening, and 2) removing computations that can be done offline. Loop merging, unrolling and loop flattening saves clock cycles if used properly. AutoESL also provides directives to merge/flatten loops automatically.

As discussed in the previous section, we implemented some of standard C functions in baseline stage. In the design, some of those functions are used for a limited number of times. For example, we are using *pow* function 11 times in *PathDelays* module

with constant input. Therefore, replaced these with a constant.

We describe the code restructuring optimizations for each of the modules in the following.

PathDelays: We optimized *PathDelays* in software using loop merging and removing functions that can be calculated offline. First, we removed two of the seven blocks because they are doing initialization. Then we did loop merging to reduce the number of blocks to three. Figure 2.4 shows the final structured code of *PathDelays*. Each block uses data from the previous block. Therefore, the optimal hardware will implement this design in three clock cycles. We unrolled the loops in each block as shown in Figure 2.4. Unrolled loops decreased number of clock cycles, however, this is limited by the access to the array. In AutoESL, arrays are implemented as block ram. Code restructuring of *PathDelays* reduced BRAMs, FF and LUT usage by 8 to 5(36%), 7581 to 4070 (46%) and 10170 to 4631 (54%) respectively. The number of clock cycles decreased from 12430 to 2000 due to loop merging and removing some of the initialization step to offline. Decreased number of clock cycles resulted in increase in throughput drastically by 83%. The code size reduced from 300 lines to 180 lines.

PathGains: The initial code had 17 blocks. Figure 2.5 shows most but not all of these blocks. Each block is embedded inside a loop. 4 of 17 blocks are responsible for 80% of computation. These are blocks 1, 2, 3 and 8 in Figure 2.5. Block 1 initializes an $N \times 256 \times 2$ array with random values from a linear feedback shift register. Blocks 2 and 3 perform shifting and FIR filtering operations. Block 8 applies the Doppler effect and calculates the gain for each path. We focused on these blocks and performed loop merging, expression balancing, and loop unrolling. In Figure 2.6, we presented some of the optimizations. In Figure 2.6 (a), we merged two loops. This is a snippet of code for FIR output calculation. This merging reduced the number of clock cycles from 50215 to 41250, a 20% reduction. Then, we performed expression balancing and loop unrolling

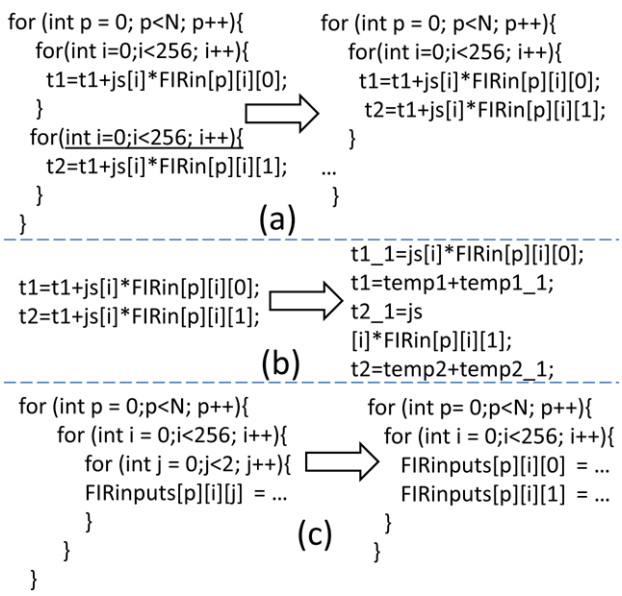


Figure 2.6. Optimizations on the *PathGains* module including: (a) Loop merging, (b) Expression balancing, (c) Loop unrolling.

as shown in Figure 2.6 (a) and (b). We applied expression balancing to FIR calculation and gain calculation. We unrolled the innermost loop of the FIR initialization and FIR shifting as shown in Figure 2.6 (c). These two optimizations reduced the number of clock cycles from 41250 to 29730, and then from 29730 to 20785. The area before and after restructuring code did not change significantly. The number of BRAMs and DSP48 remained same, and number of FF and LUT decreased slightly. All designs had the same clock period. In general, optimizing the code in software resulted in a 59%(20215 to 20785) reduction in the number of clock cycles. Due to code restructuring, C code was reduced to 700 lines to 600 lines.

ChannelFunction: Once again this module did not require much optimization since we coded this module with restructured way. This module uses 7 BRAM, 56 DSP, 4448 FF, and 4968 LUT.

2.3.4 Bit-Width Optimization

Bit-width optimization impacts both throughput and area. We manually calculated the required bits for all major variables and arrays. We achieved this by calculating values of variables and arrays using floating type. Then we find out needed bit width for each variable by analyzing the max/min range for each variable. Then using the AutoESL's internal arbitrary precision integer and fixed-point data types (ap_int, ap_fixed), we defined new types. This step required more time and effort than any other step.

PathDelays: We applied bit-width optimization on the restructured code. This reduced DSP48E, FF and LUT by 51% (39 to 19), 89% (4070 to 424), 87% (4631 to 563) respectively from the previous stage (Restructured). It also increased the throughput by 90% due to decreased number of clock cycles from 2000 to 191. The number of BRAMs remained same.

PathGains: We performed bit-width optimization on top of restructured code for the *PathGains* module. *PathGains* is largest module, and bit-width optimization took a substantial amount of time. This optimization increased throughput by 56%. Again the throughput increase is resulted from decrease in number of clock cycles from 20785 to 7967. Bit-width optimization reduced the number of BRAMs, DSP48E, FF and LUT by 50% (84 to 42), 58% (112 to 47), 69% (67088 to 20399), and 65% (64176 to 22121) respectively.

ChannelFunction: Finally, we applied bit-width optimization to *ChannelFunction*. It decreased number of clock cycles from 902 to 529 which resulted in increased throughput by 40%. This step also reduced the number of DSP48E, FF and LUT by 78% (56 to 12), 84% (4448 to 702), and 79% (4968 to 994) respectively. Number of BRAM remained same.

2.3.5 Pipelining/Unrolling/Partitioning (PUP)

AutoESL provides a *pipeline* directive to pipeline functions and loops. AutoESL partitions small arrays inside the region and unrolls the loop where the pipeline directive is used. If there are large arrays (not automatically partitionable), then the user can specify that they be divided into separate memories (BRAMs) or as registers. AutoESL's *dataflow* directive performs coarse grain pipelining at the function and loop level. The dataflow directive increases concurrency between functions and loops by executing the function or loop as soon as the data is available. Thus, it creates a memory between the blocks. Currently, the dataflow directive has limited capabilities. For example, it can be used where data is accessed in a FIFO order. We tried using the dataflow directive in our designs, and it significantly reduced the performance (by a factor of approximately 40X); it did result in a smaller design, but overall this tradeoff was not beneficial. Therefore we did not use the dataflow directive in our final design.

We describe the specific optimizations that we performed on each of the modules in more detail in the following.

PathDelays: We applied *pipeline* and *partiiton* on the bit-width optimized code. We partitioned 5 BRAMs into registers. In AutoESL, *complete* partitioning a BRAM results discrete registers. We used one *pipeline* directive on top of function with initiation interval (II) = 1. The final design is shown in Figure 2.4. Our design had 4 clock cycles with 5.394 ns of clock period. Since partitioning BRAM and pipelining increase amount of parallelism, it also increased area. The number of DSP48E, FF and LUT increased by 57% (19 to 30), 85% (424 to 786), and 651% (563 to 4230).

PathGains: Since large BRAMs limit the pipelining, we partitioned the BRAMs that are accessed frequently within a pipelined region. We partitioned 12 of the 42 BRAMs by mapping them to registers. We pipelined the entire function, thus all of the

loops were unrolled. This increased the throughput by 93% due to decreased number of clock cycles from 7967 to 501. The area also increased due to duplicated hardware resources. The BRAMs decreased by 12 since moved this data into registers. DSP48E usage increased by 82% (47 to 86), FF increased by 68% (20399 to 34421), and LUT increased by 75% (22121 to 38893).

ChannelFunction: We applied pipelining with an II=1 and partitioned seven BRAMs in *ChannelFunction*. This improved throughput by 98% and resulted higher area. DSP48E usage increased from 12 to 40, FFs increased from 702 to 2392 and LUTs increased from 994 to 5008.

2.4 Results

The experimental results are obtained using Xilinx AutoESL (Version 2011.1) and ISE (Version 13.1). For each design we report the number of BRAM, DSP, FF, LUT, and the latency. The latency is calculated by multiplying number of clock cycles with clock period. We compare the latency of the software version against the hardware for each design. Then we present the speedup of final hardware version over optimized software version. For each experiment we target an xc6vlx240t FPGA.

2.4.1 Verification/Integration

After applying each optimization, we verify the correctness of each hardware design against the original software using the AutoESL simulator (autosim). After verifying the correctness of each module in each step, we integrate the final optimized version of the three modules from PUP stage to make a complete emulator. This is done to understand how the HLS tools handle large designs. Theoretically, the tools should be able to optimize the complete design in an optimal manner. But this is not always possible due to the complexities of the HLS problems. There are two different ways to

integrate three sub modules into one.

Integration using AutoESL: In this case, we create a top level function in C (which we call *emulator*) for the three modules. Then we synthesis the *emulator* with the optimization pragmas from the PUP stage. This may create a slower and larger design if AutoESL is unable to correctly optimize the resulting large data flow graph in an optimal manner.

Manual Integration: Here, we create a top module in Verilog and manually integrate the optimized designs for *PathDelays*, *PathGains* and *ChannelFunction*. Then we synthesis the top module using ISE. It is a divide and conquer method using a hand partitioned method for synthesizing the sub modules.

The block diagram of the emulator is shown in Figure 2.12. In the integrated design, we use two different clocks. One clock for *PathGains* which is 100 Mhz, and other clock for *PathDelays* and *ChannelFunction* which is 151 Mhz. We use two different frequencies as described in Table 2.2. The modules *PathGains* and *PathDelays* run in parallel and their outputs (gain, weight, indexstart) are consumed by *ChannelFunction* function to calculate *SampleOut* complex signal.

The *PathDelays* module calculates *indexstart* and *weight*. In each case four and six clock cycles, respectively, are required to calculate the *SampleOut* signal. This requires a total 62 ns is needed to calculate *SampleOut* with 10 clock cycles at 151 Mhz (151 Mhz=6.62 ns period, $10 \times 6.62 = 62ns$). The *PathGains* runs slowest and it outputs results every 4994 ns (0.2 Mhz). In our statistical channel model, we use the same result of the *PathGains* (same *gains*) for every 80 SampleOut calculation. (*PathGains* runs in 0.2 Mhz (4994 ns and 4994/62 ns =80). The *PathGains* module writes output to registers. We represented it with FIFO as shown in the Figure 2.12.

We verified the correctness of emulator of both designs. The input to the emulator are the sample input signal, carrier frequency, receiver speed, LOS gain, and path

velocities. The output of the emulator is the sampled output signal in a complex form.

2.5 Experimental Results

We present area results first. We present this in two figures. The first shows the number of BRAM with DSP (Figure 2.7). The second displays FF with LUT (Figure 2.8). This is essentially a graphical recap of the optimizations described in the previous sections. *ChannelFunction* has same code for baseline and restructured designs.

One of the biggest application tradeoffs that we explored is the number of paths per channel. This highly depends on the application scenario. Some situations require fewer paths than others. For example, communication in a city, or mountainous area typically require more path to accurately model the channel due to reflections of the signal. The same is true for indoor communications. Our model uses five paths as a default parameter; other common models such as Stanford model and ITU model use three paths and six paths [52, 83].

A big advantage of HLS is design space exploration. In this case it allows us set parameters such as number of paths using minimal modifications to the C code, and synthesizing the architecture to determine its design metrics. We studied how different paths change the hardware implementation. Figure 2.9 shows how the area (in number of slices) changes as the number of paths increase. In general it is linear relationship, e.g., an implementation with five paths is 5$\times$ larger than the design with one path. In Figure 2.9, *PathGains* does not have 5$\times$ relationship due to constant area of the CORDIC core. The throughput does not change since the computation required by adding the different paths is for the most part independent, and thus can be parallelized. All of the other reported results use five paths. Table 2.1 shows more detailed area results of *PathDelays*, *PathGains* and *ChannelFunction* for five path implementation. The slice results are equivalent to the rightmost three bars in Figure 2.9.

Table 2.1. Device utilization characteristics for PathDelays, PathGains and ChannelFunction for five paths.

	Slices	LUT	FF	DSP48E	BRAM
PathDelays	584	1843	411	30	0
PathGains	2783	8759	7044	53	30
ChannelFunction	1131	3469	1798	40	0

Table 2.2. Clock cycles, clock period and latency of each sub module.

	Clock cycles	Clock period (ns)/ Frequency (MHz)	Latency (ns)/ Throughput (MHz)
PathDelays	4	5.394 /185 Mhz	21 /47 Mhz
PathGains	501	9.97 /100 Mhz	4994 /0.2 Mhz
ChannelFunction	6	6.62 /151 Mhz	37 /26 Mhz

Table 2.3. Device utilization characteristics of complete emulator for emulators integrated by AutoESL and Manually.

	Slices	LUT	FF	DSP	BRAM
Emulator (Manual)	5020	14221	9564	123	30
Emulator (AutoESL)	14283	47190	30691	70	60

Figure 2.10 compares the latency of the software implementation with the latency of four designs (Baseline, restructured, bit-width and PUP). The latency of the software is measured on a Intel i7 3.4GHz multi-core machine. We applied OpenMP on top of restructuring code to get optimized software implementation. We compared the hardware performance against the best optimized software performance. The optimized software version of *PathDelays* and *PathGains* has 29X and 18X times better performance than initial software. This is shown in Figure 2.11. The module *ChannelFunction* has little parallelism and the computation in module *ChannelFunction* is not so complex. Thus, there was little room for the optimization of *ChannelFunction*. We used the same

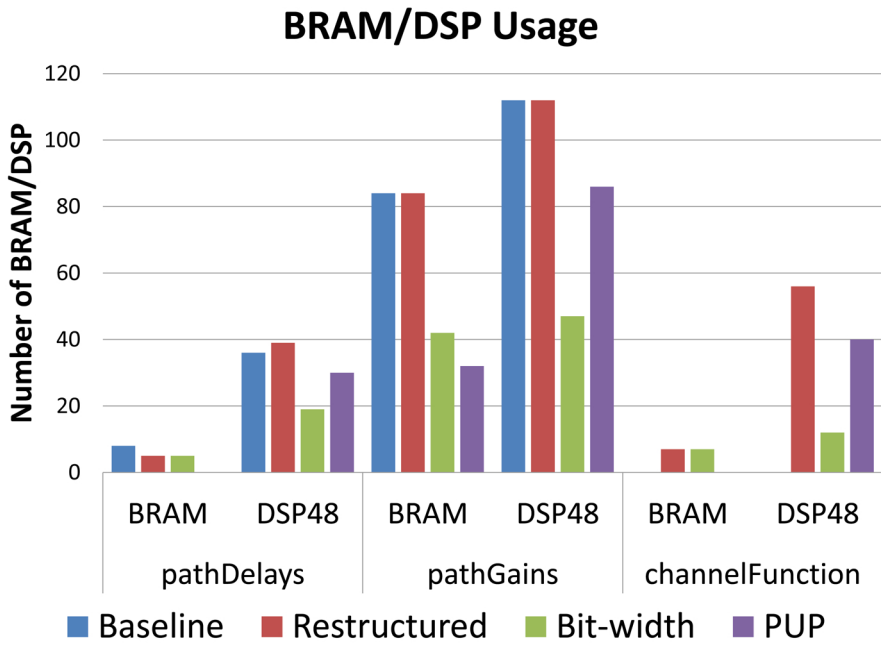


Figure 2.7. Number of BRAM/DSP for Baseline, Restructured, Bit-width and PUP optimizations for *PathDelays*, *PathGains* and *ChannelFunction*.

ChannelFunction implementation for all of our comparisons.

In general, as shown in the Figure 2.10, hardware implementations corresponding to baseline and restructured versions are slower than the optimized software version. This is due to the clock cycle difference between CPUs and FPGAs. For example, the baseline implementations are slower than the software equivalents by a factor of 106X, 17X and 4.8X of *PathDelays*, *PathGains*, and *ChannelFunction*.

In Figure 2.10, the final hardware performance results obtained by PUP optimization are compared to the software performance. The hardware results are better by 46X, 6X and 42X over the equivalent software version. *PathGains* gave the worst speedup. This is largely due to the fact that it uses CORDIC which requires 79 cycles thus limiting the initiation interval. The comparison of the baseline software code with our final hardware implementation has significantly better results; the hardware implementation of *PathDelays* and *PathGains* is 200,000X and 2000X faster respectively. In other words,

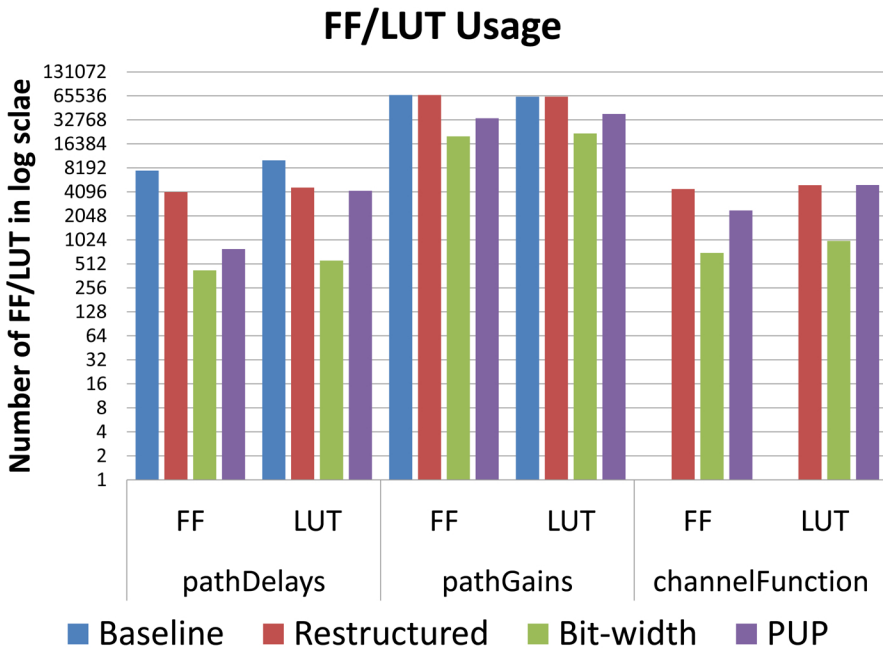


Figure 2.8. Number of FF/LUT for Baseline, Restructured, Bit-width and PUP optimizations of *PathDelays*, *PathGains* and *ChannelFunction*.

restructuring the code gives significant software performance benefits.

Table 2.2 presents the number of clock cycles, clock period (frequency) and latency of each sub module. The integrated emulator runs in 62 ns (16 Mhz). This is 41X times faster than software emulator. The software emulator uses the same optimized sub modules.

In the first row of Table 2.3, we present area results for manually integrated version of emulator. In the second row of Table 2.3, we present the area results for AutoESL based integrated version of the emulator. The AutoESL based integration has 5X larger area than manual integrated one while being 2X times slower than manually integrated emulator.

We spent a total of five weeks to design the emulator. Of that, three weeks went towards understanding the application and writing the restructured code. Two weeks was spent performing bit-width optimization and PUP with the majority of this time spent on

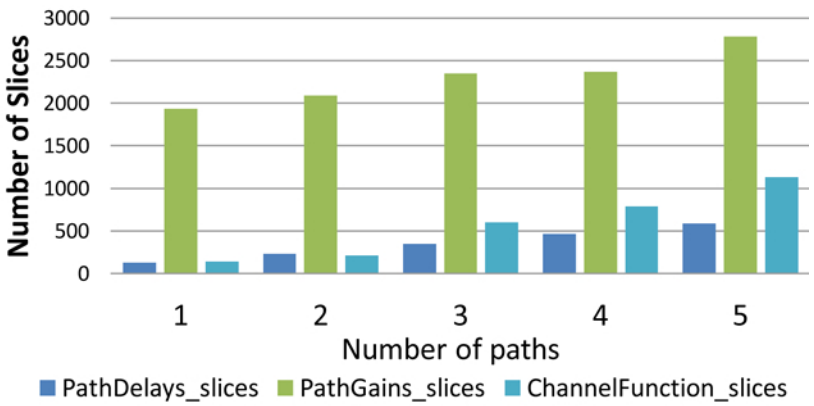


Figure 2.9. Slice numbers for scaling number of paths

bit-width optimization.

2.6 Discussion

Quality of Result: The ultimate way to compare the results from AutoESL is to compare the final optimized design with hand coded version. This is a difficult endeavor and one that we would argue is not totally necessary. HLS tools are reaching the point where if you code the input design appropriately, then you will get a design close to if you designed it from RTL. And the HLS tools allows you to quickly to change the architecture to see its effects. This is not to say that HLS tools are a panacea. In fact, we hope this article relays the fact that HLS tools require a good understanding of how the hardware and the synthesis process works. At this point these tools are still far from giving great results on code that was designed by a software programmer. The user of these tools needs to understand how the tool will synthesis the final architecture in order to get the best results.

Previous research [23, 98, 27] designed different models and use different fading and use different parameters. To our best knowledge, this is the first work that implements statistical model. The emulator by Iskander et al. [71] only gives a software version in

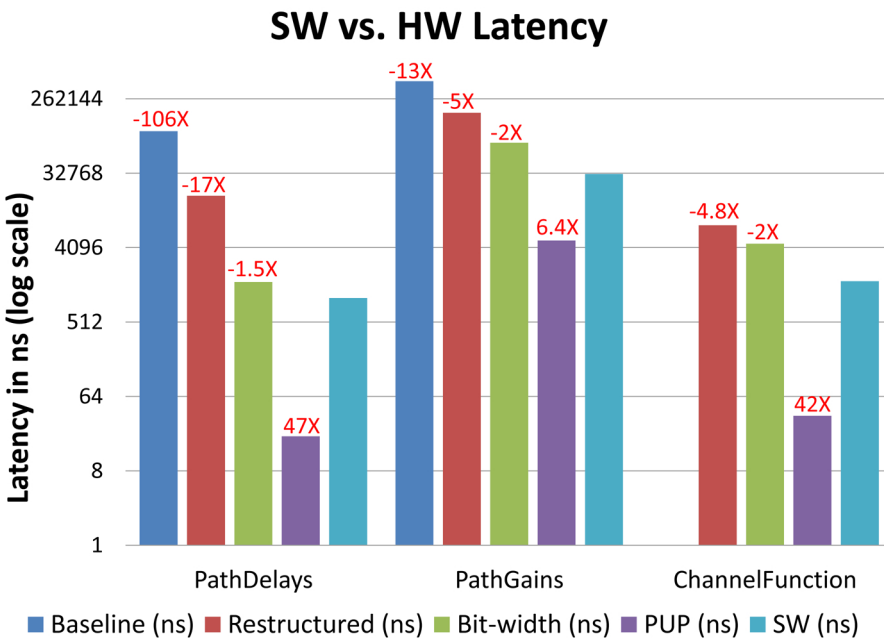


Figure 2.10. Software versus hardware latency for Baseline, Restructured, Bit width and PUP optimizations of *PathDelays*, *PathGains* and *ChannelFunction*. Negative (-X) means slower by X than software(SW). Positive X means faster by X than software.

Matlab. Our performance results are thousands of times better than this Matlab version. Another way to justify the result is we estimate the optimal performance and area for each module. However, the estimation may not be accurate for larger modules such as *PathGains*. Thus it may not be fair comparison. Therefore, it is not possible to do fair comparison with other hand coded designs at this point.

2.7 Related Work

There are several previously published wireless channel simulators and emulators using an FPGA [23, 98, 27]. Borries et al. [23] focused on stationary 802.11 laptop type wireless connections. Our emulator is looking at broader scope of networks by modeling longer and dynamically varying channel delays, Doppler effects and fast fading. This requires accounting for multipath, Doppler effects, and link loss. The emulator by Murphy et al. [98] uses a Nakagami fading model and accounts for Doppler effects.

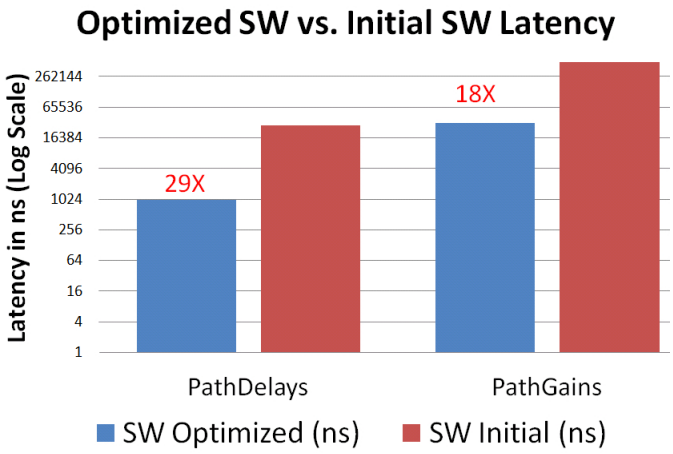


Figure 2.11. Performance of initial and optimized versions of software for *PathDelays* and *PathGains*.

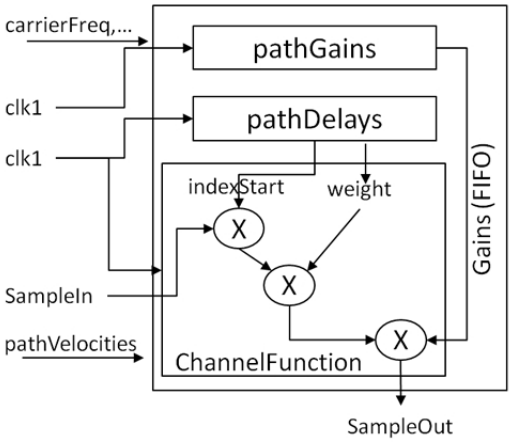


Figure 2.12. Block diagram of the emulator.

However, they do not handle continuously and smoothly varying delays. And while their emulator can model a Doppler shift, it is an approximation of the Doppler effect. In case where we do not have to up and down convert, Doppler scaling is handled completely by the continuous delay model. If we do have to down and upconvert, then that delay model in combination with our Doppler shift correction gives us the complete Doppler Effect.

Buscemi et al. presented study of scalability of digital wireless channel emulators using cluster of FPGAs [27]. They implemented tapped delay line (TDL) channel model

in VHDL and prototyped it on a two node FPGA system. In our opinion, the major contribution of the work is the system design of many DWCE across an FPGA based compute cluster. Our work differs in two ways. First, we implement statistical wireless channel model; the channel model that we use is significantly more complicated with respect to implementation. It is outside the scope of this paper to argue whether the channel model that we use is better, but it is clear that it is more complex (in terms of hardware implementation) and thus makes for a more interesting HLS case study. Secondly, we focus more on the design of the DWCE using HLS tools rather than the overall system design. Our DWCE could be used within the system that they propose.

Our FPGA implementation is similar to the wireless channel simulator developed by Iskander et al. [71]. They implemented multipath fading channel simulator in Matlab. Our emulator is several thousand times faster than the simulator.

There is a substantial amount of research around the use of high-level synthesis tools to implement a wide variety of applications on FPGAs [41]. AutoESL is used to implement applications including compressive sensing [36], stereo matching [113], a sphere detector for broadband wireless systems [103], and real-time HDTV [132]. Catapult-C has been used to implement 64-QAM decoder [122], and a sphere decoder [99]. ROCCC is used to design a Smith-Waterman algorithm [28] and a molecular dynamics algorithm [128]. Impulse C is used to implement a tomographic reconstruction algorithm [135].

2.8 Conclusion

In this paper, we presented the design and implementation of wireless channel emulator using high-level synthesis tool (AutoESL). We described the underlying model of our wireless channel emulator. We developed an FPGA implementation of the emulator that met the design constraints for a single channel. We presented a good design practice

(design flow) that is used to implement complex applications with high-level synthesis. We studied how the different stages of this high-level synthesis design flow affect the area and performance results. This chapter shows that there is huge variance in performance from a baseline design to optimized design (10x-106X). These variance is caused by several factors such as restructuring, bit width optimizations and PUP. Next, we will highlight some of the lessons learned from this experience.

Lessons learned: Our experience shows that achieving the target performance and area depends a lot on the code size, the quality of the restructured code and the application. We spent a relatively small amount time achieving the target goal for the smaller modules, i.e., those with fewer lines of code. We spent a disproportionate amount of time optimizing *PathGains* as it contained significantly more lines of code. Initially, we did not restructure the *PathGains* code well (we achieved 25 kHz with poor restructuring). Then we spent time deeply analyzing the code, e.g., drawing sample hardware architectures, manually performing some design space exploration to derive different micro architectures. This gave us insight into what to optimize and how. An important lesson is HLS tools require that the user understands what architectures are being generated. And while they allow you to more quickly get to a specific RTL implementation, a good design requires someone who deeply understands what the optimizations are doing. This is especially true for larger applications with tight constraints (i.e., *PathGains*).

Another major issue that we faced was the synthesis times for the larger applications. In our initial trials at restructuring the *PathGains* module, AutoESL would run for hours, run out of memory and then crash. Better optimized designs solved this problem; therefore the synthesis time is not necessarily a function of the number of lines of code rather a function of how many lines of RTL code the HLS tool creates.

Final lesson is HLS may give very poor design in initial stage. The initial results

are somewhat very conservative and getting better design highly depends on designer's ability to understand the application (domain knowledge) and designer's knowledge of current HLS tool.

Acknowledgements

This chapter contains materials that is published in International Conference on Field-Programmable Technology (FPT), 2012. Matai, Janarbek; Meng, Pingfan; Wu, Lingjuan; Weals, Brad; Kastner, Ryan. The chapter also contains additional materials (mainly larger figures) that was cut from the publication due to space constraints. The work described in this chapter is a done in collaboration with Pingfan Meng, Lingjuan Wu and Brad Weals. The dissertation author is the primary investigator and author of this work.

Chapter 3

A Complete Face Recognition System

3.1 Introduction

Face recognition is a challenging research area from perspectives of both software (algorithmic solutions) and hardware (physical implementations). A number of face recognition algorithms have been developed in the past decades [125] with various hardware based implementations [114, 33, 35, 101, 61, 109, 131]. All previous hardware implementations assume input to the face recognition system to be an unknown *face* image. Current hardware based face recognition systems are limited by the input image and they fail if the input is not a face image. Practical face recognition system does not require input to be a face, instead would recognize face(s) from any arbitrary video which may or may not contain face(s) potentially in the presence of other objects. Therefore, an ideal face recognition system should first have a face detection subsystem which is necessary for finding a face in an arbitrary frame, and also a face recognition subsystem which identifies the unknown face image (from face detection subsystem).

We define the c*omplete face recognition system* as a system which interfaces with a video source, detects all face(s) images in each frame, and sends only the detected face images to face recognition subsystem which in turn identifies the detected face images. We designed and implemented a real-time and complete face recognition system

consisting of a face detection subsystem, a downsampling module and a face recognition subsystem. The face detection subsystem uses the system developed by Cho et al. [38, 37], which is publicly available at [2]. The face recognition subsystem uses Eigenface or PCA (Principal Component Analysis) algorithm [125]. The complete system interfaces with a camera, sends the video data to the face detection subsystem, which in turn sends detected faces to the face recognition subsystem via the downsampling module as shown in Figure 3.1. Our face recognition system automatically identifies or verifies a person from a digital image, a video frame or a video source while previous works [114, 33, 35, 101, 61, 109, 131] simply implemented what we describe as the face. The complete face recognition system works as below:

1. Face detection subsystem looks for faces in many different sizes across each frame. When a face is found, the (x, y) coordinates, width and height of the detected face is sent to the downsampling module.

2. The downsampling module receives a signal from the face detection subsystem indicating a face is detected, it reads the detected coordinates, width, height and every individual pixel value. It subsequently downsamples the face to 20×20 size.

3. The downsampled 20×20 image data is sent to the face recognition subsystem which returns the index of a person whose face image is closest to the unknown input image.

In this work, we describe the design and implementation of a face recognition architecture using Eigenface algorithm. We design and implement a face recognition subsystem on an FPGA using both pipelined and non pipelined architectures. In each case, we evaluate system performance on a different number of images. Then we show how to integrate face recognition and face detection using a downsampling module which

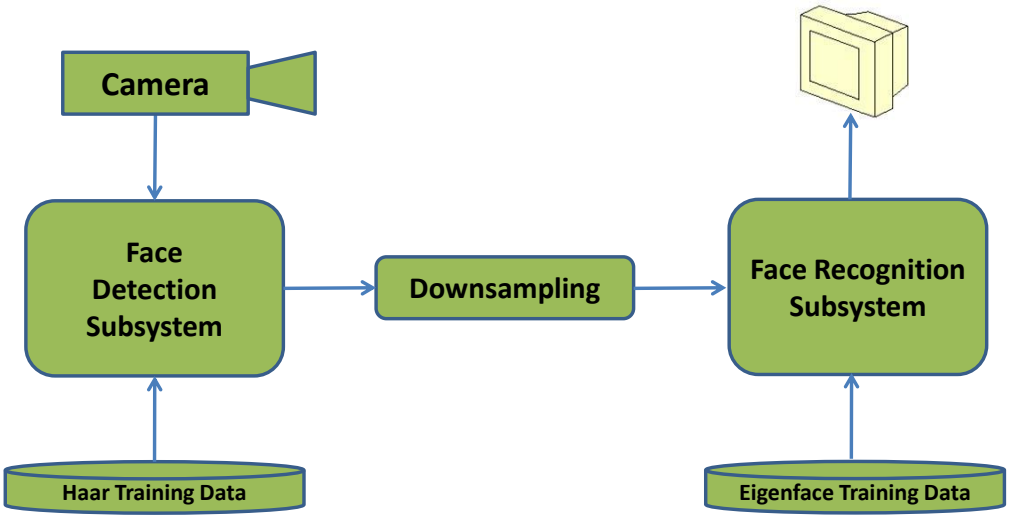


Figure 3.1. Overview of our complete face recognition system on an FPGA. Video data is received from the camera and sent to the face detection subsystem which finds location of the face(s). These face(s) can be in any size; the face detection subsystem looks for faces in a sliding window starting at 20×20 and going all the way to the size of the frame. The architecture then performs downsampling of the detected face to 20×20, and sends these 400 pixel values to the recognition subsystem. The face recognition subsystem identifies the person and sends the name of the person to the display.

is responsible for preprocessing detected face images from the detection subsystem to satisfy the requirements of the face recognition subsystem.

The rest of this chapter is organized as follows: Section 3.2 present about the face recognition subsystems. In Section 3.3, we explain the complete face recognition system architecture and its implementation results. Section 3.5 provides concluding remarks.

3.2 Face Recognition subsystem

The face recognition subsystem assumes 20×20 image data as an input, then identifies the image based on training data set. In this section, we introduce the architecture and implementation of the face recognition subsystem.

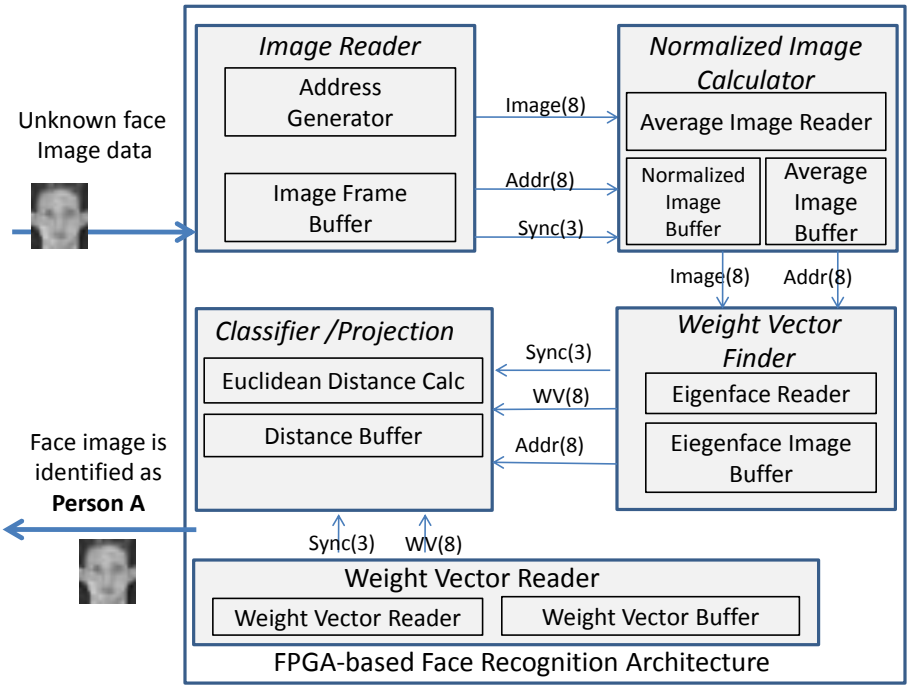


Figure 3.2. The block diagram of face recognition subsystem implemented on Virtex-5 FPGA.

3.2.1 Architecture of the Face Recognition Subsystem

The block diagram of the face recognition subsystem is shown in Figure 3.2 which consists Image Reader, Average Image Calculator, Weight Vector Finder, Weight Vector Reader and Classifier/Projection modules. These building blocks are explained in more detail as follows:

Image Reader

The image reader module reads 20×20 image and stores each pixel of an unknown image on the Image Frame Buffer. These are the pixels of an unknown image that is to be recognized. Already stored pixels are sent to Normalized Image Calculator Module in order to start normalized image calculation.

Normalized Image Calculator

The normalized image calculator module finds the differences between average image and input image. The average image reader reads the image pixels from average image buffer, then input image pixels are subtracted to find normalized image. The normalized image is stored in the normalized image buffer.

Weight Vector Finder

The weight vector finder module calculates weight values for input image using previously calculated normalized image and Eigenvector values. The Eigenvector values are read by Eigenface reader from Eigenface image buffer. The Eigenvector values are generated in PC and stored in a block RAM. The weight vector finder is the most computationally expensive step in software face recognition, and the calculation complexity of each weight vector is parallelized in hardware.

Weight Vector Reader

The weight vector reader is used by classifier/projection module for retrieving weight vector values that are generated in training stage and stored in the block RAM.

Classifier/Projection

The classifier module utilizes weight vectors (from weight vector reader module) and weight vector for unknown image (from weight vector finder module). Then classifier finds the distance between each weight vector from weight vector reader module and weight vector of unknown image. For each calculation of distance, we compare the current distance value with previous. If the current value is smaller, then it is stored to the distance buffer. Otherwise, the value in distance buffer remains unchanged. Finally, the index of a person which corresponds to a minimum distance in distance buffer is sent to display (or other output device) as an identified face.

3.2.2 FPGA Implementation

The implementation of the face recognition subsystem is performed in two steps. The first step generates the training data and the second step is face recognition.

Training Data Generation

The training data is generated using OpenCV library [24]. The training data generation is the first step in face recognition subsystem implementation [125]. We used two different kinds of face image databases as training data. Firstly, we evaluated feasibility of the face recognition subsystem using ORL database of faces from Cambridge University Computer Laboratory [7]. We refer ORL database as a "*set1*" throughout the remainder of the article. Using 100 images of 10 different individuals from the *set1*, we generated training data using OpenCV [24]. All 100 images has the same size of 20×20 pixels, and each image is aligned according to its nose and eye locations in order to generate good training data as described in [125]. We also collected 60 images of 6 individuals in our lab at University of California, San Diego which we refer as "*set2*". The images in *set2* are converted to gray scale, downsampled to 20×20, aligned by nose and eye locations, then given to OpenCV to generate training data. In the following sections, we introduce details of the implementation based on data *set1*.

The training data provides us with an average image Ψ, weight vectors for each image Ω_i, and Eigenvectors μ_i. Assuming $\Gamma_1, \Gamma_2 ... \Gamma_{60}$ represent the initial 60 images provided for training, the following image data is generated:

- An average image Ψ of size 20 by 20
- A weight vector $\Omega_1, \Omega_2, ... \Omega_{60}$ for each image. The size of weight vector Ω_i for image i is 59. In total, we have to save a 59×60 matrix for each Ω_i in the form of $\Omega = \omega_1, \omega_1, ..., \omega_{59}$.

- 59 Eigenvectors of size 20×20. The set of Eigenvectors is = $\mu_1, \mu_2 ... \mu_{59}$

In essence, the training data transforms the sixty images into a linear combination of weight vectors and eigenvectors. For instance, Γ_1 can be represented as below:

$$\Gamma_1 - \Psi = \omega_1 * \mu_1 + \omega_2 * \mu_2 + ... + \omega_{59} * \mu_{59} \tag{3.1}$$

Face recognition

After generating the training data, we store the average image Ψ (size of Ψ is 20×20), weight vectors for each image Ω_i (size of Ω_i is 59×1), and Eigenvectors μ_i (size of μ_i is 20×20) in a block RAM. Then, we implement face recognition in three steps: normalization, weight vector calculation and projection to find if $\Gamma_{unknown}$ belongs to any of 6 individual's face in training data. The architecture of the face recognition when using 60 images (10 images of 6 different individuals) is shown in Figure 3.3.

Normalization: Given an 20×20 unknown input image $\Gamma_{unknown}$, the first step of the face recognition is the calculation of normalized image Φ. Given the training data, it is straightforward to calculate the normalized image. The average image Ψ is subtracted from unknown image pixel by pixel since both the average image and the unknown input image have the same size (20×20). The average image buffer is stored in a block RAM, and input image buffer is implemented for storing the unknown input image. Since there are 400 operations and each operation is independent, the subtraction of the average image pixels from unknown image pixels can be performed in parallel. After the normalized image is calculated, the resulting pixels are sent to weight vector calculation step.

Weight vector calculation step: The architecture finds the weight vector for an unknown image using Equation 3.2. In this step, the $\mu_1, \mu_2 ... \mu_{59}$ Eigenvectors and normalized image Φ are used to calculate weight vector of unknown input image. At

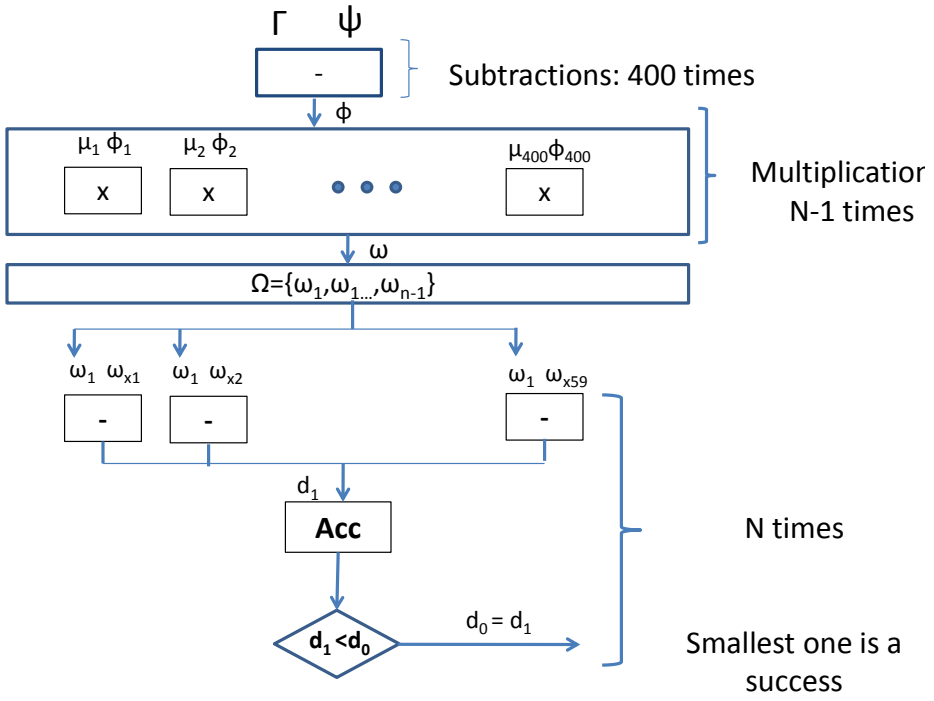


Figure 3.3. The n is the number of images. In this case $n = 60$. The $\mu_1, \mu_2 ... \mu_{400}$ are values of one eigenvector. ω_{xi} are weight values corresponding to image x. The x is between 1 and 60 in our case. d_0 is sufficiently large number and d_1 is the distance between the weight vector of unknown image and first image in training set. This process continues with the number of images. Last, the index of a person who has least distance between unknown image is returned as a result.

this point, the normalized image Ψ is calculated and stored in a register. Therefore, the calculation of each of 59 weight vector element ω_i is be parallelized since Eigenvectors are read from block RAM independently of each other. Eigenvector buffer has 23600 16 bits elements (the 59 eigenvectors each contain $20 \times 20 = 400$ 16 bit elements). The normalized image is a 20×20 matrix.

$$\omega_i = \mu_i^T \Phi \tag{3.2}$$

where $i = 1$ to 59.

Projection step: The Euclidean distance between weight vector of unknown input image and weight vectors of trained image are calculated using a nearest neighbor search. There are 60 weight vectors of each trained image each containing 59 values. We use the Euclidean distance to calculate the distances $d_1, d_2 ... d_{60}$ between each of training images and unknown image using weight vectors. The weight vectors of training images are stored in weight vector buffer in a block RAM. The size of weight vector buffer is 59×60. The Euclidean distance calculation is performed on all 60 weight vectors Ω_i. The weight vector of unknown input image and this Euclidean distance calculation are independent operations, and therefore these two operations are performed in parallel. For each calculation of distance value, we compare the new calculated distance value with the distance value in a distance buffer (old distance value). If the newly calculated value is smaller than the old distance value, then we overwrite the newly calculated distance value and index which corresponds to this value to the distance buffer. This process continues 60 times. Finally, the index of smallest value among $(d_1, d_2 ... d_{60})$ is returned from distance buffer as the index of the person identified.

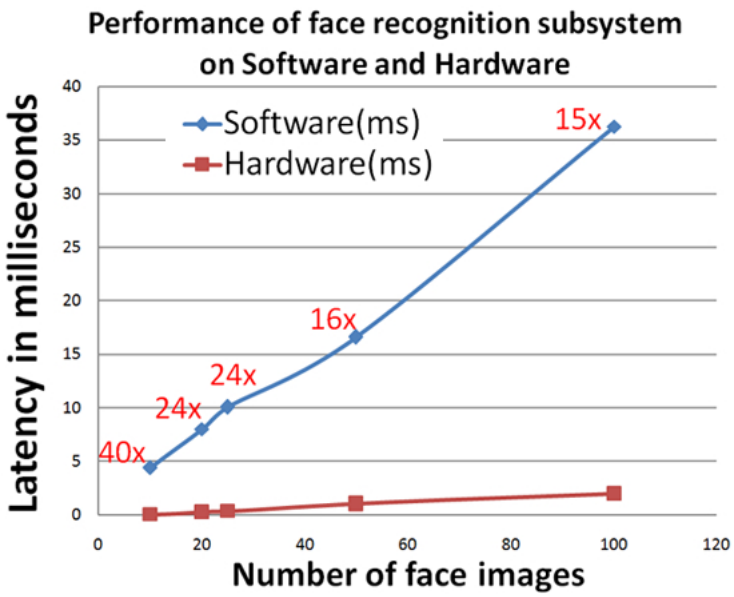


Figure 3.4. The performance comparisons between software and hardware implementation of the face recognition subsystem.

3.3 Experimental Results

This section presents experimental results using the two data sets. The first set contains 100 images from *set1* and the second is 60 face images from *set2*. Figure 3.4 shows the performance comparisons between the software and hardware implementations of the face recognition subsystem using 10, 20, 25, 50 and 100 images from *set1*. When using 100 images, the face recognition subsystem achieves average speed up of 15X over the equivalent software implementation. The software experiment was done on multi-core machine machine with Core2 Duo CPU@3.33 GHz, RAM 4 GB specification.

Figure 3.5 (a) and (b) show latency and latency cycles respectively for 40, 50 and 60 face images from *set2* with pipelined and non-pipelined implementations. The device utilization summary when using *set2* with pipelined and non-pipelined implementations is also shown in Figure 3.5 (c) in number of slices, LUTs, RAMs (BRAMs), and DSP48s.

We used ModelSim for simulation and verified the correctness of the face recognition subsystem using LEDS (South, East, North and Center LEDs) on an FPGA board. Since there are only 6 different individuals, 4-bit number is assigned to represent four LEDs on the board. In the next section, we furthermore integrate the subsystems to make the complete FPGA based real-time face recognition system.

3.4 Implementation of the Complete Face Recognition System

In this section, we present the downsampling module used to connect the face detection and the recognition subsystems. Then we describe the complete face recognition system which is a combination of all of these subsystems. Figure 3.6 provides an overview of the architecture for the complete face recognition system.

The downsampling module is notified when a face is detected by the face detection subsystem. After being notified, the downsampling module reads the face image data using the coordinates, width and height given by the face detection subsystem. According to the size of detected face image data, the downsampling module reduces the detected face to 20×20 and sends these 400 pixel values to the face recognition subsystem. The downsampling module resizes each detected face so that they are suitable for as input into the face recognition subsystem. The downsampling module has several ways to deal with down sampling of detected face.

We introduce a factor which is used to calculate how many pixels we would skip in order to downsample a $x \times x$ image into a 20×20 image. The factor depends on the size of detected face. For instance, if the size of detected face is 60×60, then the factor would be 3 in order to downsample the 60×60 size face image to 20×20. We can find the factor using $factor = detectedfacesize/20$. Finally, when the detected face is appropriately downsampled, the downsampling module checks if the face recognition

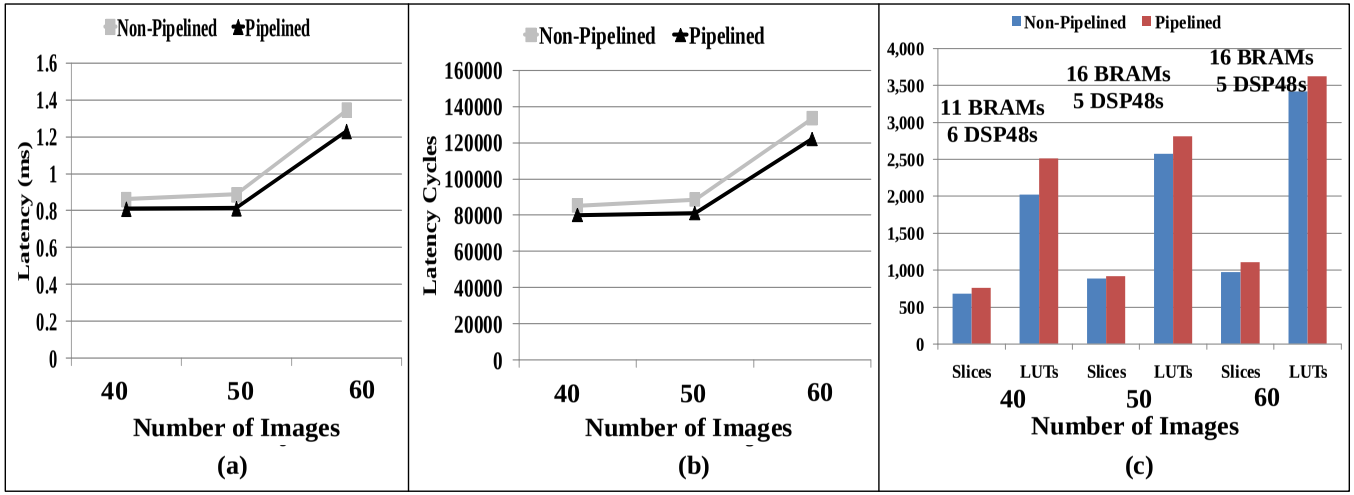


Figure 3.5. Part (b) shows the latency cycles for pipelined and non-pipelined implementations using 40, 50 and 60 images. Part (c) shows the device utilization summary for the number of slices, LUTs, block RAMs (BRAMs) and DSP48s for both pipelined and non-pipelined implementations.

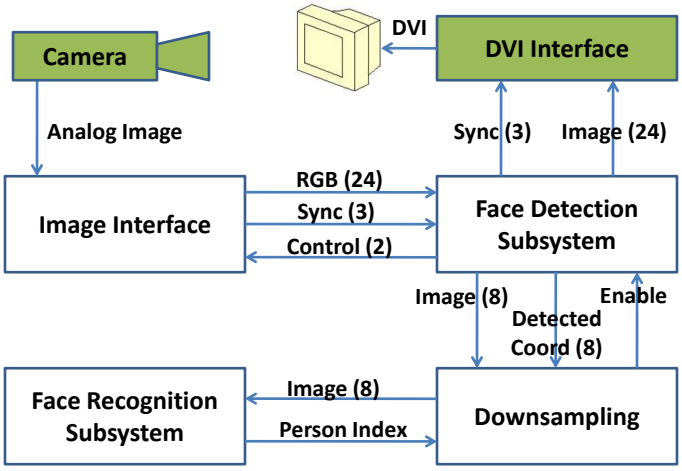


Figure 3.6. The architecture for the complete face recognition system consisting of the face detection and face recognition subsystems.

Table 3.1. Device Utilization Table for the Complete Face Recognition System

Logic Utilization	Used	Available	Utilization
Slices	8,683	11,200	77%
Number of Slice LUTs	32,480	44,800	72%
Number of Block RAMs	84	148	56%
Number of DSP48s	11	128	8%

subsystem is busy. If the face recognition subsystem is available, it reads 20×20 image and returns the index of a person which belongs to the detected face. According to the returned index of a person, we draw a box around the detected face with predefined color. Each individual's face in the set is represented by an index and each index is associated with a color.

The implementation was simulated/verified with ModelSim, and then implemented on a Virtex-5 FPGA. Table 3.1 shows the device utilization of the complete face recognition system on a Virtex-5 FPGA board. According to the experimental results, the complete face recognition system runs at 45 frames per second on VGA data.

3.5 Conclusion

This paper presented the design and implementation of a complete FPGA-based real-time face recognition system which runs at 45 frames per second. This system consists of three subsystems: face detection, downsampling and face recognition. All the modules are designed and implemented on a Virtex-5 FPGA. We presented the architectural integration of the face detection and face recognition subsystems as a complete system on physical hardware. Different experimental results of the face recognition subsystem are presented for pipelined and non-pipelined implementations.

Acknowledgements

This chapter contains materials from a work that was published in International Symposium on Field-Programmable Custom Computing Machines (FCCM), 2011. Matai, Janarbek; Irturk, Ali; Kastner, Ryan. The chapter also contains additional materials that was omitted from the publication due to space constraints. The dissertation author is the primary investigator and author of this work.

Chapter 4

Canonical Huffman Encoding

4.1 Introduction

The main focus of this chapter is to evaluate feasibility of using high-level synthesis for implementing irregular application (kernels) on an FPGA. In the rest of this chapter, we present design and implementation of a complete pipeline of Canonical Huffman Encoding using high-level synthesis. It consists of several irregular kernels. We will discuss design and implementation of these kernels with HLS. Canonical Huffman encoding is one of the main functionalities of lossless data compression engines. Lossless data compression is a key ingredient for efficient data storage, and Huffman coding is amongst the most popular algorithm for variable length coding [69]. Given a set of data symbols and their frequencies of occurrence, Huffman coding generates codewords in a way that assigns shorter codes to more frequent symbols to minimize the average code length. Since it guarantees optimality, Huffman coding has been widely adopted for various applications [53]. In modern multi-stage compression designs, it often functions as a back-end of the system to boost compression performance after a domain-specific front-end as in GZIP [47], JPEG [110], and MP3 [116].

Canonical Huffman coding has two main benefits over traditional Huffman coding. In basic Huffman coding, the encoder passes the complete Huffman tree structure to the

decoder. Therefore, the decoder must traverse the tree to decode every encoded symbol. On the other hand, canonical Huffman coding only transfers the number of bits for each symbol to the decoder, and the decoder reconstructs the codeword for each symbol. This makes the decoder more efficient both in memory usage and computation requirements.

Data centers are one of the biggest users of data encoding for efficient storage and networking, which is typically run on high-end multi-core processors. This trend is changing recently with increased focus on energy efficient and specialized computation in data centers [85]. For example, IBM made a GZIP comparable compression accelerator [86] for their server system. We target the scenario where the previous stage of compressor (e.g., LZ77) produces multi-giga byte throughput with parallelized logic, which requires a high throughput, and ideally energy efficient, data compression engine. For example, to match a 4GB/s throughput, the Huffman encoder must be able to build up 40,000 dynamic Huffman trees per second assuming a generation of a new Huffman tree for every 100KB input data.

The primary goal of this work is to understand the trade-off between performance and power consumption in developing a Canonical Huffman Encoder using high-level synthesis on an FPGA. This is, to the best of our knowledge, the first hardware accelerated implementation of the complete pipeline stages of *Canonical Huffman Encoding (CHE)*. To evaluate generated design from high-level synthesis, we compared the benefits and drawbacks of different computation platforms, e.g., FPGA, low-power processor, and high-end processor. We create highly optimized software implementations targeting an embedded ARM processor and a high-end Intel Core i7 processor. The specific contributions of this paper are:

1. The development of highly optimized software implementations of canonical Huffman encoding.

2. A detailed design space exploration for the hardware accelerated implementations using high-level synthesis tools.

3. A comparison of the performance and power/energy consumption of these hardware and software implementations on a variety of platforms.

The remainder of this paper is organized as follows: Section 4.2 provides algorithmic description of canonical Huffman encoding. In Section 4.3 and Section 4.4, we present detailed hardware and software implementations and optimizations, respectively. In Section 4.5, we present experimental results. Section 4.6 presents related work. We conclude in Section 4.7.

4.2 Canonical Huffman Encoding (CHE)

In basic Huffman coding, the decoder decompresses the data by traversing the Huffman tree from the root until it hits the leaf node. This has two major drawbacks: it requires storing the entire Huffman tree which increases memory usage. Furthermore, traversing the tree for each symbol is computationally expensive. CHE addresses these two issues by creating codes using a standardized format.

Figure 4.1 shows the CHE process. The *Filter* module only passes symbols with non-zero frequencies. Then the encoder creates a Huffman tree in the same way as the basic Huffman encoding. The *Sort* module rearranges the symbols in ascending order based upon their frequencies. Next, the *Create Tree* module builds the Huffman tree using three steps: 1) it uses the two minimum frequent nodes as an initial sub-tree and generates a new parent node by adding their frequencies; 2) it adds the new intermediate node to the list and sorts them again; and 3) it selects the two minimum elements from the list and repeat these steps until one element remains. As a result, we get a Huffman tree, and by labelling each left and right edge to 0 and 1, we create codewords for symbols.

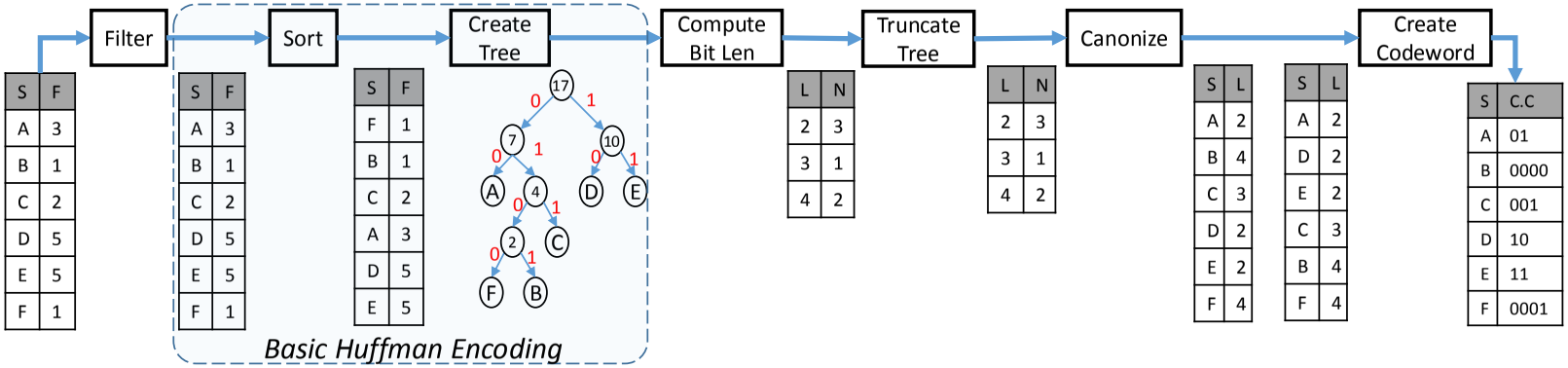


Figure 4.1. The symbols are filtered and sorted, and used to build a Huffman tree. Instead of passing the entire tree to the decoder (as is done in "basic" Huffman coding), the encoding is done such that only the length of the symbols in the tree is required by the decoder.

For example, the codeword for A is 00 and codeword for B is 0101. This completes the basic Huffman encoding process. The CHE only sends the length of each Huffman codeword, but requires additional computation as explained in the following.

The *Compute Bit Len* module calculates the bit lengths of each codeword. It saves this information to a list where the key is length and value is the number of codewords with that length. In the example case, we have 3 symbols (A,D,E) with the code length of 2. Therefore, the output list contains L=2 and N=3. The *Truncate Tree* module rebalances the Huffman tree when it is very tall and/or unbalanced. This improves decoder speed at the cost of a slight increase in encoding time. We set the maximum height of the tree to 27.

Using output from the *Truncate Tree* module, the *Canonize* module creates two sorted lists. The first list contains symbols and frequencies sorted by symbol. The second list contains symbols and frequencies sorted by frequency. These lists are used for faster creation of the canonical Huffman codewords.

The *Create Codeword* module creates uniquely decodable codewords based on the following rules: 1) Shorter length codes have a higher numeric value than the same length prefix of longer codes. 2) Codes with the same length increase by one as the symbol value increases. According to the second rule, codes with same length increase by one. This means if we know the starting symbol for each code length, we can construct the canonical Huffman code in one pass. One way to calculate the starting canonical code for each code length is as follows: $for\ l = K\ to\ 1; Start[l] := [Start[l+1] + N[l+1]]$ where $Start[l]$ is the starting canonical codeword for a length l, K is the number of different code lengths, and $N[l]$ is the number of symbols with length l. In CHE, the first codeword for the symbol with the longest bit length starts all zeros. Therefore, the symbol B is the first symbol with longest codeword so it is assigned 0000. The next symbol with length 4 is F and is assigned 0001 by the second rule. The starting symbol

for the next code length (next code length is 3) is calculated based on the first rule and increases by one for the rest.

In this paper, after calculating codewords, we do a bit reverse of the codeword. This is a requirement of the application on hand, and we skip the details due to space constraints.

The CHE pipeline includes many complex and inherently sequential computations. For example, the *Create Tree* module needs to track the correct order of the created sub trees, requiring careful memory management. Additionally, there is very limited parallelism that can be exploited. We designed the hardware using a high-level synthesis tool, and created highly optimized software for ARM and Intel Core i7 processors. In the following sections, we will report results, and highlight the benefits and pitfalls of each approach. We first discuss the hardware architecture and the implementation of the CHE design using HLS. Then we present the optimized software design of the CHE.

4.3 Hardware Implementations

We created HLS architectures with different goals. *Latency Optimized* is designed to improve latency by parallelizing the computation in each module, and *Throughput Optimized* targets a high throughput design by exploiting task level parallelism. Since their block diagrams are very similar, we only present the block diagram of the *Throughput Optimized* architecture as shown in Figure 4.2. For the sake of simplicity, it only shows the interfaces with block rams (BRAMs). To create these designs (*Latency Optimized*, and *Throughput Optimized*), we start from a software C code which we name a *Baseline* design. Then we restructure parts of the code (*Restructured* design) as discussed below targeting efficient hardware architectures.

The input to the system is a list of symbols and frequencies stored in Symbol-Frequency (*SF*) BRAM. The size of *SF* is $48 \times n$ bits where 16 bits are used for symbol,

32 bits are used for frequency, and n is the number of elements in the list. The *Filter* module reads from the *SF* BRAM and writes the output to the next *SF* BRAM. Also it passes the number of non-zero elements to the *Sort* module. The *Sort* module writes the list sorted by frequency into two different *SF* BRAMs. Using the sorted list, the *Create Tree* module creates a Huffman tree and stores it into three BRAMs (*Parent Address*, *Left*, and *Right*). Using the Huffman tree information, the *Compute Bit Len* module calculates the bit length of each symbol and stores this information to a *Bit Len* BRAM. We set the maximum number of entries to 64, covering up to maximum 64-bit frequency number, which is sufficient for most applications given that our Huffman tree creation rebalances its height. The *Truncate Tree* module rebalances the tree height and copies the bit length information of each codeword into two different BRAMs with the size of 27, which is the maximum depth of the tree. The *Canonize* module walks through each symbol from the *Sort* module and assigns the appropriate bit length using the *BitLen* of each symbol. The output of the *Canonize* module is a list of pairs where list contains symbols and its bit lengths.

We implemented the algorithm on an FPGA using the Vivado High-level Synthesis (HLS) tool. The *Baseline* design has no optimizations. We developed a *Restructured* design on top of the *Baseline* design. After creating the restructured design, we optimize the restructured design for latency and throughput. To implement above hardware designs, we first profiled the algorithm on an ARM with different optimizations. Figure 4.6 shows initial (naive) running time of each modules of the design on an ARM processor. Among these, *Radix Sort*, *Create Tree*, *Compute Bit Length* are most computationally intensive. We focused our design space exploration on these sub modules and optimized them in HLS to generate an efficient design.

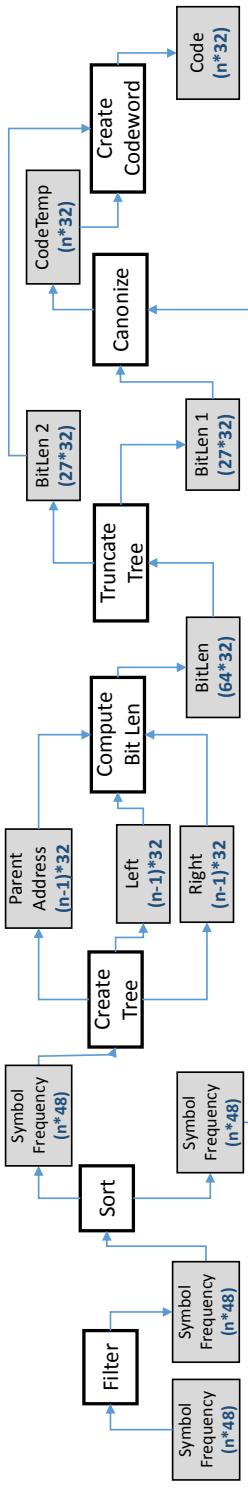


Figure 4.2. The block diagram for our hardware implementation of canonical Huffman encoding. The gray blocks represent BRAMs with its size in bits. The white blocks correspond to the computational cores.

4.3.1 Radix Sort

The radix sorting algorithm arranges the input data for each radix from left to right (least significant digit) or right to left (most significant digit) in a stable manner. In a decimal system, the radix takes values from 0 to 9. In our system, we are sorting the frequencies, which are represented using a 32-bit number. We treat a 32-bit number as 4-digit number with radix $r = 2^{32/4} = 2^8 = 256$. In serial radix sort, the input data is sorted by each radix k times where k is radix ($k = 4$ in our case). Algorithm 4 describes the radix sorting algorithm which used counting sort to perform the individual radix sorts.

Algorithm 4: Counting sort

```
1  Procedure CountingSort()
       Data: A[]
       /* A[] input array                                  */
       /* N is input array size                            */
       /* k is number of different radix                   */
       Result: Out
2      HISTOGRAM-KEY:
3      forall the i ← 0 to k do
4          Bucket[i] ← 0
5          temp[j] ← A[j]
6      end
7      PREFIX-SUM:
8      First[0] ← 0
9      forall the i ← 1 to k do
10         First[i] ← Bucket[i − 1] + First[i − 1]
11     end
12     COPY-OUTPUT:
13     forall the j ← 0 to N − 1 do
14         i ← A[j]
15         Out[First[i]] ← temp[j]
16         First[i] ← First[i] + 1
17     end
```

In order to implement parallel radix sort, we made two architectural modifications to the serial algorithm. First, we pipelined the counting sort portions (there are four

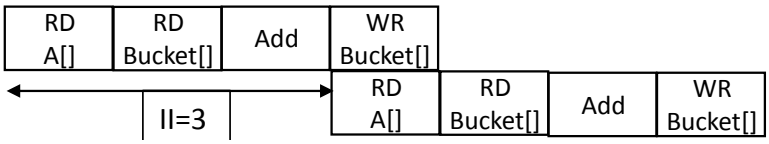


Figure 4.3. A naively optimized code has RAW dependencies which requires an $II = 3$.

counting sorts in the algorithm). This exploits coarse grained parallelism among these four stages of the radix sort architecture using the dataflow pipelining pragma in the Vivado HLS tool. Next we optimized the individual counting sort portions of the algorithm. In the current counting sort implementation there is a histogram calculation (Algorithm 4 line number 6). When implemented with an HLS tool, this translates to an architecture which is similar to Figure 4.3. With this code, we achieve an initiation interval (II) equal to 3 due to RAW dependencies. Ideally we want an $II = 1$. Since the histogram calculation is in a loop, achieving an $II = 1$ boasts performance by orders of magnitude. Achieving $II = 1$ requires an additional accumulator, which is shown in the pseudo HLS code in Listing 4.1. If the current and previous values of the histogram are the same, we increment the accumulator; otherwise we save the accumulator value to previous value's location and start a new accumulator for the current value. In addition to that, using dependency pragma, we instruct HLS to ignore RAW dependency.

```
1 #pragma DEPENDENCE var=Bucket RAW false
2 val = radix; //A[j]
3 if(old_val==val){
4   accu = accu + 1 ;
5 }
6 else {
7   Bucket[old_val] = accu;
8   accu = Bucket[val]+1;
9 }
10 old_val =val;
```

Listing 4.1. An efficient histogram calculation with II of 1.

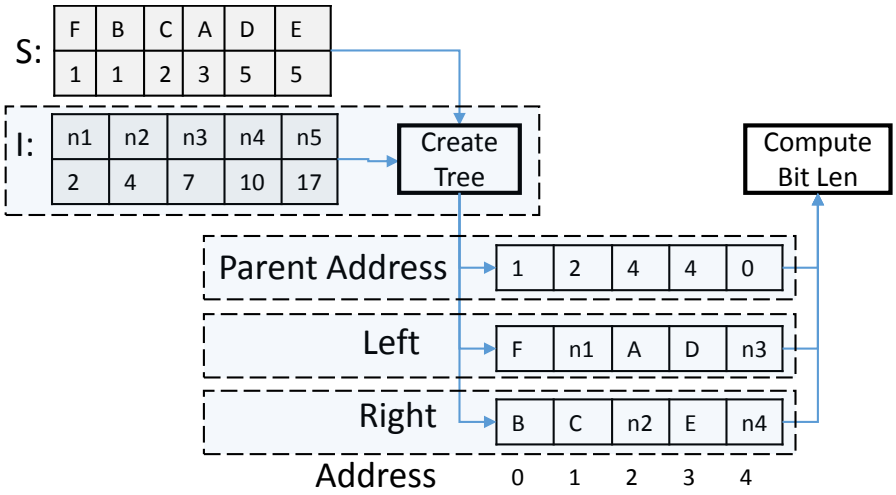


Figure 4.4. The architecture for efficient Huffman tree creation. This architecture creates a Huffman tree in one pass by avoiding resorting of the elements.

4.3.2 Huffman Tree Creation

In order to create the Huffman tree, the basic algorithm creates a sub tree of the two elements with minimum frequency and adds intermediate node whose frequency is the sum of $f1$ and $f2$ to the list in sorted order ($f1$ and $f2$ are frequencies of those selected elements). This requires re-sorting the list each time when we add a new element to the list. At each step we remove the two elements with minimum frequencies and insert a new node with the aggregated frequency of those selected nodes. This means that the generated nodes are produced in non-decreasing sequence order. Thus, instead of adding the intermediate node to the sorted list, we used another BRAM to store the intermediate nodes in a FIFO.

With this modification, we eliminate the process of resorting. The architecture is shown in Figure 5.19. The S queue stores the input symbol/frequency list. The I Queue stores the intermediate nodes. The size of S is n, and size of I is $n-1$. *Create Tree* stores tree information on *Left*, *Right*, and *Parent Address* BRAMs. The changed algorithm works as follows. Initially, the algorithm selects the two minimum elements from the

S queue in a similar manner to basic Huffman encoding. Then the algorithm adds the intermediate node $n1$ to the *I* queue. It selects a new element $e1$ from *S* queue. If the frequency of $e1$ is smaller than frequency of $n1$, we make $e1$ the left child. Otherwise, we make $n1$ the right child. If the *I* queue is empty (after selecting the left child), we select another element $e2$ from *S* and make it the right child. Then we add their frequency values, make a new intermediate node $n2$, and add it to *I*. This process continues until there is no element left in the *S* queue. If there are elements in the *I* queue, we create sub trees by making the first element the left child and second element the right child.

This eliminates need for resorting by using additional BRAM. While we are constructing the Huffman tree with this method, we store the tree information on three different BRAMs. The *Left* and *Right* BRAMs store the left and right children of each sub tree. The first left/right child is stored on address 0 of Left/Right. The *Parent Address* BRAM has the same address as its children but stores an address of the parent of that location. It points to the parent address of its children.

4.3.3 Parallel Bit Lengths Calculation

After storing tree information on *Left*, *Right*, and *Parent Address* BRAMs, calculating the bit length for each code is straightforward. The *Compute Bit Len* function starts from address 0 of *Left* and *Right* BRAMs and tracks their parents location from the *Parent Address* BRAM. For example, *B* and *F* have the same parent since they are both in address 0 in respective BRAMs. The address of the parent of *B* and *F* is located at 1 which is stored in the *Parent Address* BRAM at address 0. From address 1, we can locate the grandparent of *F* and *B* at address 2. From address 2, we can locate next ancestor of *B* and *F* at address 4. When we check address 4 and we find out it is zero, that means we have reached the root node. Therefore, bit length of *F* and *B* is equal to 4.

The data structures (*Left*, *Right*, and *Parent Address*) allow efficient and parallel

bit length calculation. In these data structures, the symbols are stored from left to right, and we can track any symbol's parents to root starting from that symbol's position. In our design, we exploit this property and initiated parallel processes working from different symbols (Huffman tree leaf nodes) towards the root node to calculate bit lengths of symbols in parallel. Figure 4.5 shows an example where two processes are working to calculate the bit lengths in parallel. Each process operates on data in its region, e.g., *Process 2* only needs data for symbol *D* and *E* symbols.

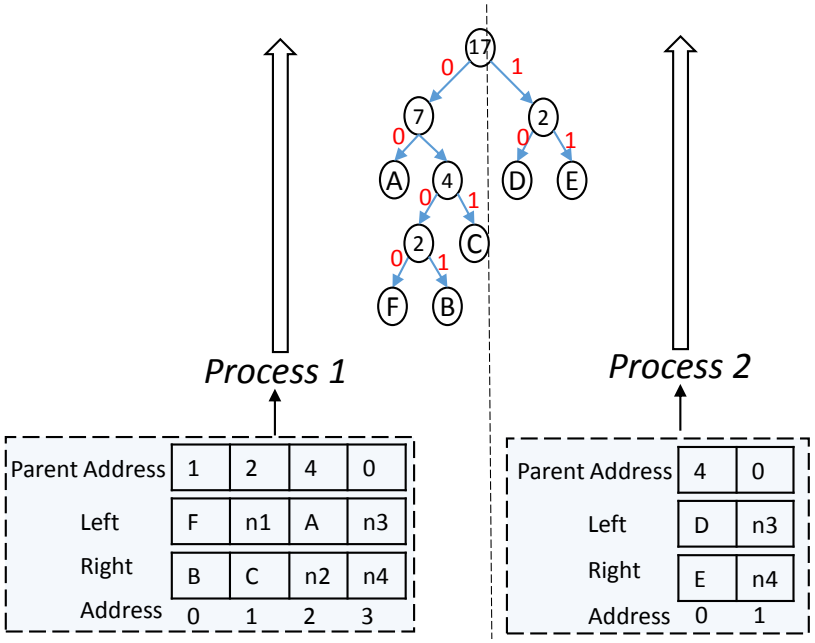


Figure 4.5. A Parallel Bit Lengths Calculation. Each process has its own set of data which allows for fully parallel bit length calculation.

4.3.4 Canonization

In *Canonize*, the algorithm has to create two lists; one sorted by symbols and another sorted by frequencies. In our design, we changed the *Canonize* module by eliminating the list which is sorted by frequencies. The second list is needed by *CreateCodeword* to track the number of codewords with the same length. We can track

the number of symbols having the same code length by using 27 counters since any codeword need at most 27 bits. This optimization efficiently reduces the running time of the *Canonize* by half. Therefore, output of the *Canonize* is only one list in our design. However, it slightly increases the running time of *CreateCodeword*.

4.3.5 Codeword Creation

In addition to that, using dependency pragma, we instruct HLS to ignore RAW dependency. In the *Create Codeword* module, the algorithm does the bit reverse of each code length. Code lengths can be up to 27 bits. The software bit reverse does not synthesis to efficient hardware. Therefore, we coded an efficient bit reverse in HLS that results in logic as good as a custom bit reverse using the coding technique given in [136]. Listing 4.2 shows an example bit reverse for 27 bit number. Bit lengths can be up to 27-bits requiring to write twenty seven of these functions in our design. Since these functions are implemented efficiently in HLS, we inline them in our design which increases performance with a slight increase of area.

```
1 #pragma pipeline
2 for(int k = 0; k < 27; k++) {
3   j = (j << 1) | ((i >> k) & 0x1);
4 }
```

Listing 4.2. Efficient bit reverse for 27-bit number

Restructured Design: This design includes manual restructured and the optimized designs for *Radix Sort*, *Create Tree*, *Compute Bit Length*, *Canonize*, and *Create Codeword* modules, which were described earlier in this section.

Latency Optimized Design: On top of the *Restructured* design, we pipeline computations in individual modules using the *pipeline pragma* in the high-level synthesis tool. The *pipeline pragma* pipelines the computations in a region exploiting fine grained parallelism. Once restructured, rest of the computations in individual modules of the

CHE are sequential. e.g., computations in iterations execute dependent *read, compute, write* operations in each iteration of loop. This allows pipelining of only these primitive operations in each iteration of the loop. This is done by pipelining the most inner loop of each module.

Throughput Optimized Design: This design further optimizes the *Latency Optimized* design to achieve coarse grained parallelism. The goal of this design is to improve throughput by exploiting coarse grained parallelism among the tasks (through task level pipelining). We achieve task level pipeline in the high-level synthesis by using a *dataflow* directive. However, the current *dataflow* directive only works if the input/output of functions are read/written by only one process. We solved this issue by duplicating the input/outputs which are read/written by more than one process. Listing 4.3 shows the pseudocode. For example, the output of *Sort* is read by two processes (*Create Tree* and *Canonoize*). Therefore, we duplicated the output of *Sort* into two different arrays (BRAMs: *SF_SORT1, SF_SORT2*) inside the *Sort* module. This is shown in Listing 4.3 Line 9. For simplicity, we omitted BRAM duplication parts for the rest of the code in Listing 4.3. This incurs additional logic to duplicate this data as shown in Listing 4.4. This has some adverse effect on the final latency, but it improves the overall throughput.

```
1  CanonicalHuffman(SF[SIZE], Code[SIZE]){
2  #pragma dataflow
3  SF_TEMP1[SIZE];
4
5  SF_SORT1[SIZE];
6  SF_SORT2[SIZE];
7
8  Filter(SF, SF_TEMP1);
9  Sort(SF_TEMP1, SF_SORT1, SF_SORT2);
10 CreateTree(SF_SORT1, PA, L, R);
11
12 //Separate data in PA,L, R
13 //into PA1, L1, R1, PA2, L2, R2
14
15 //Parallel bit lenght calculation
16 ComputeBitLen1(PA1, L1, R1, Bitlen1);
```

```
17 ComputeBitLen1(PA2, L2, R2, Bitlen1);
18
19 //Merge BitLen1 and BitLen2 to BitLenFinal
20
21 TruncateTree(BitlenFinal, Bitlen3, Bitlen4);
22 Canonize(Bitlen4, SF_SORT2, CodeTemp);
23 CreateCodeword(Bitlen4, CodeTemp, Code);
24 }
```

Listing 4.3. Pseudo-code for the throughput optimized design.

```
1 Sort(SF_TEMP1, SF_SORT1, SF_SORT2){
2 //Sort logic
3 SF_SORTED = ... ;
4 //Additional logic
5   for(int i=0;i<n;i++){
6     SF_SORT1[i] = SF_SORTED[i];
7     SF_SORT2[i] = SF_SORTED[i];
8   }
9 }
```

Listing 4.4. Additional logic to support task level parallelism.

4.4 Software Implementations

The initial *Naive* software implementation is a functionally correct design which is not optimized for efficient memory management. In *Baseline* design, we used efficient memory management through optimized data structures to optimize the naive implementation. An example optimization is using pointers efficiently instead of arrays whenever possible. The code has the same functionality as the code in HLS. In addition to the memory optimization using pointers, we did following optimizations.

Software Optimization (SO)

We do the same optimization for the *Canonize* and *Create Codeword* functions as we did in HLS implementation. This cuts the running time of *Canonize* by almost half.

Compiler Settings (CS)

We do compiler optimizations on top of software optimization using -O3 compiler flag. These compiler optimization levels do common compiler optimizations such as common sub expression elimination and constant propagation. On top of that, we did manual loop vectorization and loop unrolling whenever it gives better performance.

4.5 Experimental Results

In this section, we present the performance, area, power and energy consumption results of our canonical Huffman encoding implementations for Xilinx Zynq FPGA, ARM Cortex-A9, and Intel Core i7 processors. All systems are tested with 256 and 704 symbol frequencies (as dictated by the of former LZ77 stage) as well as 536 as a median value in order to show scalability trends. The latency is reported in microseconds μs and throughput is reported in number of canonical Huffman encodings performed per second. In this work, Vivado HLS 2013.4 is used for the hardware implementations. The final optimized designs are implemented on a Xilinx Zynq FPGA (xc7z020clg484-1) and the functionality is tested using real-world inputs.

4.5.1 Software Implementations

ARM Cortex-A9: We used ARM Cortex-A9 667MHz dual-core processor with 32/32 KB Instruction/Data Cache and 512 Kbyte L2 cache. In order to get highest performance, we run our design on the processor without using an operating system. The latency results are calculated using the number of clock cycles times the clock period (1.49925 ns = 667 MHz). The number of clock cycles are collected using CP15 performance monitoring unit registers of ARM processor.

Figure 4.6 presents performance of software implementations running on the ARM processor. The initial *Naive* design is implemented without any optimizations. The

Baseline design is optimized on top of the *Naive* design using efficient data structures and memory management on the ARM processor. On top of *Baseline* design then we apply Software Optimization (SO) and Compiler Setting (CS) in that order. In *SO* design, the run time of the *Canonize* module is reduced by 2.2X (114 us to 53 us). This added little overhead on the *CreateCodeword* module by increasing its run time from 75 us to 103 us. Overall, the running time of *SO* design is decreased from 622 us to 589 us. In the final design (*SO+CS*), the -O3 optimization decreases the running time of all modules, resulting in total running time of the design being decreased from 589 us to 220 us.

Intel Core i7: We also implemented the same software optimizations on a multi-core CPU implementation - an Intel Core i7-3820 CPU running at 3.6 GHz. Figure 4.7 presents performance of the various software implementations (*Naive*, *Baseline*, *SO* and *CS*). Due to fast running time of the algorithm on a powerful processor, the software optimization (SO) has very little impact on the final running time giving only 2 us of saving of total running time. The final optimized with *SO+CS* has the fastest running time.

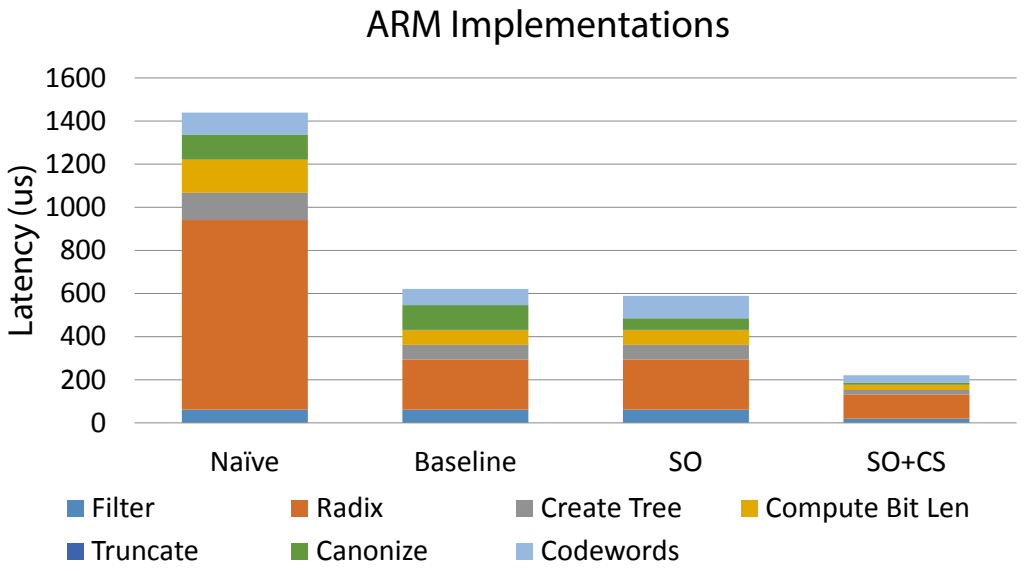


Figure 4.6. The latency results of the various software implementations running on an ARM Cortex-A9 processor.

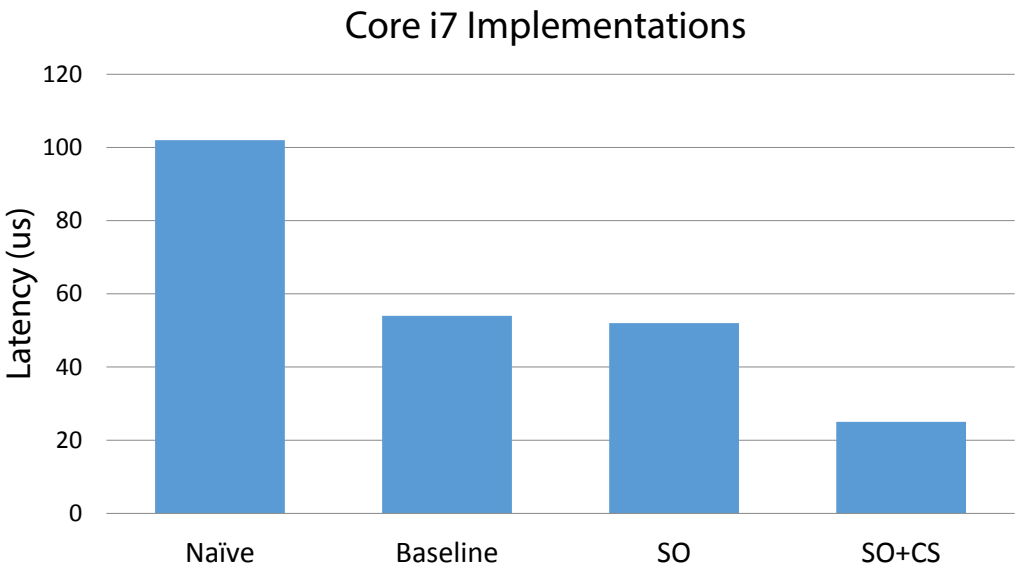


Figure 4.7. The latency results of the various software implementations running on an Intel Core i7 processor.

4.5.2 Hardware Implementations

We implemented various hardware described in Section 4.3 (Baseline, Restructured, Latency Optimized, and Throughput Optimized). Table 5.4 gives the area utilizations and performance results for these different designs when using 704 input size (536 non zero elements), and Figure 4.8 shows their latency and throughput results.

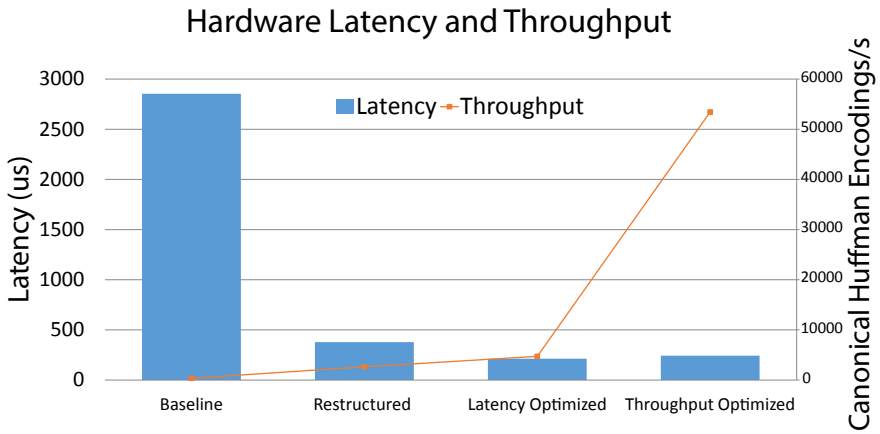


Figure 4.8. The latency and throughput of hardware implementations.

Table 4.1. Hardware area and performance results.

	Area		Performance		
	Slices	BRAM	Clock Cycles	II	Frequency
Baseline	2067	15	130977	130978	45
Restructured	761	14	90336	90337	173
Latency Optimized	1159	14	33769	33770	133
Throughput Optimized	1836	62	41321	3186	170

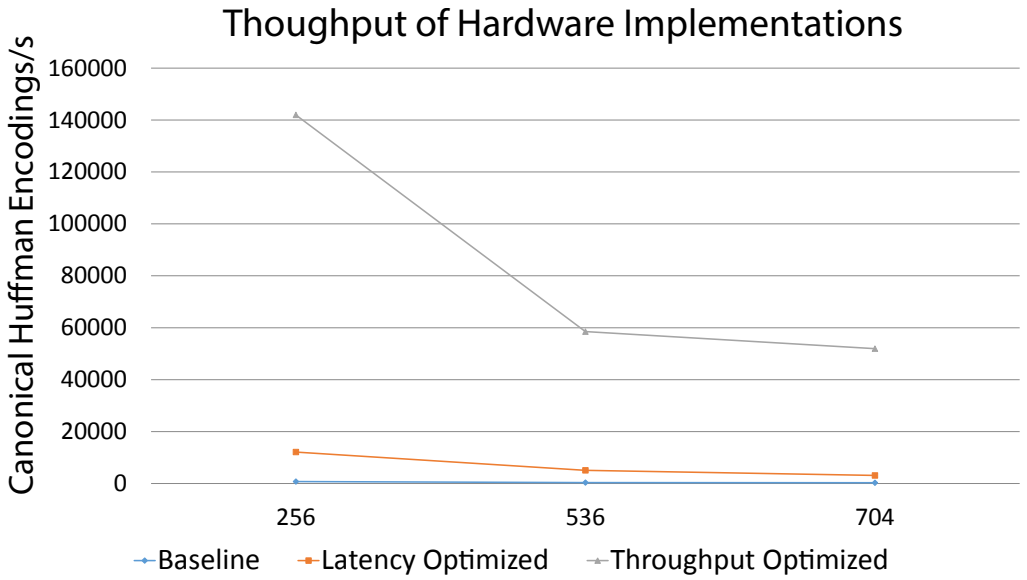


Figure 4.9. The throughput of hardware implementation for different input sizes.

The number of slices in the *Baseline* design is reduced in *Restructured* design due to writing better HLS friendly code. (e.g., optimized bit reverse module). The *Latency Optimized* design has higher slices and higher performance due to fine grained parallelism. The *Throughput Optimized* designs use more BRAMs due to duplication of the input/output data for the purpose of overcoming limitations of *dataflow directive* in the HLS tool (each module can only have one read and write operation per port). The *Throughput Optimized* also has higher throughput due to the additional resources required to take advantage of both coarse grained and fine grained parallelism.

The latency $(clock\_cycles \times clock\_period)$ measures the time to perform one

canonical Huffman encoding operation. The throughput $(Initiation\_Interval(II) \times clock\_period)$ is the number of canonical Huffman encoding operations per second. In the case of pipelining, the latency and throughput operations may not be equivalent. The latency reduces from *Baseline* design to *Latency Optimized* due to restructuring and pipelining. The latency of *Throughput Optimized* design increases from 212 μs to 242 μs (largely due to 33769 and 41321 clock cycles, respectively, though the clock period is also larger).

However, this *Throughput Optimized* design has better throughput than previous designs. The *Throughput Optimized* accepts new data every 3186 clock cycles while *Latency Optimized* design accepts new data in every 33770 clock cycles. Therefore, *Throughput Optimized* has higher throughput than the *Latency Optimized* design. Figure 4.9 shows the throughput for three designs for non-zero input sizes 256, 536 and 704. The y-axis shows the throughput (the number of canonical Huffman encodings per second) and x-axis shows input sizes.

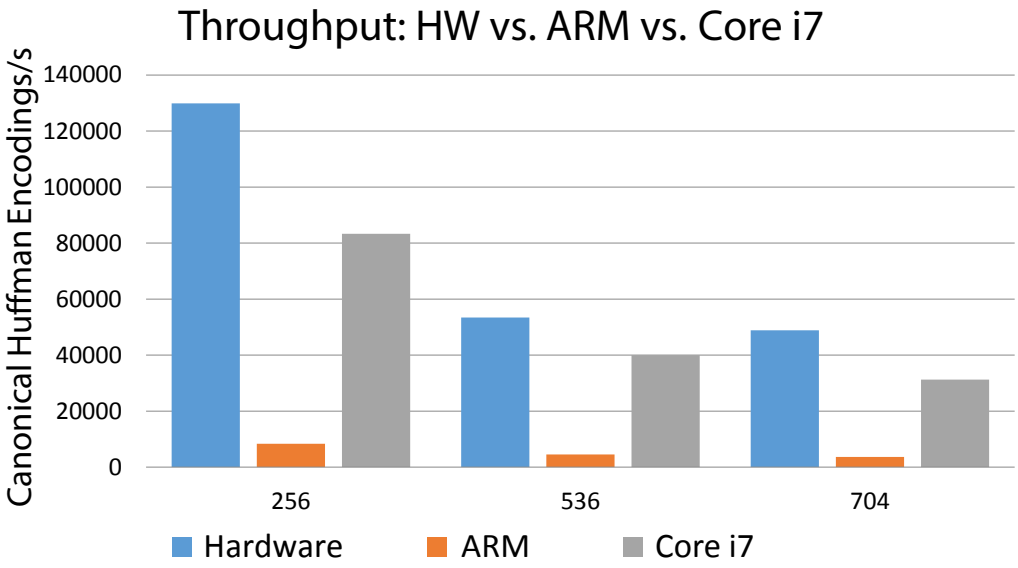


Figure 4.10. Throughput for different input sizes: HW vs. ARM vs. Core i7

4.5.3 Hardware and Software Comparison

In this section, we discuss efficiency of our various hardware and software implementations in terms of four aspects: performance, power/energy consumption, productivity, and the benefits and drawbacks of designing using HLS tools as compared to directly coding RTL.

Performance: Figure 4.10 compares the throughputs of the best Xilinx Zynq FPGA design, ARM Cortex-A9 design, and Intel Core i7 design for three different input sizes. Overall, the hardware implementation has 13-16X better throughput than the ARM design due to exploiting task level parallelism. The hardware design achieves 1.5-1.8X speed-up over highly optimized design on a Intel Core i7 processor.

Power and Energy Consumption: The primary focus of this paper is to understand the energy efficiency of canonical Huffman encoding on different hardware and software platforms. We measured the ARM and FPGA portions of Zynq device power consumptions using power domains VCCPINT and VCCINT as described [12] in real time. The power consumption of ARM design is measured by running the design as a standalone application (i.e., without an operating system). The ARM implementation consumes around 344 mWatt in real time. The power consumption of FPGA part is measured in two ways; the first measurement is obtained by using Xpower tool which estimates around 380 mWatts. Then we calculated the power consumption of our design from real time *voltage* and *current* by running our design on a Zynq chip. Our design consumes maximum 170 mWatts among 250 runs. The Intel Core i7 power when running the canonical Huffman encoding is measured using Intel Power Gadget tool-kit in a real time[4]. The Core i7 consumes between 27-30 Watts when running the software.

Figure 4.11 (170 mWatt is used as FPGA power) shows the energy consumption of the different platforms. This is measured in the number of canonical Huffman encodings

performed per Watt for the FPGA, ARM, and Core i7 designs. The FPGA is the most energy efficient. For three different sizes, hardware implementation has around 230X more encodings per Watt than the Core i7, and the ARM implementation has around 9X more CHEs than Core i7 design.

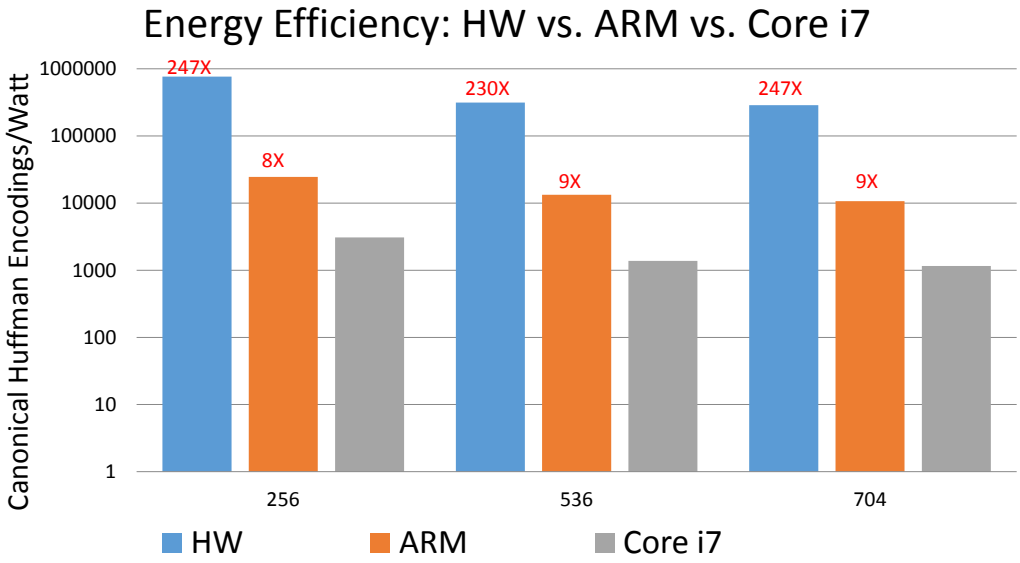


Figure 4.11. Power efficiency for various input sizes: HW vs. ARM vs. Core i7

Productivity: We spent about one month designing the algorithm on an FPGA using the Xilinx Vivado HLS tool. Approximately 25% of the time was spent on learning the algorithm and initial planning which also includes writing a C code for Core i7 and ARM processors. The rest of the time (approximately 75% of the total time) was spent on designing optimized FPGA solution using HLS. This includes time spent to create the Baseline, Restructured Latency Optimized, and Throughput Optimized implementations. Our final hardware design is parametrizable for different inputs which allows easy design space exploration in a short time while achieving higher throughput with significantly less energy consumption than the Intel and ARM processors.

HLS vs RTL: An obvious next step would be to implement the best hardware implementation using RTL to obtain even better results. This would likely increase the

clock frequency, and provide better throughput and energy consumption. However, this is an expensive proposition in terms of designer time. For example, the tree creation/bit length calculation modules require careful coding if one decides to design them in RTL. HLS provides good results because limited parallelism in individual modules can be easily exploited by pipelining most inner loops in C, and that would be the primarily optimization target that one would exploit when writing RTL. Certainly, HLS provides a huge benefit in terms of design space exploration, and we advise first using the HLS tools to find a suitable micro architecture, and then develop the RTL from that micro architecture in order to optimize it further.

4.6 Related Work

Many previous works [17, 116, 75] focus on a hardware implementation for Huffman *decoding* because it is frequently used in mobile devices with tight energy budget, and the decoding algorithm has high levels of parallelism. On the other hand, Huffman *encoding* is naturally sequential, and it is difficult to parallelize in hardware. However, as we show in this paper, this does not mean that it is not beneficial to pursue hardware acceleration for this application; when carefully designed, a hardware accelerator can be implemented in a high throughput and energy efficient manner.

There have been several hardware accelerated encoder implementations both in the ASIC and FPGA domain to achieve higher throughput for real-time applications [81, 97, 106, 112, 119]. Some designs [81, 97, 106, 119] focus on efficient memory architectures for accessing the Huffman tree in the context of JPEG image encoding and decoding. In some applications, the Huffman table is provided. In such cases the designs focused on the stream replacement process of encoding.

Some previous work develops a complete Huffman coding and decoding engine. For example, the work by Rigler et al [112] implements Huffman code generation for

the GZIP algorithm on an FPGA. They employed a parent and the depth RAM to track the parent and depth of each node in tree creation. Recent work by IBM [86] designed GZIP compression/decompression streaming engine where they only implemented static Huffman encoder. This significantly lowers the quality of recompression.

This paper focuses on canonical Huffman encoding since our data center scenario must adapt the frequency distribution of new input patterns under the context of general data compression combining with Lempel-Ziv 77 [138]. This is well suited to canonical Huffman en coding. We provide comparison of a number of optimized hardware and software implementations on a variety of platforms and target energy efficiency.

4.7 Conclusion

One may assume this application is not suitable for hardware implementation because the algorithm is inherently sequential and has many irregular kernels. On the contrary, the hardware implementation produces a superior result both in terms of throughput and energy efficiency. To demonstrate these, we developed a number of performance and low power design and implementation of canonical Huffman encoding in both hardware and software running on the Virtex Zynq FPGA, ARM Cortex-A9, and Intel Core i7 platforms. We demonstrated several optimization techniques for complex and inherently sequential irregular applications (hard to code by hand due to complexity and sequentiality) such as Huffman tree creation can be easily done in high-level synthesis in a short amount of time with high performance and low power. Our final design has around 13-16X times higher throughput than highly optimized design on ARM and it is more energy efficient than design on high end multi-core CPU. We designed the systems on an FPGA and verified the functionality on a Zynq device. We briefly summarize the main lessons learned: Expectedly, applications that contain irregular kernels are hard to synthesis with modern high-level synthesis tools than applications that purely contain

regular kernels. This difficulty rooted from the fact that irregular kernels have properties such as indirect memory access, pointer chasing that are hard for modern high-level synthesis tools to parallelize. One solution to this problem is finding the irregular kernel bottlenecks and making them high-level synthesis friendly by restructuring the portion of the kernel that causes problem. In high-level synthesis design, it is always required code restructuring for synthesizing irregular kernels efficiently (e.g., Huffman tree creation). It is often involves completely re-writing (for restructuring purposes) the whole kernel or application from scratch. While completely designing a new application reduces the benefit of modern high-level synthesis , it allows us to build a database of build re-usable templates for common irregular kernels in high-level synthesis. In next chapter, we will formally discuss some representative examples of code restructuring.

Acknowledgements

This chapter contains materials from a work that was published in The 25th IEEE International Conference on Application-specific Systems, Architectures and Processors (ASAP), 2014. Matai, Janarbek; Kim, Joo-Young; Kastner, Ryan. The dissertation author is the primary investigator and author of this work.

Chapter 5

Restructured HLS Code

5.1 Introduction

Recent state-of-the-art HLS tools reduce the effort required to design FPGA applications. These tools eliminate the need to design an architecture that specifies every excruciating detail of what occurs on a cycle by cycle basis. In addition, HLS substantially reduces the amount of code that must be written for a given application, and in theory enable any software designer to implement an application on an FPGA.

However, in order to get a good quality of result (QoR), designers must write *restructured code*, i.e., code that maps well into hardware, and often represents the eccentricities of the tool chain, which requires deep understanding of micro-architectural constructs. As noted in previous chapters, "standard", off-the-shelf code typically yields very poor QoR that are orders of magnitude slower than CPU designs, even when using HLS optimizations [82, 87]. We presented the impact of code restructuring in Chapter 2 and presented comprehensive examples of code restructuring in Chapter 4.

In this chapter, we focus on restructured code and present further study about the impact of restructured code on the final generated hardware from high-level synthesis. We start with a formal definition of restructured code and then we define regular and irregular kernels in the context of high-level synthesis. Then we present selected common

restructured coding techniques on different kernels and applications.

Restructured code is necessary to create an efficient hardware implementation; it typically differs substantially from a software implementation - even one that is highly optimized. Recent studies suggest that restructuring code is an essential step to generate an efficient FPGA design [41, 36, 88, 87, 80]. This means that in order to get efficient designs from HLS tools, a user must write restructured code with the underlying hardware architecture in mind. Writing restructured code requires significant hardware design expertise, and domain specific knowledge. The difficulty of writing restructured code limits the FPGA application development to a select number of designers. In other words, if we wish to expand FPGA design to a larger number of people, we must make HLS tools more friendly to those that do not have substantial amount of hardware design knowledge.

In the following, we demonstrate how to restructure code regular kernels and irregular kernels. We define these terms in the context of high-level synthesis.

A *regular kernel* has one of the following characteristics:

1. The kernel is contains loops which process data using a fixed number of iterations.
2. The kernel does not contains indirect memory accesses, e.g., A[B[i]].
3. A kernel outputs its final result (i.e., it finishes execution) when the last loop or (last instruction) finishes its execution. That is, the termination of kernel does not depend on undefined behavior.

An *irregular kernel* has one or more of the following characteristics:

1. The kernel is inherently sequential, i.e., the execution of current instruction depends on execution result of previous instruction (e.g., A[i]= ...; B[i] = ... A[i] ...; etc.).

2. If kernel contains an assignment that accesses memory via an indexed variable, and memory access pattern is not predictable (e.g., `A[B[i]]`).

3. Kernel contains loops where loop condition contains variables or arrays that are calculated inside of a loop and have an unpredictable pattern. For example, in this code `array[]` is used as part of the loop condition, and its value is modified inside the loop. `while ((j > 0) && (array[j-1] > index)) { array[j] = array[j-1]; j = j - 1; }`

Previous works shows that *irregular* programs are more difficult to optimize in high-level synthesis than *regular* programs [34, 126]. We examine the code reconstructing techniques for both types of programs using different examples. While regular kernels are easier to optimize in HLS when the size of input is small, they are difficult to optimize as the input size grows. We present regular kernels examples (e.g., matrix multiplication, and integral image calculation) and irregular kernels (e.g., histogram, and Huffman tree) that must be restructured in order to generate efficient hardware using high-level synthesis. In the remainder of this chapter, we discuss the impact of restructured code on the final hardware generated from high-level synthesis. And we propose a restructuring techniques based on best practices. The specific contributions of this chapter include:

1. A study on the importance of restructuring code to obtain FPGA designs with good QoR (Quality of Result). We define QoR as hardware that has high performance and low area usage.

2. Eight common code restructuring techniques and their impact on QoR for regular and irregular kernels.

3. A list of challenges and possible solutions for developing a novel tool flow that opens up FPGA design to a broader set of programmers.

The remainder of this chapter is organized as follows: Section 5.2 describes restructured code using eight examples that demonstrate the impact of code restructuring. We start with small examples (prefix sum, histogram, and SpMV). Then we describe how to restructure larger kernels using matrix multiplication, FFT, Huffman tree creation, convolution, and face detection as examples. For the larger kernels, we also present the impact of code restructuring by providing performance and area results of restructured and non-restructured codes. Section 5.3 presents a study about impact of using existing restructured code to make FPGA designs easier for software programmers. Section 5.4 presents a list of challenges and possible solutions for developing a programming model for software programmers. We conclude in Section 5.5.

5.2 Restructured Code

Optimal high-level synthesis code is not easy to write for most "non-expert" designers because it requires low-level hardware knowledge. In order to achieve a good result, it is important to write restructured code, which is often vastly different from the initial implementation. That is, an inexperienced designer will likely write the high-level synthesis code as if she was writing the application for implementation on a processor architecture. This rarely maps well to hardware.

Writing restructured code is necessary to produce good FPGA designs using high-level synthesis. Yet, it is not often obvious the best way to do this. In this section, we present a number of examples, and show how to restructure them in order to achieve good results. We provide the initial software code, and describe how to modify it into reconstructed code. We hope that this provides insight on the skills and knowledge necessary to perform the restructuring process. Then we present a formal specification of converting normal application code to restructured code for five case studies.

5.2.1 Prefix sum

Prefix sum is a common kernel used in many applications, e.g., recurrence relations, compaction problems, string comparison, polynomial evaluation, histogram, radix sort, and quick sort [21]. Prefix sum requires restructuring in order to create an efficient FPGA design.

Prefix sum is the cumulative sum of a sequence of numbers. Assume that the input sequence is stored in the array $in[]$, and the prefix sum is put into array $out[]$. Prefix sum works as follows:

$$out[0] = in[0] \tag{5.1}$$

$$out[1] = in[0] + in[1] \tag{5.2}$$

$$out[2] = in[0] + in[1] + in[2] \tag{5.3}$$

$$out[3] = in[0] + in[1] + in[2] + in[3] \tag{5.4}$$

$$\cdots \tag{5.5}$$

Listing 5.1 presents the baseline C code for prefix sum. We start the optimization process using pragmas without changing the code. We name this design *Baseline Optimized (BO)* meaning that the optimizations were done on top of the baseline code. The ideal optimization yields $II = 1$ for the loop in the code. Unfortunately, even for a relatively simple kernel like prefix sum, the designer must change the code in order to get an $II = 1$.

```
1 void prefixsum(int in[SIZE], int out[SIZE])
2 {
3   for(i=1;i<SIZE-1;i++){
4     out[i]=out[i-1]+in[i]
5   }
6 }
```

Listing 5.1. Initial prefix sum code. This would work well when implemented on a processor. However it does not map well to an FPGA.

Another important factor is creating a loop whose performance scales as it is unrolled. Ideally, as we increase the unrolling factor, the performance of the kernel increases in a linear manner. Unfortunately, this is not the case for the baseline C code for prefix sum. We must rewrite the code to allow unrolling to scale as expected.

We unrolled the loop by a factor of 4 with the hope of processing the data input $4\times$ faster. Then we performed a cyclic partitioning on the *in* and *out* arrays by a factor of 4 in order to meet the data access required by using an unroll factor of 4. We also pipelined the loop. Listing 5.2 shows code after applying these optimizations. Based upon these optimizations, we expect to get a speed up by a factor of 4.

```
1 void prefixsum(int in[SIZE], int out[SIZE])
2 {
3   #pragma HLS ARRAY_PARTITION variable=out cyclic factor=4 dim=1
4   #pragma HLS ARRAY_PARTITION variable=in cyclic factor=4 dim=1
5   for(i=1;i<SIZE-1;i++){
6     #pragma HLS UNROLL factor=4
7     #pragma HLS PIPELINE
8     out[i]=out[i-1]+in[i]
9   }
10 }
```

Listing 5.2. Baseline optimized (BO) prefix sum code

Unfortunately these optimizations do not provide a $4\times$ speedup. Even for this relatively "simple" code, the designer must change the code in order to meet the expected performance improvements. In this case, the designer must break the data dependency between $out[i]$ and $out[i-1]$. Listing 5.1 translates into hardware architecture depicted in Figure 5.1 (a). In order to get the desired speed up, the designer must change the architecture in Figure 5.1 (a) to an architecture like that in Figure 5.1 (b), which introduces a register *A* to store the input *in*. The architecture in Figure 5.1 (b) provides

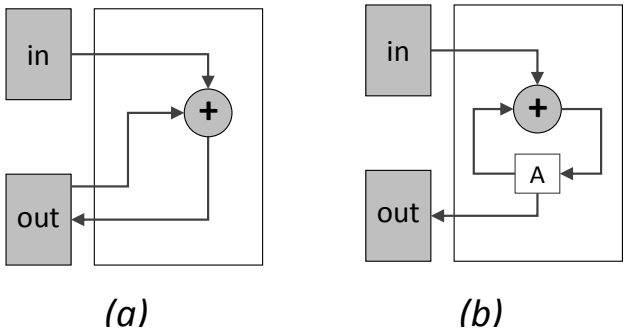


Figure 5.1. Part a) is the hardware architecture created by the code in Listing 5.1. Part b) is the hardware architectures corresponding to the restructured code in Listing 5.3. Breaking the dependency between the *out* array creates a more optimal design.

a hardware architecture that gives the desired linear speed up based upon the unrolling factor. Listing 5.3 shows restructured code for the architecture in Figure 5.1 (b) that provides such a linear speed up when the loop is unrolled. We call this code *Restructured Optimized 1 (RO1).*

```
1 void prefixsum(int in[SIZE], int out[SIZE]){
2 #pragma HLS ARRAY_PARTITION variable=out cyclic factor=4 dim=1
3 #pragma HLS ARRAY_PARTITION variable=in cyclic factor=4 dim=1
4 A=in[0];
5 for(i=0;i<SIZE;i++){
6   #pragma HLS UNROLL factor=4
7   #pragma HLS PIPELINE
8   A = A+in[i];
9   out[i] =  A;
10  }
11 }
```

Listing 5.3. Restructured optimized (RO1) prefix sum code

Now we explore another restructuring technique for the prefix sum using a reduction pattern [91]. We name this design restructured optimized (*RO2*). Figure 5.2 (a) presents 8 input prefix sum using a reduction pattern. Eight inputs requires three reduction stages.

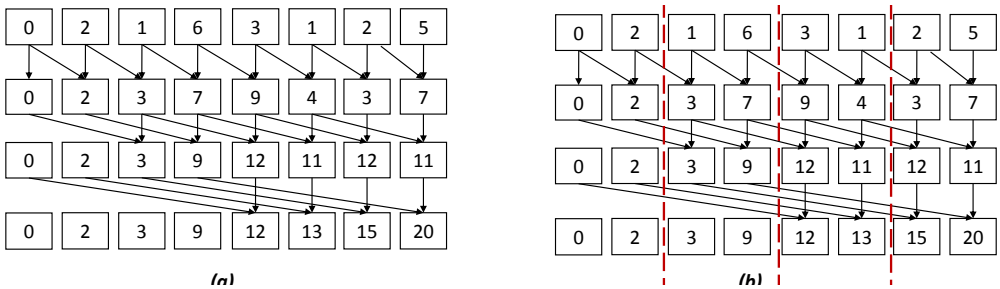


Figure 5.2. a) Eight input prefix sum using reduction. b) A potential partitioning of the prefix sum.

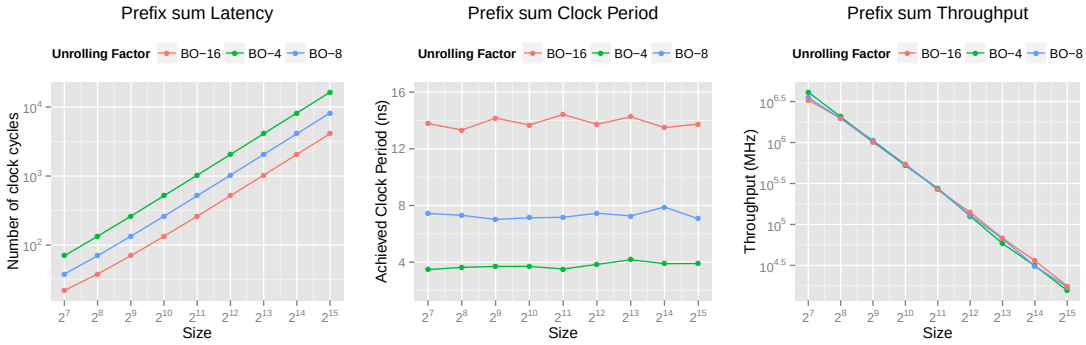


Figure 5.3. BO-4, BO-8, BO-16 are three different designs with unrolling factor of 4, 8, and 16. The clock period varies between 2.75-3.75 nano seconds. Overall, the throughput is almost identical for these designs despite the fact that the unrolling factor should expose more parallelism.

These stages can be pipelined using the `dataflow` pragma. Using the reduction pattern alone does not improve the throughput. Thus, we applied cyclic partitioning and unrolling on top of the restructured *RO2* design. The complete code for this architecture can be found in Appendix A.1.

Next, we will discuss hardware performance area results of *BO*, *RO1* and *RO2*. Figure 5.3, Figure 5.4, and Figure 5.5 show results of designs BO, RO1, and RO2, respectively. Each of these figures show latency (number of clock cycles), achieved clock period, and throughput (number of samples per second). All of the results are obtained from Vivado HLS 2014.1 after place and route. The latency of *BO-X* designs scale according to the unroll factor $(X = 4, 8, 16)$ as shown in Figure 5.3 (first figure).

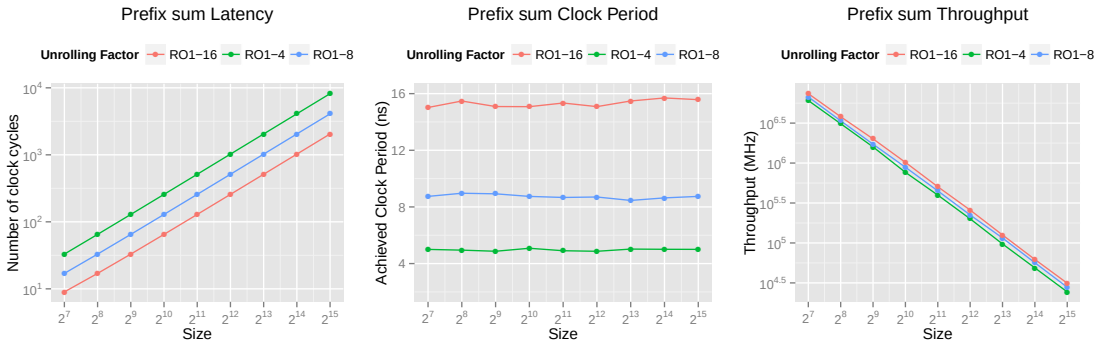


Figure 5.4. RO1-4, RO1-8, and RO1-16 are three different designs with unrolling factor of 4, 8, and 16, respectively. The latency decreases from *Baseline Optimized*, and it does scale based on unroll factor. The clock period varies between 3.75 and 4.75 nano seconds. Overall, the throughput increases little bit.

With an unrolling factor $(X = 4, 8, 16)$, the frequency decreases by the same amount of unrolling factor $(X = 4, 8, 16)$. In other words, the frequency decreases according to the unrolling factor $(X = 4, 8, 16)$. As a result, the throughput of the *BO-X* designs in Figure 5.3 does not scale as expected. In fact, there is not relative benefit for doing this unrolling optimization.

Recall the design *RO1* corresponds to the design with the data dependency removed (Listing 5.3). Figure 5.4 shows that by removing this dependency we get a better latency than the *BO* designs (the original design with the data dependency). Yet the throughout for these *RO1* designs with different unrolling factors remains very similar. This is due to the fact that the clock period for the unrolled designs increases with the unrolling factor – the same trend that we saw for the *BO* design (see Figure 5.3). The throughputs are better than *BO*, yet they still do not scale with the unrolling factor as we would hope.

The design *RO2* uses a reduction pattern. Figure 5.5 shows that the latency has a similar pattern to both *BO* and *RO1*, i.e., the number of cycles decreases as the unrolling factor increases. The major difference is in the frequency. The clock period for each of

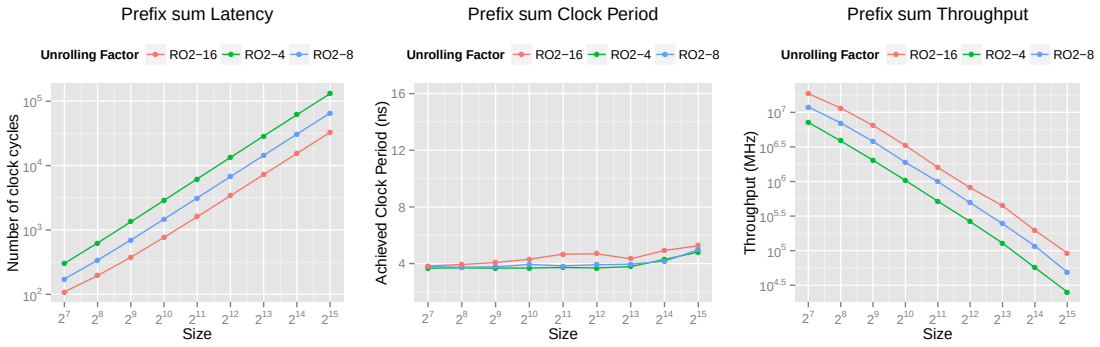


Figure 5.5. RO2-4, RO2-8, RO2-16 are three different designs with unrolling factor of 4, 8, and 16. The latency scales based on unroll factor. Clock period varies between 3.58 and 5.54 nano seconds. Here the throughput scales based on unroll factor.

the designs is between 3.5 and 5.6 ns. Thus, the throughput exhibits the desired scaling as the unrolling factor increases. This is different result that both of the previous designs. Overall, the throughput is better than both *BO* and *RO1*, and this design scales with the unrolling factor.

5.2.2 Histogram

Histogram is a common kernel used in image processing, signal processing, data processing (e.g., databases) and many other domains. Figure 5.6 presents an example of histogram calculation. Given a set of data, a histogram counts the occurrence of each element (Bin=elements, Count=occurrence). The result is plotted in a graph.

Listing 5.4 shows a baseline implementation of a histogram algorithm, e.g., one that a designer would code without regard to optimizing it for hardware. This algorithm calculates histogram of image which has size *IMAGE_SIZE*, and it outputs a histogram of pixels to the *hist* array which has size of *HIST_SIZE*. We call this design *Baseline Optimized (BO)*.

```
1 void hist(int pixel[IMAGE_SIZE], int hist[HIST_SIZE]){
2 int val;
3 for(int i=0;i<IMAGE_SIZE;i++){
4
```

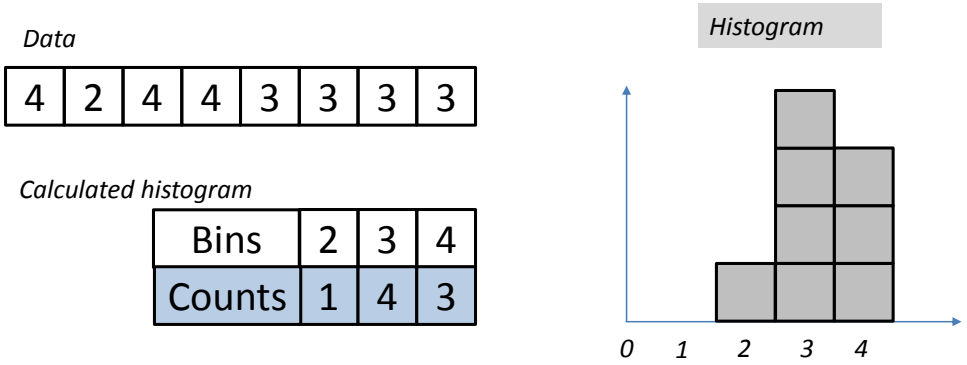


Figure 5.6. A histogram kernel counts the occurrences of the elements in a set of data.

```
5 #pragma HLS PIPELINE
6   val = pixel[i];
7   hist[val] = hist[val] + 1;
8 }
9
```

Listing 5.4. The baseline optimized (BO) code for the histogram kernel.

To optimize this code for hardware, we can direct the high-level synthesis tool to exploit instruction level parallelism by applying the `pipeline` pragma to the body of the `for` loop. We are unable to achieve an $II = 1$ due to a read-after-write (RAW) dependency. We are reading from the array *hist*[] and writing to the same array in every iteration of the loop. Figure 5.7 (a) shows the hardware architecture for the code in Listing 5.4.

We can eliminate the read after write dependency using an architecture shown in Figure 5.7 (b). The architecture uses an accumulator (*accu*), comparator, and old value of current address. This architecture updates the accumulator if the current address and old address are the same. Otherwise, it writes the accumulator value to the old address and reads a new accumulator value from the current address. In this case, the code reads from and writes to a different address. Thus, the RAW dependence can be ignored. Most

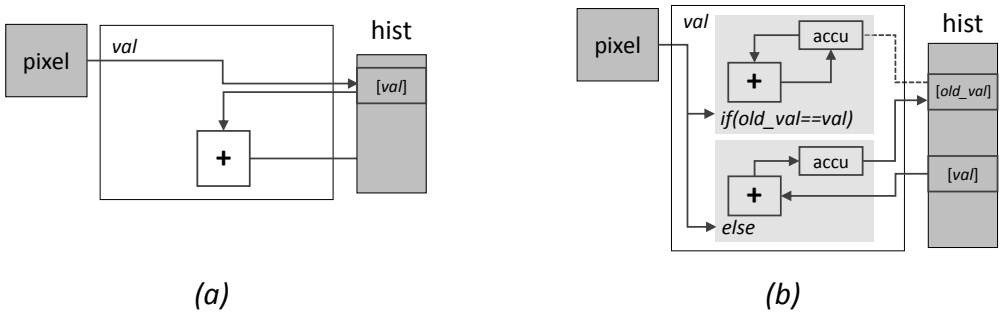


Figure 5.7. Hardware architectures created by software code versus hardware architectures created by restructured code for prefix sum.

HLS tools have an directive to force the tool to ignore a RAW dependency.

Listing 5.5 shows a restructured code which provides $II = 1$ that represents the architecture in Figure 5.7 (b). This restructured version eliminates the RAW dependency from Listing 5.4. We call this design *Restructured Optimized 1 (RO1).*

```
1 #pragma HLS DEPENDENCE variable=hist array intra RAW false
2 for(i=0;i<IMAGE_SIZE;i++){
3    #pragma HLS PIPELINE II=1
4   val = pixel[i];
5   if(old == val)
6     accu = accu + 1;
7   else {
8     hist[old] = accu;
9     accu = hist[val]+1;
10  } old = val;
11 }
```

Listing 5.5. The Restructured Optimized 1 version of the histogram kernel. The code enables an $II = 1$. It performs the histogram kernel in two parts. First it compares current and old values of the element under consideration. If they are the same it increments the accumulator (*accu*) register. The *accu* is a register so it does not cause any stall which reduce the II. In the else clause the elements are not the same. Therefore we are reading and writing to different locations of the *hist* array. Thus we can ignore the RAW dependency.

We can further optimized the histogram kernel by running several of the architectures in parallel to boost the throughput. We start with the RO1 architecture as described

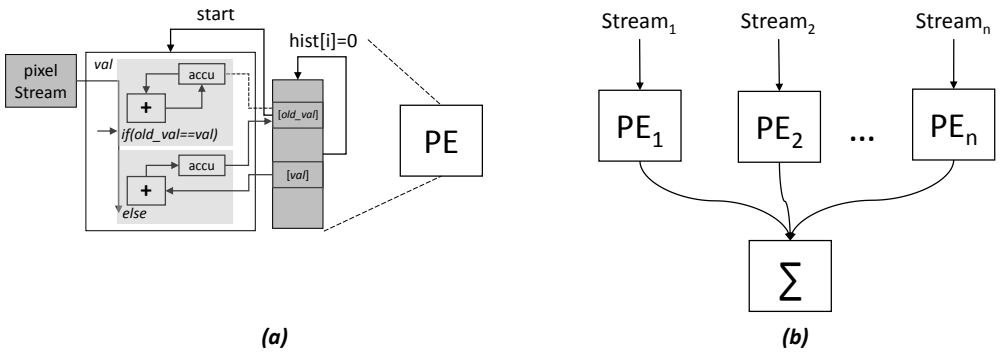


Figure 5.8. Histogram: a) PE for histogram, b) Communicating PEs.

in Figure 5.7 (b). The goal is to create multiple versions of this architecture, which we call a processing element (PE), split the input data into different streams, and feed them independently into a PE. Then we must merge the results from all of the PEs. This idea is shown in Figure 5.8. We can this the *Restructured Optimized (RO2)* design.

Due to the limitations of current high-level synthesis tool, we must initialize the storage in each PE in order to run them in parallel. This is shown in Figure 5.8 (a). It is the architecture in Figure 5.7 (b) with some additional logic to initialize the *hist*[] array. Listing 5.6 provides the source code for the *RO2*. In general, *RO2* has higher memory usage because it has intermediate local buffers(hist1, hist2) in lines 39 and 40. Here we formulate required number of local memory for *RO2*. If there are n processing elements, and each processing element requires m local memories. Then *RO2* will use total $n \times m$ local memory if we use BRAMs (mapping arrays to BRAMs). Ideally, we want higher throughput while using less number of local memories. To use less local memories for design like *RO2* in HLS, we map intermediate arrays (as in the case of *RO2*) to a local streams. In Listing 5.6, we mapped hist1 and hist2 into local streams in lines 41 and 42.

```
12 #include "hist.h"
13 void histo(int pixel[IMAGE_SIZE2], int hist[HIST_SIZE]){
14 #pragma HLS DEPENDENCE variable=hist array intra RAW false
15   int val=1;
16   int old_val=2;
17   unsigned int accu=0;
```

```
18   //Initialization
19   for(int i=0;i<HIST_SIZE;i++){
20     hist[i]=0;
21   }
22
23 //PE Code below
24 ...
25
26 }
27
28 //Merge function
29 void Merge(DTYPE hist1[HIST_SIZE], DTYPE hist2[HIST_SIZE], DTYPE final[HIST_SIZE]) {
30   for(int i=0;i<HIST_SIZE;i++){
31     final[i] = hist1[i] + hist2[i];
32   }
33 }
34 //Top level function
35 void hist(int pixelA[IMAGE_SIZE2], int pixelB[IMAGE_SIZE2], int hist[HIST_SIZE]){
36 #pragma HLS INTERFACE ap_fifo port=pixelA
37 #pragma HLS INTERFACE ap_fifo port=pixelB
38 #pragma HLS DATAFLOW
39 DTYPE hist1[HIST_SIZE];
40 DTYPE hist2[HIST_SIZE];
41 #pragma HLS STREAM variable=hist1
42 #pragma HLS STREAM variable=hist2
43
44 histo (pixelA, hist1);
45 histo (pixelB, hist2);
46
47 Merge(hist1, hist2, hist);
48 }
```

Listing 5.6. HLS code for architecture in Figure 5.8 (b)

Next, we present results of the three different architectures for the histogram kernel. Figure 5.9, shows the results of *BO*; Figure 5.10 is *RO1*; and the results for design *RO2* is in Figure 5.11. Each of these figures has a graph for latency (total number of clock cycles), achieved clock period, and throughput (number of samples per second). All results are obtained from Vivado HLS 2014.1 after place and route.

The results for the *BO* in Figure 5.9 indicates that unrolling does not help with this architecture. This is due to the RAW dependency. The clock periods are all relatively the

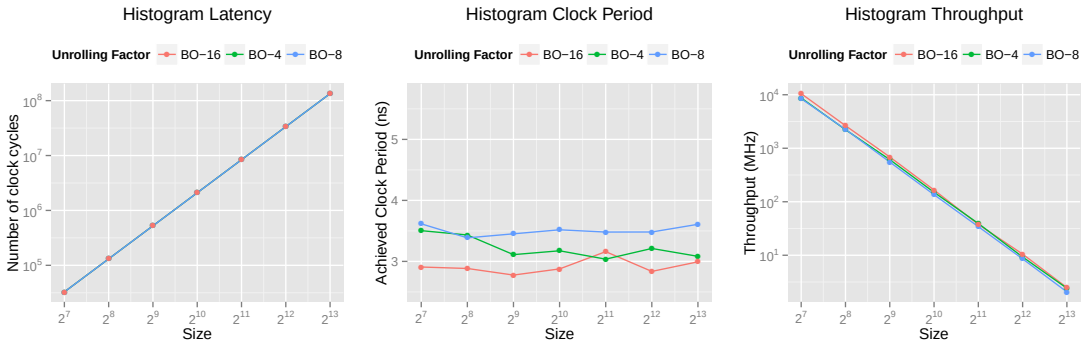


Figure 5.9. *Baseline Optimized (BO)* implementation of the histogram kernel. The latency does not scale based on unroll factor. Clock period varies between 2.75-3.75 ns. Thus, the throughput for the different unrolled architectures are very similar.

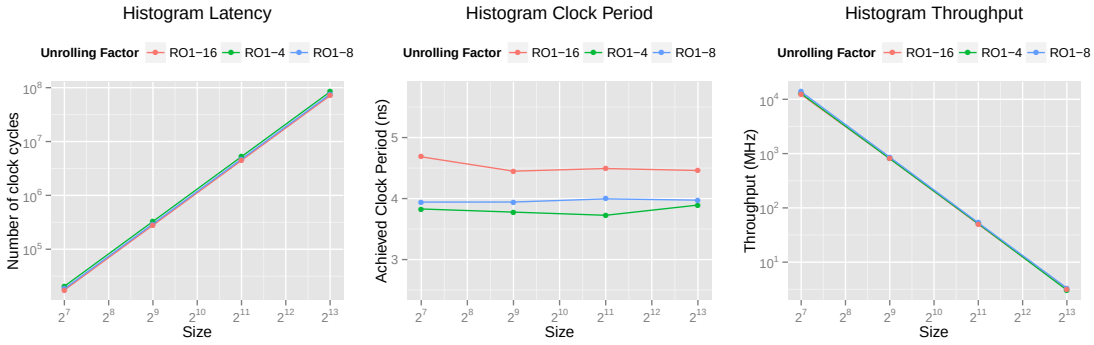


Figure 5.10. Histogram: *Restructured Optimized 1*. Latency decreases from *Baseline Optimized* and does not scale based on unroll factor. Clock period varies between 3.75-4.75 nano seconds. Throughput increases little bit.

same (between 2.75 and 3.75 ns). The throughput decreases as the number of elements increases, and unrolling factor shows almost no benefit.

The *RO1* provides overall better throughput than *BO*. This is due to the ability to lower the *II* by eliminating the RAW dependency. The unrolling factor does not help the latency, and the clock periods are all relatively similar. Thus, the unrolling does not increase the throughput when measured in samples per second.

RO2 does provide better throughput than *BO* and *RO1* because it has achieved better clock period with the same latency. However, *RO2* uses more memory than *BO* and *RO1*. *RO2* uses larger number of local memories because there is more parallelism

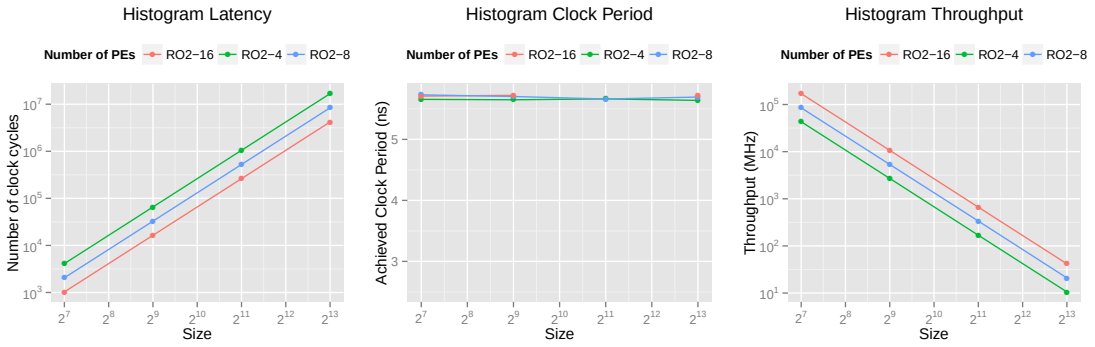


Figure 5.11. Histogram: *Restructured Optimized 2*. RO-4,RO-8, RO-16 are designs with different number of PEs. Latency does scale based on the number of PEs. Clock period varies between 5.54-5.75 nano seconds. Throughput scales based on the number of PEs.

(4,8,16) PEs operating in parallel.

5.2.3 SpMV: Sparse Matrix Vector Multiplication

Sparse matrix vector multiplication (SpMV) is a common kernel that found in application related to graph algorithms, compressive sensing, digital signal processing, and computer vision. SpMV does multiplication of a sparse matrix M with a vector V. M being a sparse means that many of its elements are zero.

We focus on a sparse matrix vector multiplication using the compressed sparse row (CSR) matrix format. CSR format stores the sparse (matrix with many zero elements) in a compressed format to save memory. Figure 5.12 (a) shows an example of a sparse matrix M. For the sake of simplicity, we assume sparse matrix has a size of 4×4. Figure 5.12 (b) shows the corresponding compressed sparse row matrix format for M. In CSR format, there are three structures namely *ptr*, *ColInd*, and *Val*, and corresponding values of these data structures are also given in Figure 5.12 (b). Data structure *ptr* is a list of value of starting index of each row. The last element in *ptr* is the ending index of the last row. Thus the number of elements in *ptr* is equal to the number of rows plus one. For example, the first element in *ptr* is 0 (Figure 5.12) and second element is 2. This means

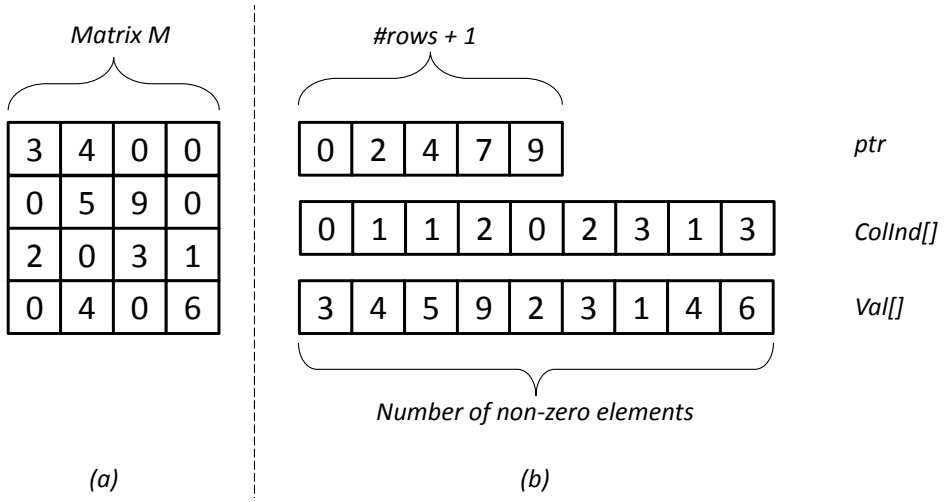


Figure 5.12. a) A sparse matrix (normal representation) M of size 4×4, b) Compressed Sparse Row matrix (CSR) format for M. Since first row starts from 0 and there is 2 non-zero elements (3 and 4) the second row must start from 2. Thus the second element of ptr must be 2. *Val* is constructed by non-zero elements so it contains all non-zero elements from the matrix M. *ColInd* is the corresponding index of values *Val* in the original matrix.

the first row starts from 0 and it end at 2 (the first row of sparse matrix has 2 non-zero elements) where the second row starts. *Val* is an array of (left to right, then top to bottom) non-zero values of matrix M. *ColInd* is index of columns corresponding to the *Val*. The number of elements in *Val* and *ColInd* are equal to the number of non-zero elements in the sparse matrix.

```
1 void spmv(int num_rows, DTYPE ptr[NUM_ROWS+1], DTYPE ColInd[NNZ], DTYPE Val[NNZ], DTYPE
      y[SIZE], DTYPE x[SIZE])
2 {
3   L1:for (int i = 0; i < NUM_ROWS; ++i) {
4     L2:for(k = p[i]; k < p[i+1]; ++k)
5     {
6       #pragma HLS PIPELINE II=1
7       y0 = y0 + Val[k] * x[ColInd[k]];
8     }
9     y[i] = y0;
10  }
11 }
```

Listing 5.7. Initial C code for Sparse Matrix Vector Multiplication

```
1 void spmv(int num_rows, DTYPE ptr[NUM_ROWS+1], DTYPE ColInd[NNZ], DTYPE Val[NNZ], DTYPE
      y[SIZE], DTYPE x[SIZE])
2 {
3   L1:for (int i = 0; i < NUM_ROWS; ++i) {
4     DTYPE a =ptr[i];
5     DTYPE b =ptr[i+1];
6
7     L2:for(k = a; k < b; ++k)
8     {
9       #pragma HLS PIPELINE II=1
10      y0 = y0 + Val[k] * x[ColInd[k]];
11    }
12    y[i] = y0;
13  }
14 }
```

Listing 5.8. Sparse Matrix Vector Multiplication that provides II equal 1 regardless of underlying HLS tool

Listing 5.7 presents an initial C code for SpVM. We pipelined the loop $L2$ to achieve $I = 1$. Depending on the underlying HLS tool, $II = 1$ might not be possible. For example, Vivado HLS 2014.1 does not achieve $II = 1$ when the Listing 5.7 is given as an input. On the other hand, Vivado HLS 2014.4 achieves $II = 1$ when the Listing 5.7 is given as an input. We will discuss a possible scenario which prevents achieving $II = 1$ for loop $L2$. $II = 1$ is not possible when HLS tool loads $ptr[]$ at the staring cycle of $L2$. To solve this problem, HLS tool must load the $ptr[]$ before starting the $L2$ or we must manually modify the code in Listing 5.7 so that we force the HLS tool to load the value of $ptr[]$ before the loop $L2$ starts. The modified code is shown in Listing 5.8. In the rest of this section, we assume our HLS tool load the value of $ptr[]$ before the loop starts.

Using the code in Listing 5.7, we want to show that impact of different HLS (pipeline, unroll) optimization on this code. HLS design of irregular programs is tricky because some optimizations such as unroll does has an adverse effect when used incorrectly. For example, the unroll pragma is expected to decrease the number of clock cycles, but if it is used incorrectly (e.g., places in wrong place), it will increase the number of

clock cycles.

We generated 10 different designs for the sparse matrix vector multiplication by adding pragmas to Listing 5.7. We call these designs Baseline Optimized (BO). One of the designs (Case 5) is shown in Listing 5.9. In Listing 5.9, we did following optimizations; we applied the pipeline pragma to L2; we unrolled L2 by a factor of 2; and since we are unrolling L2 by a factor of 2, we partitioned the input arrays with cyclic partition with the same factor (a factor of 2) to increase parallelism so that L2 can access data in parallel.

```
12 void spmv(int num_rows, DTYPE ptr[NUM_ROWS+1], DTYPE ColInd[NNZ], DTYPE Val[NNZ], DTYPE
       y[SIZE], DTYPE x[SIZE])
13 {
14 #pragma HLS ARRAY_PARTITION variable=Val cyclic factor=2 dim=1
15 #pragma HLS ARRAY_PARTITION variable=x cyclic factor=2 dim=1
16 #pragma HLS ARRAY_PARTITION variable=ColInd cyclic factor=2 dim=1
17
18   L1:for (int i = 0; i < NUM_ROWS; ++i) {
19     L2:for(k = ptr[i]; k < ptr[i+1]; ++k)
20     {
21       #pragma HLS PIPELINE II=1
22       #pragma HLS UNROLL factor=2
23
24       y0 = y0 + Val[k] * x[ColInd[k]];
25     }
26     y[i] = y0;
27   }
28 }
```

Listing 5.9. Baseline Optimized for Case 6 from Table 5.1.

We show the set of optimizations used in the remaining nine designs in Table 5.1. Next, we will discuss why we choose these optimizations and their impact on the final generated hardware results. First of all, there are two loops in initial C code in Listing 5.7. Since there are two loops (L1 and L2), we must cover optimizations that can be applied to both of them. We can do different combinations of loop pipelining and unrolling optimizations combined with data partitioning for the code in Listing 5.7. These

optimizations covers both loops. By combining these optimizations, we generate total of 10 different HLS designs (Case 5 was discussed earlier and presented in Listing 5.9).

Table 5.1. Optimizations on a sparse matrix-vector multiplication.

	Optimizations		Results		
	L1	L2	Latency	DSP	FF/LUT
Case 1	-	-	220758	4	331/219
Case 2	-	pipeline	22153	4	302 / 239
Case 3	pipeline	-	220758	4	331 / 219
Case 4	unroll=2	-	220630	4	448 / 485
Case 5	-	pipeline, unroll=2	13703	8	593 / 440
Case 6	-	pipeline, unroll=2, cyclic=2	14215	8	850 / 694
Case 7	-	pipeline, unroll=4	14084	16	1230 / 693
Case 8	-	pipeline, unroll=4, cyclic=4	14852	16	2290 / 2158
Case 9	-	pipeline, unroll=8	14846	32	1681 / 1143
Case 10	-	pipeline, unroll=8, cyclic=8	16638	32	6391 / 7439

Table 5.1 shows the results of these different designs. Case 1 has no optimizations. This is equivalent to synthesizing the code in Listing 5.7. Case 1 has a latency of 220758 clock cycles, and it uses 4 DSPs 331 FFs, and 219 LUTs. In Case 2, we pipelined L2 which reduces the number of clock cycles from 220758 to 22153. The FFs decreased to 302, and the LUTs increased to 239. Case 3 pipelines L1. This design has a latency of 220758 cycles, which is the same latency as Case 1. This is because the pipeline directive is being ignored due to undefined loop bounds in L2. In other words, we can not pipeline L1. Case 4 unrolls loop L1 by a factor of two. This does not decrease the number of clock cycles as expected by a factor of 2. This is because need to partition the data accessed in the inner loop($ColInd$, Val indexed by value of ptr) by the unrolling factor. Since these data structures are accessed randomly, we can not partition the data into separate memories. Case 4 has similar results as Case 3.

In Case 5, we pipelined and unrolled loop L2. This results in a design with 13703 clock cycles; it has 8 DSPs, and increased the number of FFs and LUTs by approximately

a factor of two compared to the baseline design (Case 1). Case 5 gives results as expected by the unrolling factor (i.e., the clock cycles decrease, and the area increases by a factor of two). Then we applied cyclic partitioning to the arrays in Case 6 (Listing 5.9). The optimizations in Case 6 increased the latency and the area. This means partitioning does not help. To get deeper understanding of effectiveness of unrolling and partitioning factor, we did increased unrolling and partitioning factor in Case 7 through Case 10. As we increase the unrolling factor, we expect the latency to decrease. Conversely, this does not happen and the area increases. In Case 10 area increased dramatically.

To conclude, Cases 7 through 10 and Case 3 and 4 do not provide the results as expected. It is often the case that designers blindly apply directives; this is rarely effective. In these designs, the directives do not match well with the code, and thus the underlying hardware architectures are not optimal. For example, we can not unroll or pipeline loop L1 when loop L2 contains undefined loop bounds. These designs provide an example where optimizations in HLS have adverse effect. For example, Case 10 increases the area dramatically. As a result, only Case 3 and Case 5 provide good results amongst these 10 designs.

We choose to further explore Case 2 in order to increase the throughput. Figure 5.13 shows hardware architecture corresponding to Case 2. In Case 2, we pipeline the MAC unit in the Figure 5.13.

We create a design called Restructured Optimized where we run several architectures of Case 2 in parallel. In other words, we make the architecture in Figure 5.13 as one PE (processing element), and run several PEs in parallel. Each of these PEs performs the computation on different regions of the matrix vector multiplication. Figure 5.14 presents an example hardware architecture of the RO design when using two PEs in parallel.

As these examples (prefix sum, histogram, SpMV) show, even the most trivial kernels require substantial hardware expertise to optimize these kernels. Some example

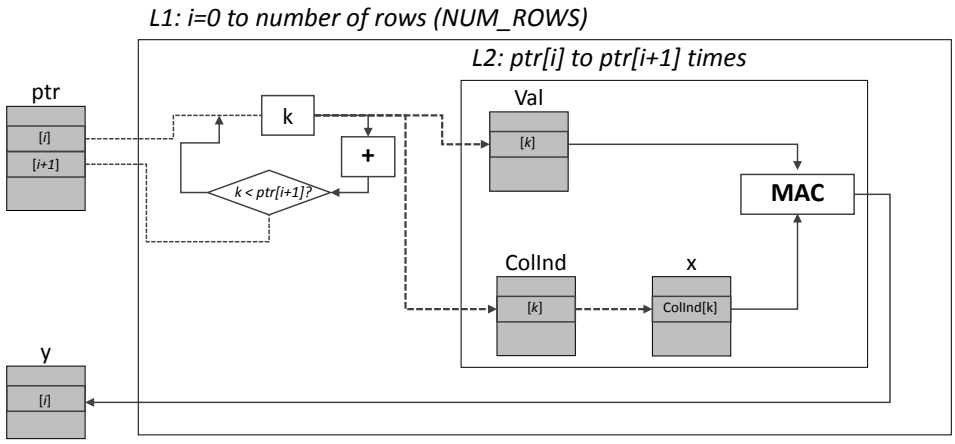


Figure 5.13. a) Hardware architecture of a sparse matrix vector multiplication. ptr, y, Val, $ColInd$, x are arrays stored in BRAMs. The outer loop iterates from 0 to number of rows in the matrix from ptr. Each time two elements are read from ptr; the first one $(ptr[i])$ is the lower and the second one $(ptr[i+1])$ is the upper bound of inner loop (k in the figure iterates from the first ptr[i] to ptr[i+1]. The inner loop calculates one element of y at a time accessing Val, $ColInd$ and x with index k.

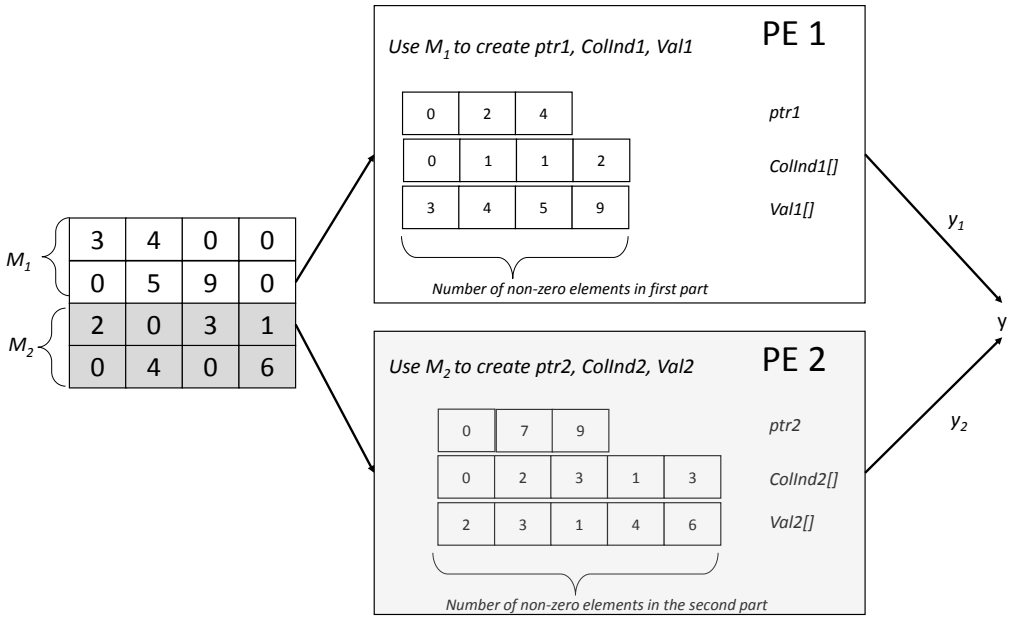


Figure 5.14. a) Hardware architecture of a sparse matrix vector multiplication using two PEs. We divide the sparse matrix into two halves (M_1 and M_2) and run different PEs on different portions of the data. In this example, M_1 is feed into PE1, and M_2 is feed into PE2. We collect y_1 and y_2 to craete y. Here y_1 and y_2 have a size of 2 and y has a size of 4.

kernels (prefix sum and histogram) mainly show optimization for instruction-level parallelism (getting $II = 1$). We also explored different effects of HLS optimizations for irregular kernel (e.g, SpMV). We learned that applying different powerful HLS optimizations sometimes does generate worse hardware than less powerful optimizations. We also presented increasing ways of increasing throughput using several PEs (mainly task level parallelism). Task level parallelism is important concept to design hardware in HLS. Next, we will explore deeper into task level parallelism using more complex kernels.

5.2.4 FFT

Fast Fourier Transform (FFT) is a common kernel found in applications such as wireless communication, digital signal processing, and image processing applications. FFT converts (computes) a time domain signal to the frequency domain. There are a number of algorithms to compute FFT. In this section, we walk through the design of the Cooley-Tukey FFT [43] algorithm for implementation in hardware. We start with a common version of the code, and then describe how to restructure it to achieve a better hardware design. Our design results are competitive with highly optimized FFT IP core from the Xilinx CoreGen library.

An N-point FFT has $\log_2 N + 1$ stages. Bit reverse is the first stage. This stage swaps a value of the input data with the element located at the bit reversed address in the array. After this bit reverse stage, we perform $\log_2 N$ stages of butterfly operations where N is the number of FFT points. Each of these butterfly stages has the same computational complexity. We define each of these sub stages as a task.

Figure 5.15 provides a graphical depiction of an 8-point FFT. The first stage swaps the data; this is done to insure that the output frequency data is presented in order. The data is swapped by taking the index of an element, getting the bit reversed index, and then swapping those two elements. For example, consider the data at $x(1)$ or equivalently

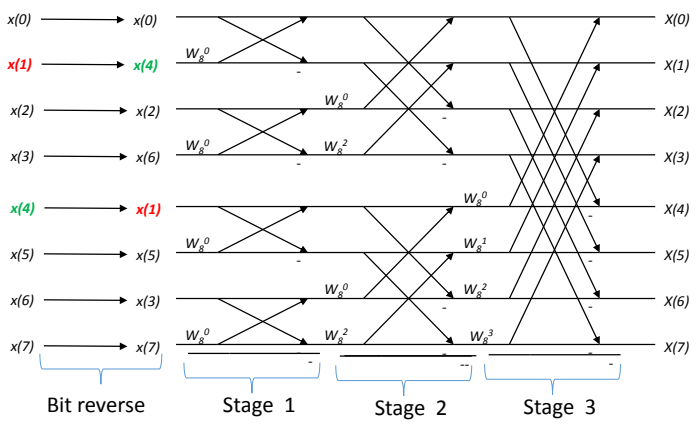


Figure 5.15. An 8-point FFT. The first stage swaps the data using a bit reverse algorithm. The next three stages perform butterfly operations.

$x(0b001)$. Performing a bit reversal on the binary number $0b001$ yields $0b100 = 4$. Thus, we swap the data values $x(1)$ and $x(4)$. The remainder of the data is swapped as shown in Figure 5.15.

After the bit reverse stage, we perform $\log_2 8 = 3$ stages of butterfly operations. A butterfly has two inputs and two outputs. Consider the upper-left-most butterfly in the Figure 5.15. The inputs are $x(0)$ and $x(4)$.

Listing 5.10 provides a high level overview of an N-point FFT. It presents a common three loop structure implementation of FFT, which is typically seen in a software version. In this code, the loop in line number 10 iterates each stage of N-point FFT. Therefore, the scale of code that constitutes body of a task (stage) is between line numbers 11 and 16. According to this code each task has two nested loops with complexity of $N/2 \times N/2$. The loops in line numbers 11 and 12 both iterates $N/2$ times. The loop in line number 11 iterates each butterfly in each stage while the loop in line number 12 iterates points which have the same twiddle factor. Since inner two loops have $N/2$ iterations, we merge them and calculate indices of each loop as needed. Therefore, final complexity of each task is $N/2$ iterations instead of $N/2 \times N/2$. We need to change the original code so that each sub task must finish in $N/2$ iterations.

```
1 void FFT(IN[], OUT[])
2 {
3 // DFT = 2^stage = points in sub DFT
4 DFTPoints = 1 << N;
5
6 // Butterfly WIDTHS in sub-DFT
7 NumButterfly = DFTPoints/2;
8
9 BitReverse((IN, IN);
10 for((i=0;i<N;i++{
11   for(j=0;j<NumButterfly; j++){ //Runs logN
12     for((k=0;k<N; k+=DFTPoints{ //Runs logN
13       //Butterfly calculation
14       ... ...
15     }
16   }
```

Listing 5.10. Software C code for N-point FFT.

```
1 void FFT(IN[], OUT[])
2 {
3
4 for(i=0;j<N/2; i++){ //Runs logN
5   //Calcualte indices as needed.
6       ... ...
7 }
```

Listing 5.11. Body of restructured stage of FFT

We chose to design non-streaming model using BRAMs between different stages of FFT. Stages of FFT has a non-sequential access to incoming data. sAs shown in Figure 5.15, in the first stage of 8-point FFT, it calculates butterflies between two consecutive incoming data. But in the next stages, it calculates butterfly using data at locations in alternating indices. (Use data from 0 and 2). Many real world applications that require FFT needs moderate sizes of FFT usually 64 up to 1024 (or even larger up to 16 KB). Therefore, it is feasible to use BRAMs between different stages of FFT. It is straightforward to draw block diagram of FFT as shown in Figure 5.16 after defining tasks and memory mode. We will name this kind of computational pattern (as shown

in Figure 5.16) a bulk synchronous model (non-streaming) which will be discussed in Chapter 6. The given a top level function code for this design is shown in Listing 5.12. For the sake of simplicity, we give an example of 8-point FFT. The software C code for 8-point FFT has three nested loops as mentioned before. The outer loop is iterating for each stage (8-point FFT has three stages). The functions ($Stage1$, $Stage2$ and $Stage3$) are calculating each stage of 8-point FFT. In general case, we will have $logN$ functions of each stage of FFT. We also made the bit reverse stage as the first function of the design. We use $MEM1$, $MEM2$ and $MEM3$ variables for buffer memories between different tasks (stages). These memories implemented by BRAM. Stages are implemented as having input from previous stage and output goes to next stage. Final output of $Stage3$ is the output of top level function. We used $dataflow$ directive to pipeline stage functions. The final throughput of the design is the same as the number of clock cycles needed for one stage. (All stage have same number of clock cycles).

```
1 void FFT(IN[], OUT[])
2 {
3 #pragma dataflow
4
5 MEM1[N];
6 MEM2[N];
7 MEM3[N];
8
9 BitReverse((IN, MEM1);
10 Stage1(MEM1, MEM2);
11 Stage2(MEM2, MEM3):
12 Stage3(MEM3, OUT);
13 }
```

Listing 5.12. Top level function of restructured 8-point FFT with non-streaming bulk synchronous model

Experimental Results: FFT

We evaluated the functionality of system by using simulator incorporated with HLS tool. The simulator evaluates the generated RTL with Modelsim. We then syn-

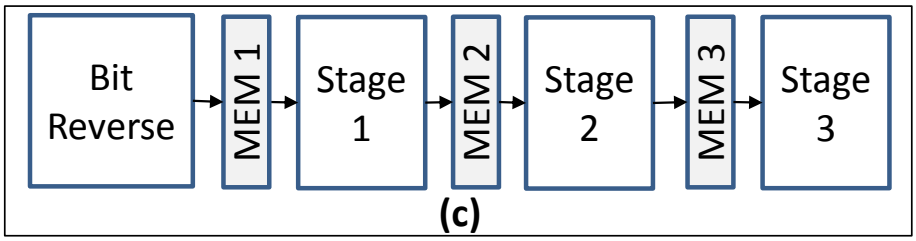


Figure 5.16. Block diagram of FFT

thesised the system with ISE 14.3 on Zynq device (xc7z020clg484-1). We tested the design with different size of FFTs (256×256, 512×512, 1024×1024, 2048×2048 and 4096×4096). We presented area(slices) and throughput (million FFTs per second) in Figure 5.17. We also compared our FFT hardware results with highly optimized CoreGen FFT IP from Xilinx. CoreGen throughputs are little bit better than our design for small sizes FFTs (256-FFT, 512-FFT, 1024-FFT). CorGen also tends to have smaller area than our design. The reason our FFT has larger area, we use larger number of bits for input data. CoreGen FFT has 14 bit input for all sizes of FFT and based on the FFT size, it uses different bit widths for each stage. Thus, CoreGen FFT has different output bit widths for each size. For example, CoreGen FFT has 23, 24, 25, 26 and 27 bit widths for FFT sizes 256×256, 512×512, 1024×1024, 2048×2048 and 4096×4096 respectively. We used same bit widths for both input and outputs in our design. Our bit widths are set to be same as CoreGen output bit widths. We did not optimized individual stage bit widths as CoreGen does. Therefore, we have lots of room to improve our area results. All in all, our results are almost competitive with highly optimized FFT core CoreGen.

5.2.5 Huffman Tree Creation

This section extends Huffman tree creation process from Chapter 4, and it covers the restructuring necessary to transform software-optimized C code for Huffman tree creation into restructured code for current HLS tools.

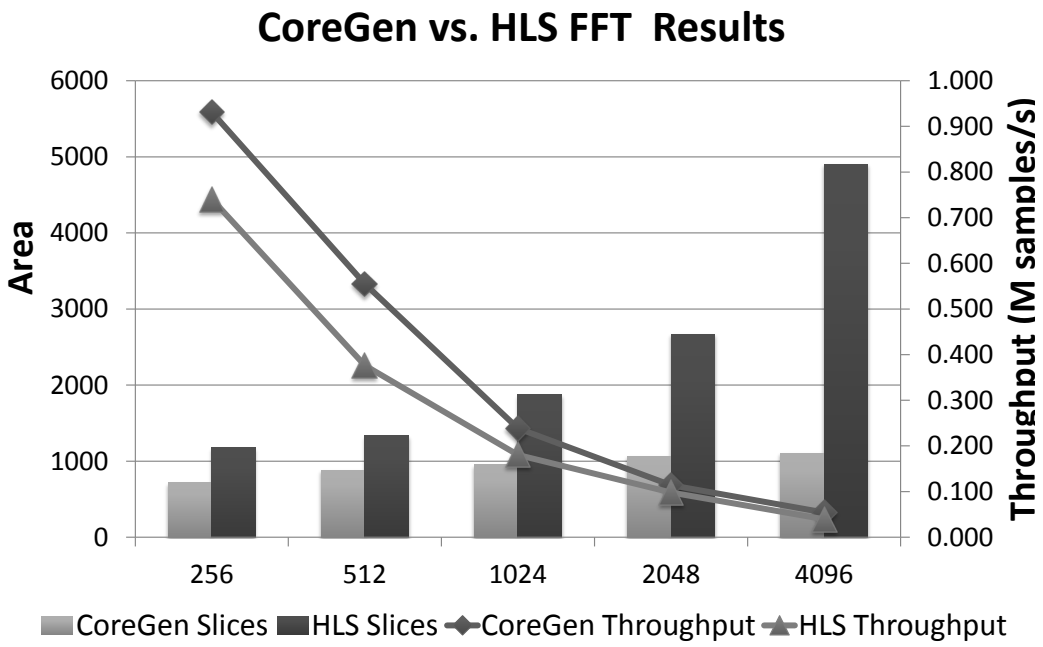


Figure 5.17. Hardware area and throughput results of our FFT and CoreGen FFT.

Listing 5.13 demonstrates software code for Huffman tree creation, and operates as follows: The input *list* is generated from an arbitrary text source and contains a list of symbols, sorted by the frequency of each symbol in the text. During tree creation, two elements with minimum frequencies (lines 2 and 3) are selected to form a new intermediate node as in line 4. The new node is added to the input list maintaining sorted order (line 7). Figure 5.18 shows a complete Huffman tree where and *F* and *B* have the minimum frequencies of 1. We select *F* and *B* to make a new node which will be added to the list with a frequency of 2. The above process continues until no element is left in the sorted input list. The result is the Huffman tree, where the bit length of each symbol is calculated by traversing the tree. For example, *F* and *B* have bit length of 4 while *A* has bit length of 2.

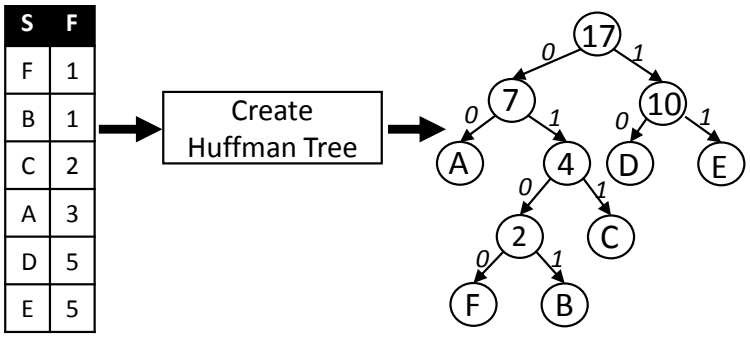


Figure 5.18. Huffman Tree: S=Symbol, F=Frequency

Software Code

Listing 5.13 presents pseudo code for a subroutine of creating a Huffman tree. This pseudo code is optimized for a CPU application, not for an FPGA design, which leads to drawbacks when it is processed by HLS tools. First, the *while* loop in line 1 is unbounded. Unbounded loop operation prevents hardware pipelining in HLS.

Second, creating a new node (line 4-7) has to re-balance or sort the input list in line 7. Given the code in Listing 5.13, it is impossible to pipeline the computations in the while loop since the function *InsertToListInSortedOrder* also contains unbounded loops. Finally, the pseudo code is implemented as a recursive function using dynamic memory allocation, which is not possible in current HLS tools.

```
1 while (!isSizeOne(List))
2   Left = extractMin(List[0])
3   Right = extractMin(List[1])
4   Node = newNode(Left->freq + Right->freq)
5   Huffman->CurrentLeft = Left
6   Huffman->CurrentRight = Right
7   InsertToListInSortedOrder(Node)
```

Listing 5.13. Initial Huffman Tree Creation code as a software code

Restructured HLS Code

In this section, we present detailed restructured code that creates efficient Huffman tree based on our previous work [87]. Listing 5.14 presents the pseudo source code that targets the architecture in Figure 5.19. Figure 5.19 presents the optimized hardware architecture to create the Huffman tree. This hardware architecture store the intermediate nodes into an hardware array, BRAM.

In the software design, the Huffman tree generation code adds the newly created node to the list maintaining sorter order. The restructured code adds the new node to an empty array and increases its index every time when a new node is created in hardware architecture. This eliminates the computation needed to sort the list on every new node. In addition, the restructured code stores the Huffman tree in a data structure that allows efficient Huffman bit length calculation. After creating initial sub Huffman trees, tree information is stored in three different structures named *ParentAddress*, *Left* and *Right*. Here *Left/Right* store the symbol of left and right children. *ParentAddress* stores the address of the parent of a location where current pointer points. Using these structures, bit length can be calculated in parallel.

In this code, the *HuffmanCreateTree* function has one input, *SF*, and three outputs, *ParentAddress*, *Left* and *Right*. It defines an array in BRAM to store the intermediate nodes in line 5. Its size is $size - 1$ because a Huffman tree with *size* of leaves has $size - 1$ of intermediate nodes. Lines 6-7 define array indices in the *Left* and *Right* BRAMs. The *while* loop in line 8 iterates over *SF* data structure to create the architecture in Figure 5.19. Lines 11-16 create a left node using the current element of *SF* if $SF.F <= IN.F$ where *SF.F* and *IN.F* are current frequencies of the *SF* and *IN* arrays. Lines 17-23 create a left node using the current element of *IN* (line 20) and saves the index (leftWA) to the *ParentAddress*. In the same way, the lines 25-37 create a right node either using an

element from *SF* or an element from *IN*. The *while* loop in line 38 creates Huffman sub tree if there are intermediate nodes remaining in *IN* array. Both while loops (lines 8 and 38) can be pipelined as shown in the code since they do not contain another loops which are unbounded as in Listing 5.13.

```
1 void HuffmanCreateTree (
2     SF[size],
3     ParentAddress[size-1],
4     Left[size-1], Right[size-1]){
5 IN[size-1];
6 LeftWA = 0;
7 RightWA = 0; i,k,j=0;
8 while (i<size)
9   #pragma HLS PIPELINE
10  k = k +1
11  if (SF.F <= IN.F){
12    LeftWA = LeftWA + 1
13    Left[LeftWA]=SF[i].S
14    Freq = SF[i].F
15    i = i +1
16    }
17  else {
18    LeftWA = LeftWA + 1
19    Left[LeftWA]= n
20    Freq = IN[i].F
21    ParentAddress[j] = LeftWA
22    j = j +1
23    }
24
25  if (SF.F <= IN.F){
26    RightWA = RightWA + 1
27    Right[RightWA]=SF[i].S;
28    i = i +1
29    IN[k] = SF.F + Freq
30    }
31  else {
32    RightWA = RightWA + 1
33    Right[RightWA]=n;
34    IN[k] = IN.F + Freq
35    ParentAddress[j] = LeftWA
36    j = j +1
37    }
38 while (j < k)
39   #pragma HLS PIPELINE
```

```
40    //Create sub trees using IN
41 }
```

Listing 5.14. Restructured Huffman Tree Creation code for HLS design based on the hardware architecture in Figure 5.19.

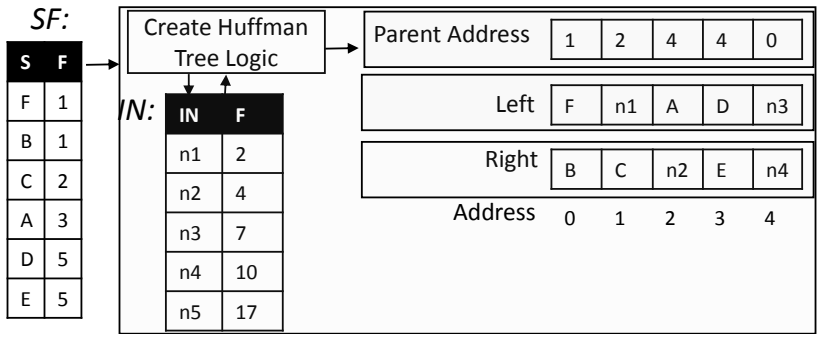


Figure 5.19. Hardware architecture of HuffmanCreateTree module: SF is an array storing symbol in S and frequency in F. IN is an array storing symbol in IN field and frequency in F field.

Discussion

HuffmanTreeCreation code is an example of *irregular kernel* that can be optimized in HLS when we write the code in a restructured way. The restructured code creates the Huffman tree in efficient way despite *while* loops that depend on data and software code that has control dependent statements.

This example shows that the optimization of a HLS design still requires hardware expertise and tool knowledge to write a program as shown in Listing 5.14. This restructured code generates more efficient hardware design based on the architecture in Figure 5.14. The software code in Listing 5.13 is more friendly and intuitive for software engineers, however, a HLS design based on the software version performs poorly due to the nature of the code structure.

Experimental Results: Huffman Tree Creation

We created a syntactic data using the LZ77 compression engine with size of 536. Designs using software code and restructured designs are optimized with HLS pragmas on top of them. We performed minimal code restructuring to the pseudo code in Listings 5.13 in order to make it synthesizable with HLS. Since the original code must sort the remaining symbol-frequency pairs, we implemented a sorting module in order to insert the newly created node into symbol-frequency table (sorted order). Table 5.2 shows area and performance results for the Huffman tree creation. Clock cycles are measured from the simulation of the design. Throughput is the number of Huffman tree creations per second. Frequency is in MHz. The first row shows results obtained by implementing the software design. The second row shows the results obtained by implementing the restructured design. The third row (Ratio) is the ratio between components of software designs versus hardware design. Larger (larger than 1) ratio means software design is bad for slices, BRAM and clock cycles. Smaller (smaller than 1) means restructured design is good for throughput and frequency. The frequency of software design is little bit better than restructured design due to limited parallelism in the software design. BRAM usage is decreased from 9 to 2 in the restructured design due to writing a restructured code that is more hardware friendly.

Table 5.2. Huffman Tree Creation.

	Area		Performance		
	Slices	BRAM	Clock Cycles	Throughput	Frequency
Software	295	9	7889921	18	145
Restructured	353	2	3142	39893	125
Ratio	0.83	4.5	2511	4.5e-4	1.16

5.2.6 Matrix multiplication

In this section, we present design and implementation of matrix multiplication.

Software Code

Listing 5.15 presents software "'C"' code for multiplying two matrices of size $N \times N$. We have to note that matrix multiplication is one of the regular kernels that is easy to optimize with HLS. We presented one of the best possible optimizations in HLS for matrix multiplication in Listing 5.15. The pipeline directive in line 10 pipelines everything under it (fully unrolls L3). Thus it needs to access arrays A and B. We used reshape directive to allow the pipeline in line 10. These optimizations create an efficient design for smaller size matrix. Once size of N grows, this design does not scale well. Next, we will discuss how to restructure the classic matrix multiplication code to create a scalable matrix multiplication.

```
1 void matrixmul( DTYPE A[][], DTYPE B[][], DTYPE C[][])
2 {
3   #pragma HLS ARRAY_RESHAPE variable=B complete dim=1
4   #pragma HLS ARRAY_RESHAPE variable=A complete dim=2
5   /* For each row i of A */
6   L1:for (int i = 0; i < N; ++i) {
7     /* For each column j of B */
8     L2:for (int j = 0; j < N; ++j)
9     {
10      #pragma HLS PIPELINE II=1
11      /* Compute C(i,j) */
12      DTYPE cij=0;// = C[i][j];
13      L3:for (int k = 0; k < N; ++k)
14         cij += A[i][k] * B[k][j];
15      C[i][j] = cij;
16    }
17  }
18 }
```

Listing 5.15. Software code of matrix multiplication.

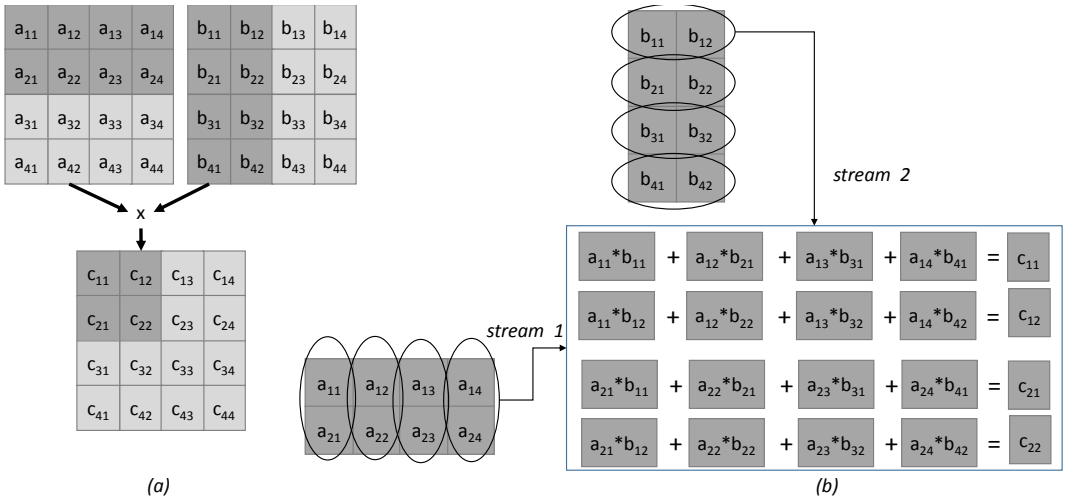


Figure 5.20. Hardware architecture for blocking matrix multiplication

Restructured HLS Code

Efficient and scalable matrix multiplication is implemented in hardware using a blocking method. Particularity, data is streamed into processing elements as a block by block. Work by Dou et al. presented streaming and blocking matrix multiplication [50]. Next, we will discuss how to implement streaming and blocking matrix multiplication in HLS. We first start from a hardware architecture. Figure 5.20 (a) demonstrates idea of blocking using an example. In this example, a block of matrix *C* is calculated using two rows and columns of matrices *A* and *B*. This allows blocking method to save the small data on local buffer and reuse it. Efficient streaming architecture is shown in Figure 5.21. The restructured source code is shown in Appendix A.2. In the code we used blockvec and blockmat structs to stream data into and out of matrix multiplication core. We always stream matrix *B* into matrix multiplication core, and we stream matrix *A* only when needed because we are reusing matrix *A*. Variable *counter* is used to determine when to read matrix *A* from the stream. In this design, we assume that data streams from/to PC. We write a test-bench to mimic the functionality of PC that sends streaming data in blocking manner. Figure 5.22 shows sequence of block streams for an example.

As shown in Figure 5.22, initially we send data from both *A* and *B*. Next, we do not send data from *A* (keep) because it is stored in the local buffer of matrix multiplication core. Testbench continues to send necessary data as shown in the rest of the Figure 5.22. Appendix A.2 also presents source code of testbench for the streaming blocking matrix multiplication.

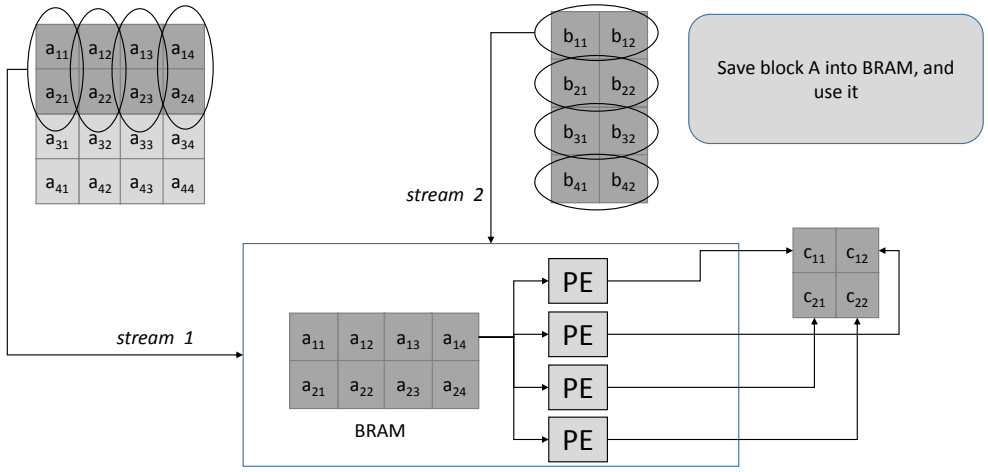


Figure 5.21. Details of hardware architecture for blocking matrix multiplication

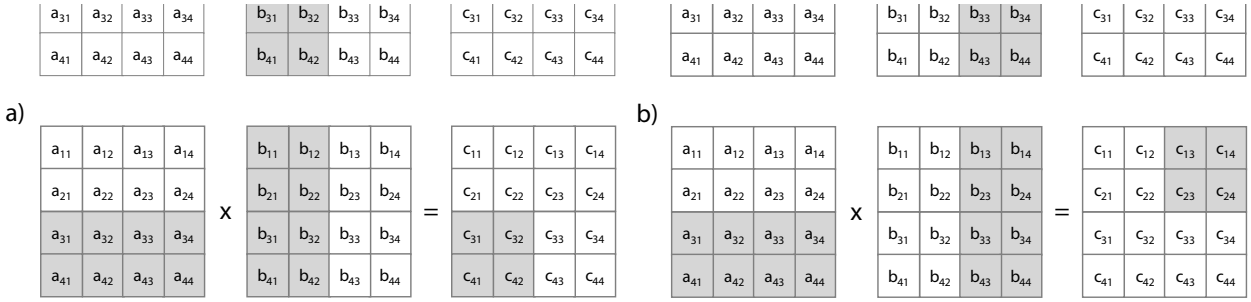


Figure 5.22. Sequences of blocks to be sent to blocking matrix multiplication

Discussion

While matrix multiplication is straightforward to optimize in HLS using pipeline and partition directives in modern high-level synthesis tools, efficient hardware based matrix multiplication is implemented using streaming architecture such as presented here [50]. Streaming architectures yields better frequency and better throughput. In general, FPGAs are good for implementing streaming designs because of the nature of the hardware. Therefore, even using HLS, it is better to create a streaming architecture in HLS.

Experimental Results: Matrix multiplication

Here we only give estimated performance results because in order to measure real performance of streaming multiplication, we must stream input data from PC (testbench). We used our testbench to emulate the behaviour of PC, and testbench sends necessary data to our streaming matrix multiplication core. We designed and implemented restructured HLS designs from Listing 5.15 on an target FPGA xc7vx485tffg1761-2 for matrix size of 32×32. Since blocking matrix multiplication is implemented as streaming, it provides better frequency and scales to larger size matrix multiplication. For example, when we synthesis the code in Listing 5.15 and restructured code Appendix A.2 for matrix size of 32×32, they achieve clock period of 4.616 ns (216 MHz) and 3.842 ns (260 MHz). Once we increase the size of matrix form 32×32 to 1024×1024, code in Listing 5.15 uses 4096 DSP48E and finishes execution in 2097168 clock cycles. This design does not fit into reasonable size FPGA. If the size of matrix is equal to 1024×1024, restructured code in Appendix A.2 achieves throughput of 1035 clock cycles. Area resources remain the same because computations in Appendix A.2 depends on size of block not size of the input matrix. While there is a communication cost associated with tiled (blocking) matrix multiplication, once initial data is received by matrix multiplication core, it constantly

outputs blocks. Thus blocking and streaming matrix multiplication achieves higher throughput.

5.2.7 Convolution

Convolution is the most common operation in many image and signal processing applications. Sobel filter is one of example of convolution. This section starts with presenting software code for a Sobel filter. Then we show how to restructure the software Sobel filter in HLS based on [13]. This restructuring is general and can be applied to any convolution kernel.

Software Code

Sobel filter convolves a given input image with two 3×3 kernels as described by Equation 5.6. For each kernel position, it calculates D_x and D_y where D_x and D_y are derivatives for x and y directions. D_x is calculated by multiplying each pixel value of the current 3×3 input image window of with its corresponding value from G_x. Then it takes the sum of the nine multiplications as a value of D_x, and repeats the process for D_y. The value $D_x + D_y$ is taken as a new value of location at the center of kernel window.

$$G_x = \begin{bmatrix} 1 & 0 & 1 \\ 2 & 0 & 2 \\ -1 & 0 & 1 \end{bmatrix} G_y = \begin{bmatrix} 1 & 2 & 1 \\ 0 & 0 & 0 \\ -1 & -2 & -1 \end{bmatrix} \tag{5.6}$$

Listing 5.16 shows one of the common ways of writing software Sobel filter in C. The two outer loops iterate over every pixel in the given input image (ignoring boundary conditions for simplicity) and the inner two loops iterate over the values of Sobel filter kernels.

```
1 int image[IMAGE_HEIGHT][IMAGE_WIDTH];
2 for(int i = 0; i < IMAGE_HEIGHT; i++)
```

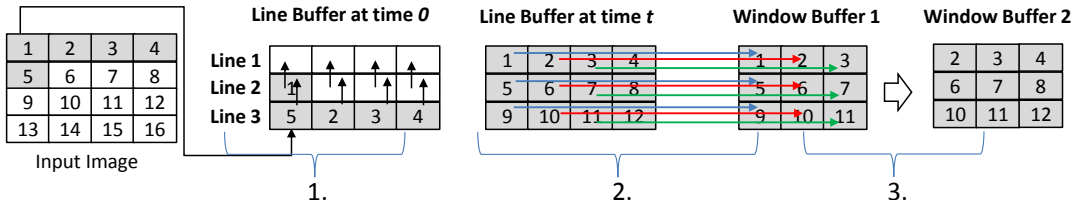


Figure 5.23. Line buffer and window buffer example for convolution operation.

```
3     for(int j=0; j < IMAGE_WIDTH; j++)
4         for(int ro = -1; ro <= 1; ro++)
5             for(int co = -1; co <=1; co++)
6                 D_x += G_x[ro][co] * image[i+ro][j+co] + ...;
7                 D_y += G_y[ro][co] * image[i+ro][j+co] + ...;
8         image[i][j] = D_x + D_y;
```

Listing 5.16. Software code of Sobel Filter.

Restructured HLS Code

We must restructure this code to generate an efficient hardware design. A common way of implementing Sobel filter in hardware is through the use of a line buffer and a window buffer. A *line buffer* is a group of memory elements that is capable of storing several lines of an input image. The number of memories, or rows in the line buffer is defined by the height of the kernel. The Sobel kernel has a size of 3×3 so we use three memories to implement the line buffer. In HLS, we use a 2D array to declare a *line buffer*, i.e., $LineBuffer[3][IMAGE\_WIDTH]$. The *window buffer* stores the values in the current window and it has the same size as the Sobel kernel (3×3). We use registers to store window buffer values in order to access them simultaneously in a clock cycle. This code is based on [13].

Figure 5.23 demonstrates how the line and window buffers work. For simplicity, we assume that the input image size is 4×4. The input image is read pixel by pixel into the line buffer. For example, pixel value 1 is copied to the first location, pixel value 2 is

copied to the second location of line buffer $line3$, and so on. While copying the input data, line buffers are shifted vertically, and the data from the most upper line buffer is discarded. After time t, the $line1$, $line2$ and $line3$ buffers are filled. Since each line buffer is implemented a a separate memory, the first *window buffer 1* can be filled by data from three line buffers in three clock cycles. The next window buffers simply discards the first column and reads the new column from the line buffers in one clock cycle, which enables the data for one Sobel filter operation to be ready every clock cycle.

The restructured HLS code for the architecture in Figure 5.23 is shown in Listing 5.17. The code from Lines 7-9 correspond to the first stage. The code from Lines 11-13 correspond to the second stage. In this stage, we design a way to read data from three line buffers to window buffers in parallel. In the last stage, we design hardware for shifting the window buffer by reading a new column from the line buffers. This process is shown in Lines 15-17. After the window buffer is filled with necessary data, we call the $sobel\_filter$ function passing the filled $WindowBuffer$ as an argument. The $sobel\_filter$ function source code is shown in Lines 21-27. The $sobel\_filter$ kernel is computed in one clock cycle using pipelining.

```
1 int LineBuffer[3][IMAGE_WIDTH];
2 int WindowBuffer[3][3];
3
4 for(int i=0; i<IMAGE_HEIGHT; i++)
5  for(int j=0; j<IMAGE_WIDTH; j++)
6   #pragma pipeline
7   LineBuffer[0][j]=LineBuffer[1][j];
8   LineBuffer[1][j]=LineBuffer[2][j];
9   LineBuffer[2][j]=image[i][j];
10
11  WindowBuffer[0][0] = LineBuffer[0][j];
12  WindowBuffer[1][0] = LineBuffer[1][j];
13  WindowBuffer[2][0] = LineBuffer[2][j];
14
15  for(int k = 0; k < 3; k++)
16    WindowBuffer[k][2] = WindowBuffer[k][1];
17    WindowBuffer[k][1] = WindowBuffer[k][0];
18
```

```
19   sobel_filter(WindowBuffer);
20
21 sobel_filter(unsigned char window[3][3]){
22 #pragma pipeline
23 for(int i=0; i < 3; i++)
24   for(int j = 0; j < 3; j++)
25     D_x = D_x + (window[i][j] * G_x[i][j]);
26     D_y = D_y + (window[i][j] * G_y[i][j]);
27     sum = D_x + D_y;
```

Listing 5.17. Restructured HLS code for the Sobel edge detection.

Discussion

The source code shown in Listing 5.17 is the restructured C code for the Sobel filter design. We used the *pipeline* pragma in line 6 to parallelize everything under line 6. In this implementation, we can fill the window buffer in every clock cycle, and process the window buffer with *sobel_function* in next clock cycle. This allows us to achieve a pixel rate of one per clock cycle. The *pipeline* pragma in (line number 7) instructs HLS to process the code below in every clock cycle. This is the optimal clock cycles achievable by manual design assuming the design processes a new pixel in each clock cycle.

Despite the fact that convolution kernels are regular kernels (automatic compiler optimizations such as polyhedral optimizations are possible) restructured code has its benefits as shown above. The source code in Listing 5.16 and Listing 5.17 have the same functionality, but result in very different hardware implementations. We can only optimize Listing 5.16 by pipelining most inner loop due to memory port limitation on *image* variable. The memory access pattern of Listing 5.16 does not allow for the outer loop to be pipelined. The restructured code from Listing 5.17 achieves the optimal number of clock cycles while the design from Listing 5.16 needs 67X more clock cycles. Clearly, the code restructuring performed in Listing 5.17 is necessary to achieve an

optimized hardware implementation.

An experienced HLS programmer would write restructured code as in Listings 5.14 and Listings 5.17 as this is a standard way to design efficient hardware. This way of thinking architecture and coding is non-trivial task for software programmers. For example, code in Listing 5.17 requires HLS programmers to think about the hardware architecture at a clock cycle level such as how data moves from input to line buffer, then shifting to window buffer.

Experimental Results: Convolution

Table 5.3 shows performance area results for the convolution designs. Software design tends to use 67 times more clock cycles than the restructured design while both designs achieving very similar frequency. As a result, the restructured design has 67 times more throughput than the software design. The software design uses less slices because of limited parallelism and does not use any BRAMs in the logic due to the nature of software code. In restructured code we stored three lines of input image to line buffers which consumes three BRAMs.

Table 5.3. Convolution.

	Area		Performance		
	Slices	BRAM	Clock Cycles	Throughput	Frequency
Software	472	0	20889601	6.2	129
Restructured	627	3	307200	417	128
Ratio	0.7	0	67	0.01	1.007

5.2.8 Face Detection

In this section, I will explain design process of implementing a face detection algorithm on an FPGA. Particularly, I will explain how to go from normal "'C"' code to

optimized restructured code for a face detection algorithm. Face detection algorithm in this section is based on the Viola and Jones face detection algorithm [130].

Software Code

Viola and Jones algorithm has two steps; first step calculates integral image of a given input image. In second stage, different scales of detection window slides over the integral image to classify each candidate region as a face or not a face. In this stage, a cascade of weak classifiers are used to decide if a candidate window is a face or not a face. Next, we present software "'C"' implementation of Viola Jones algorithm. The important part of hardware oriented face detection algorithm is calculation of integral image. Listing 5.18 presents software "'C"' code for calculating integral image of size $MAX\_HEIGHT \times MAX\_WIDTH$. Code in Listing 5.18 does not translate into efficient hardware using high-level synthesis because of several issues. First of all, current "'C"' implementation of the algorithm calculates integral image of input image using local memories. Assuming input image has size of 640×480, this software implementation must store $640 \times 480 times 8$ bits of data to local storage. This requires at least 135 BRAMs of size 18K. Second, most image processing or computer vision algorithms are implemented on hardware using a *pixel processing technique*. By pixel processing, we mean that pixels stream into the design and stream out of the design. Streaming pixels into design allows to create small local buffer (as in the case of convolution 5.2.7) to use local memory efficiently. Software code in Listing 5.18 does not written to use local memory efficiently. Next, we briefly discuss the high level hardware architecture created by Cho et al. [38], and we present restructured HLS code for the integral image calculation part.

```
1 void detectface(in_image[][], out_image[][]) {
2
3 //Integral Image calculation
4 for (i=0; i<MAX_HEIGHT; i++)
```

```
5 {
6   for (j=0; j<MAX_WIDTH; j++)
7   {
8     if (i==0)
9     {
10      if (j==0)
11      {
12        Line[j] = in_image[i][j];
13        IntegralImage[i][j] = Line[j];t
14      }
15      else
16      {
17        Line[j] = Line[j-1] + in_image[i][j];
18        IntegralImage[i][j] = Line[j];
19      }
20    }
21    else
22    {
23      if (j==0)
24      {
25        Line[j] = in_image[i][j];
26        IntegralImage[i][j] = IntegralImage[i-1][j] + Line[j];
27      }
28      else
29      {
30        Line[j] = Line[j-1] + in_image[i][j];
31        IntegralImage[i][j] = IntegralImage[i-1][j] + Line[j];
32      }
33    }
34  }
35 }
36
37 // Cascade of weak classifiers.
38 ...
39 ...
40
41 }
```

Listing 5.18. Integral image

Restructured Code

Instead of calculating integral image for each size of image, the hardware architecture fixes the sliding window. In the hardware the size of window is fixed to 21×21.

Using 21×21 sized window, it allows us to detect faces of size 21×21. In order to detect faces larger/smaller than 21×21, the input image is scaled to different sizes. Figure 5.24 presents new hardware oriented face detection algorithm. In this architecture, integral image is calculated only for 21×21 size window, and candidate window (integral image) is passed to the cascade of stages. Each cascade calculates stage sum threshold. If the calculated stage sum is above the pre-calculated stage threshold, then the window passes to the next stage. If the calculated stage sum is less then the stage threshold, the candidate fails. This means the calculation must stop for that candidate window and new window must start processing from *stage 0*.

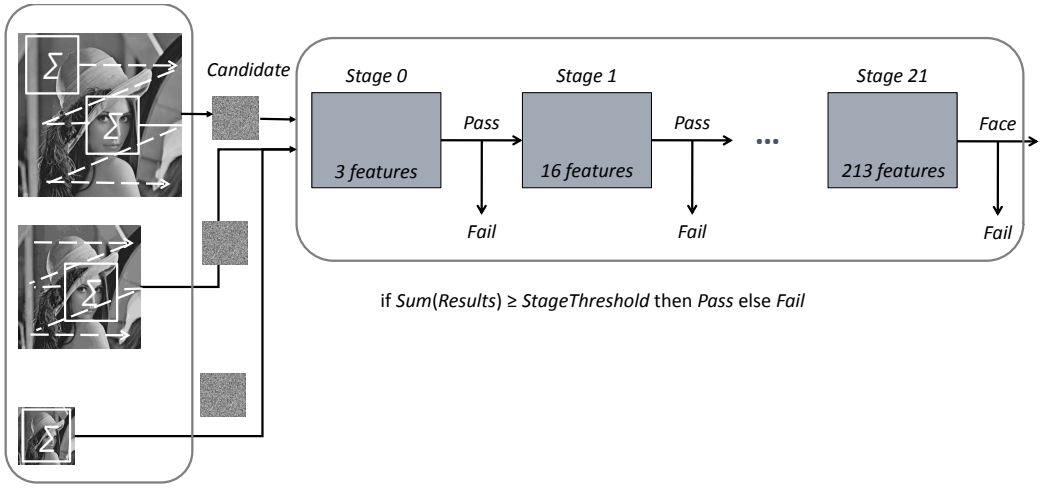


Figure 5.24. Hardware oriented face detection

Hardware architecture shown in Figure 5.24 relies on efficient integral image calculation. With this architecture, we calculate integral image for fixed size. However, if we want to calculate the integral image using the part of input image using a code like in Listing 5.18, it will take at least 21×21 clock cycles (assuming memory of input image has 1 read port). Thus we need to accelerate the calculation of integral image in hardware. Ideally, we want to create integral image every clock cycle and must achieve $II = 1$. Creating new integral image every clock cycle allows to achieve pixel rate processing.

In order to create a new integral image in each clock cycle, we designed an architecture in HLS. Our HLS architecture is the same version of architecture implemented in Verilog from the work of Cho [38]. In this architecture, integral image is created in each clock cycle by first summing pixels vertically, then by summing the pixels horizontally. We give an example that demonstrates an efficient calculation of integral image in Figure 5.25. We assume the input image has size 3×4. We create line buffers and window buffers. Line buffer and window buffer creation process are the similar to that one discussed in Section 5.2.7. Window buffer is created slightly different because integral image is the sum of pixels to left and top of location at (x,y). While copying columns of pixels from line buffer to window buffer, we must sum vertically (step 2) in order to create a vertically accumulated integral image. Then use another extra window to sum pixel in horizontal direction. This process (step 3) creates integral image for the current clock cycle. After step 1, step 2 and step 3, the architecture allows to create a new integral image in each clock cycle. For example, the next step (creating the integral image for the next clock cycle) subtracts the first column from all 3 columns, and it also copies a column (next column) from line buffers to the window buffer. This process can be done in one clock cycle.

```
1 void detectface(in_image[][], out_image[][]) {
2
3 //Integral Image calculation
4 for (j=0; j<MAX_HEIGHT; )
5 {
6   for (k=0; k<MAX_WIDTH; )
7   {
8   /*
9   This is step 1.
10  */
11  #pragma HLS PIPELINE II=1
12  LineBuffer[0][k]=LineBuffer[1][k];
13  LineBuffer[1][k]=LineBuffer[2][k];
14  LineBuffer[2][k]=LineBuffer[3][k];
15  LineBuffer[3][k]=LineBuffer[4][k];
16  LineBuffer[4][k]=LineBuffer[5][k];
```

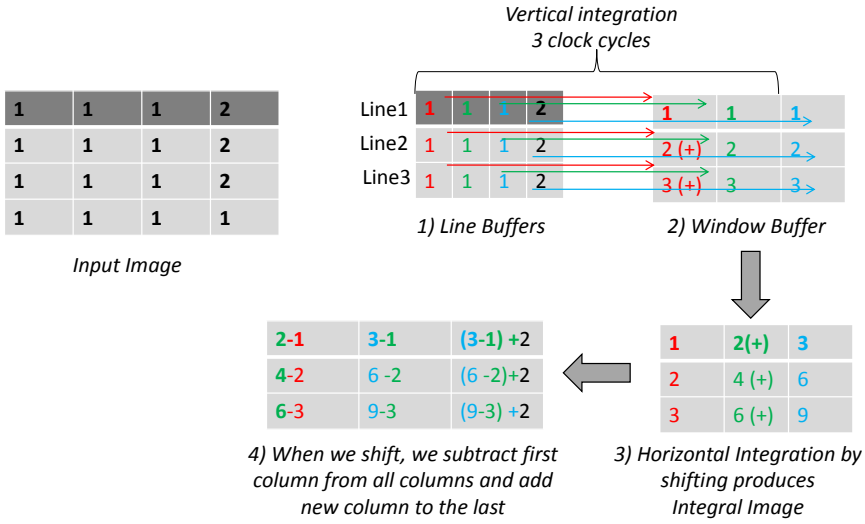


Figure 5.25. Integral image calculation hardware architecture for 4×4 size image.

```
17    LineBuffer[5][k]=LineBuffer[6][k];
18    LineBuffer[6][k]=LineBuffer[7][k];
19    LineBuffer[7][k]=LineBuffer[8][k];
20    LineBuffer[8][k]=LineBuffer[9][k];
21    LineBuffer[9][k]=LineBuffer[10][k];
22    LineBuffer[10][k]=LineBuffer[11][k];
23    LineBuffer[11][k]=LineBuffer[12][k];
24    LineBuffer[12][k]=LineBuffer[13][k];
25    LineBuffer[13][k]=LineBuffer[14][k];
26    LineBuffer[14][k]=LineBuffer[15][k];
27    LineBuffer[15][k]=LineBuffer[16][k];
28    LineBuffer[16][k]=LineBuffer[17][k];
29    LineBuffer[17][k]=LineBuffer[18][k];
30    LineBuffer[18][k]=LineBuffer[19][k];
31    LineBuffer[19][k]=LineBuffer[20][k];
32    LineBuffer[20][k]=in_image[j][k];
33
34    /*
35    This is step 2.
36    Copy pixels from LineBuffer to WindowBuffer and sum vertically.
37    */
38    WindowBuffer[0][20] = LineBuffer[0][k];
39    for(int i=1; i<II_SIZE; i++){
40      WindowBuffer[i][II_SIZE-1] = LineBuffer[i][k]+WindowBuffer[i-1][II_SIZE-1];
41    }
42
43    /*
44    Step 3 and step 4.
```

```
45   Copy pixels from LineBuffer to WindowBuffer and sum horizontally.
46   */
47
48   for (int wb_i=II_SIZE-1; wb_i>0; wb_i--)
49   {
50     for (int wb_col=II_SIZE-1; wb_i>0; wb_i--) {
51       WindowBuffer[wb_i][wb_col-1] = WindowBuffer[wb_i][wb_col];
52     }
53
54   }
55
56   //Saving the first column of integral image
57   for(int sub_col=0; sub_col<II_SIZE; sub_col++)
58   {
59     int kk=0;
60     sub_column[sub_col] = IntegralImage[sub_col][kk];
61   }
62
63   //This loop is shifting the integral window
64   for(int ii_shift=0;ii_shift<II_SIZE-1; ii_shift++)
65   {
66     for(int ii_row=0;ii_row<II_SIZE; ii_row++)
67     {
68       IntegralImage[ii_row][ii_shift] = IntegralImage[ii_row][ii_shift+1];
69     }
70
71   }
72
73   //This loop subtracts first column (sub_column) from columns right to it.
74   for(int t=0;t<II_SIZE-1; t++) {
75     for(int ii_shift=0;ii_shift<II_SIZE-1; ii_shift++)
76     {
77       IntegralImage[t][ii_shift] = IntegralImage[t][ii_shift]-sub_column[t];
78     }
79   }
80
81   //This loop creates integral image every clock cycle.
82   for(int index=0;index<II_SIZE; index++) {
83     IntegralImage[index][II_SIZE-1] = IntegralImage[index][II_SIZE-2] +
           WindowBuffer[index][II_SIZE-1];
84   }
85
86
87 // Cascade of weak classifiers using IntegralImage.
88 ....
89 ....
```

```
90 }
91 }
92 }
```

Listing 5.19. Restructured code for integral image calculation

Experimental Results: Face Detection

Here, we will present experimental results for calculating integral image using software "'C"' code and restructured HLS code. Table 5.4 presents area and performance results of calculating integral image for image size of 320×240. The *Software* represents hardware area and performance results obtained by synthesizing the code from Listing 5.18. *Restructured* means hardware area and performance results by synthesizing the code from Listing 5.19. *Restructured* uses 5 times more slices than *Software* because of existing parallelism. *Restructured* uses only 20 BRAMs while *Software* uses 258 BRAMs. *Restructured* has a throughput of 1 meaning that it can calculate one integral image in each clock cycle. *Software* must wait 921609 clock cycles to get the calculated integral image. Frequency of *Software* is also 0.44 times worse than frequency of *Restructured*. Overall, restructured code for integral image allows to create efficient face detection using high-level synthesis.

Table 5.4. Integral Image Creation.

	Area		Performance	
	Slices	BRAM	Clock Cycles	Frequency (Clock Period)
Software	295	258	921609	124 (8.022)
Restructured	1021	20	307211	278 (3.589)
Ratio	0.2	12.9	3	0.44

5.3 HLS User Study

Previous sections discussed impact of restructured code on the final generated hardware from high-level synthesis. While restructured code improves quality of hardware to be generated from high-level synthesis, writing a restructured code for software programmers is challenging task. Restructured code writing process must be considered in two different levels. First it must be considered in instruction level, and second it must be considered in task level. We discussed instruction level restructuring in Section 5.2 (e.g., prefix sum, histogram). We also give task level restructuring in Section 5.2 (e.g, FFT, convolution, face detection, matrix multiplication). In task level restructuring, the structure of different tasks and their communication plays important role in high-level synthesis. Since hardware is inherently parallel, hardware design paradigm is substantially different than software design paradigm. This paradigm shift poses challenges for software programmers when they are designing hardware even with high-level synthesis. For example, software programmers might write a code using 3-loop structure or even a recursive code for FFT which does not synthesis into hardware using high-level synthesis. Hardware efficient FFT is designed by running different FFT stages in parallel in a structured way. This kind of knowledge and expertise require software programmers to think like a hardware designer and write a code that represents parallel hardware architecture both in instruction and task level. To find out how different designers use HLS, we did a small assignment based study in class (CSE237C) in Fall 2014 quarter at UCSD. This study has two parts which are explained below:

5.3.1 User Study-1

During the class, students got hands on training regarding using high-level synthesis for 8 weeks. Initially, students did three mini assignment projects using high-level

synthesis (as part of normal class) such as FIR filter, Phase detector and DFT. For the next assignments, we divided the class into two groups (group A and group B). Then students are asked to do their fourth assignment with following conditions:

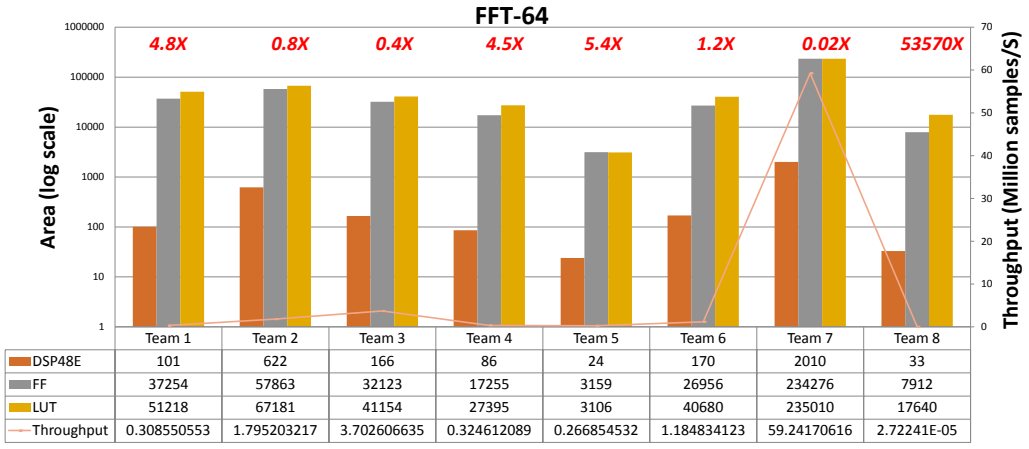
- Individuals knew FFT and sorting algorithms (explained in the class)
- They are asked to design an FPGA that fits on a device (xc7vx1140tflg1930-1)
- They are suggested to use any resources (books, papers, application notes, source codes)
- Group A was supposed to design FFT hardware for different sizes (64, 1024 and 8192)
- Group B was supposed to design sorting hardware for different sizes (1024, 16384, and 262144)
- Duration of assignment was two weeks.

Table 5.5. User Study-1.

	Size of Team		Team
	1 individual	2 individuals	Total teams
FFT (Group A)	3	5	8
Sorting (Group B)	3	6	9

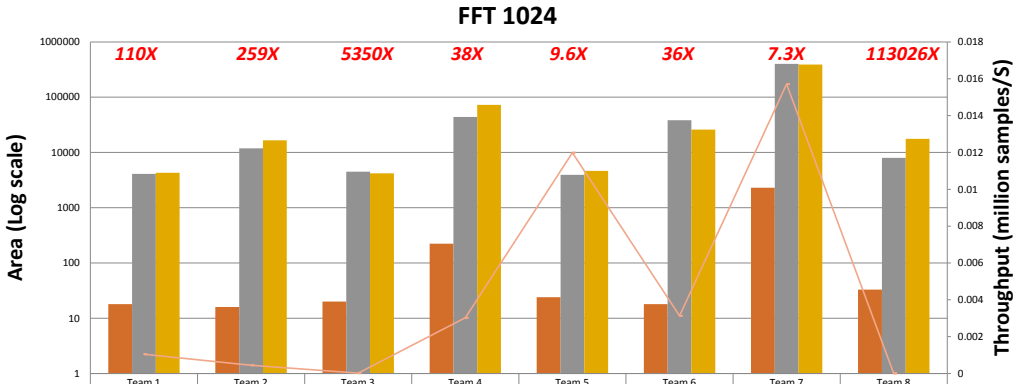
Table 5.5 presents teams and their assigned project for the first study. Figure 5.26 and Figure 5.27 present hardware area and performance (throughput) results for FFT-64 and FFT-1024. These results are from synthesis stage of HLS. We present synthesis results because 3 out of 8 teams did not get implementation (place and route) results due to long place and route time. Figure 5.28 and Figure 5.29 present hardware area and performance (throughput) results for Sorting-1024 and Sorting-16384. FFT results are

from Group A, and sorting results are from Group B. Next, we will discuss our findings. We compared these results to the one which designed and optimized by ourselves. Our HLS designs are generated from restructured code. We provided $factor = t_1/t_2$ where t_1 is throughput of our design and t_2 is throughput of each team's design. These $factor$ is shown in Figure 5.26, Figure 5.27, Figure 5.28, and Figure 5.29.



	Team 1	Team 2	Team 3	Team 4	Team 5	Team 6	Team 7	Team 8
DSP48E	101	622	166	86	24	170	2010	33
FF	37254	57863	32123	17255	3159	26956	234276	7912
LUT	51218	67181	41154	27395	3106	40680	235010	17640
Throughput	0.308550553	1.795203217	3.702606635	0.324612089	0.266854532	1.184834123	59.24170616	2.72241E-05

Figure 5.26. Hardware area and performance (throughput) results for FFT-64.



	Team 1	Team 2	Team 3	Team 4	Team 5	Team 6	Team 7	Team 8
DSP48E	18	16	20	224	24	18	2304	33
FF	4107	11852	4487	43978	3938	38231	402351	7985
LUT	4294	16547	4193	72602	4647	25985	390315	17681
Throughput	0.001046368	0.000444548	2.15628E-05	0.00302686	0.011989821	0.003131168	0.015709813	1.02072E-06

Figure 5.27. Hardware area and performance (throughput) results for FFT-1024.

Extreme optimizations: Three teams (team 2, team 3 and team 7) out of eight obtained better results than our results for FFT-64 as shown in Figure 5.26. After

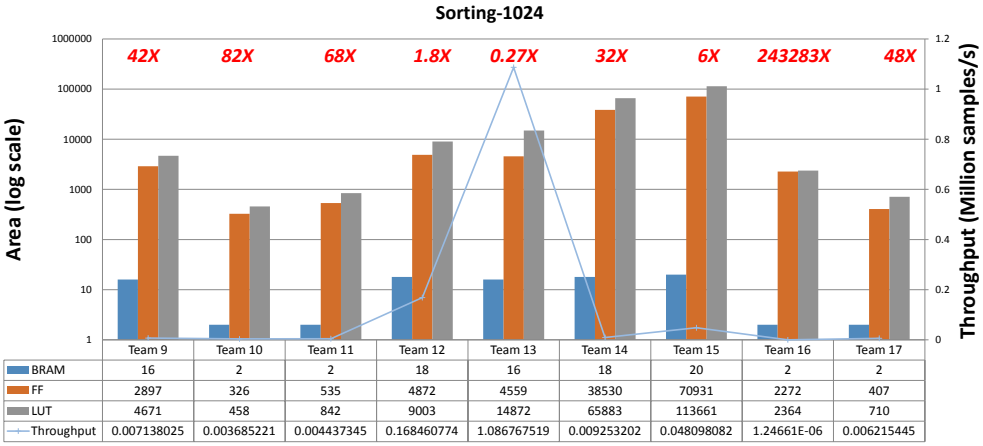
analysing their designs, we found out that their designs do not properly place and route due to *extreme optimizations* using high-level synthesis. Place and route failed because three teams (team 2, team 3 and team 7) applied optimization directives to generate fast (fully pipelined) but infeasible design. For example, pipelining and partitioning input/output array for 1024 size sorting generates 1024 registers.

Huge variations: Both for sorting and FFT, teams obtained results which have huge variations (0.02X - 53570X) for FFT and (027 - 243383X) for sorting-1024. This kind of variation resulted from unlimited or huge design space exploration capability of high-level synthesis. Since high-level synthesis allows faster but larger design space, it also increases optimizing software written in C code with high-level synthesis directives, it explores vast design space exploration. Due to unlimited (or very large) design space exploration capability provided by high-level synthesis, it becomes more difficult to find "'good"' designs among many "'bad"' designs. It becomes especially true once size of design grows larger because larger design will have more design space exploration options than smaller designs.

Unstructured software vs. Structured hardware design: In general, hardware design is a structured process. On the other hand, software design (software code writing) is unstructured process meaning that software code is usually written targeting Von Neumann architecture. For example, both FFT and sorting can be designed using recursive functions. These recursive functions are converted to nested loops when we convert it from initial unsynthesizable "'C"' code to synthesizable "'C"' code. Synthesizable "'C"' code does not necessarily express a "'C"' code that translates into an efficient hardware. For example, FFT will be written in "'C"' using 3-nested loops as shown in Listing 5.10 which does not translate into efficient hardware.

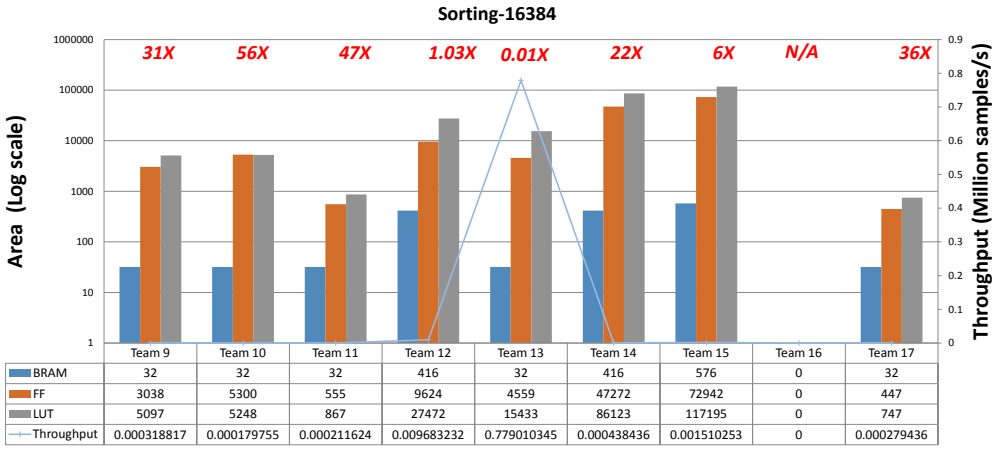
Since software code is unstructured, it is difficult to translate un-restructured code to restructured code automatically with high-level synthesis. Thus, designers must

provide hint that tells the structure of the computation by modifying the code. Next, we present results from our second study.



	Team 9	Team 10	Team 11	Team 12	Team 13	Team 14	Team 15	Team 16	Team 17
BRAM	16	2	2	18	16	18	20	2	2
FF	2897	326	535	4872	4559	38530	70931	2272	407
LUT	4671	458	842	9003	14872	65883	113661	2364	710
Throughput	0.007138025	0.003685221	0.004437345	0.168460774	1.086767519	0.009253202	0.048098082	1.24661E-06	0.006215445

Figure 5.28. Hardware area and performance (throughput) results for Sorting-1024.



	Team 9	Team 10	Team 11	Team 12	Team 13	Team 14	Team 15	Team 16	Team 17
BRAM	32	32	32	416	32	416	576	0	32
FF	3038	5300	555	9624	4559	47272	72942	0	447
LUT	5097	5248	867	27472	15433	86123	117195	0	747
Throughput	0.000318817	0.000179755	0.000211624	0.009683232	0.779010345	0.000438436	0.001510253	0	0.000279436

Figure 5.29. Hardware area and performance (throughput) results for Sorting-16384.

5.3.2 User Study-2

The summary of study is shown in Table 5.6. We switched projects between group A and group B. Different than first study, we provided list of restructured sub modules for FFT and sorting. We also asked teams to follow a certain structure, and provided an example of structured design in HLS. Students are asked to do their fifth

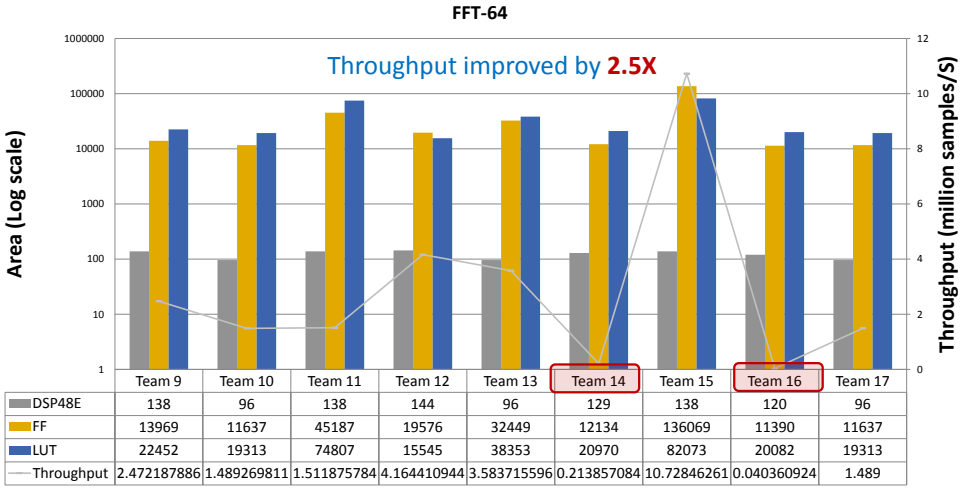
assignment with following conditions:

- Individuals knew FFT and sorting algorithms (explained in the class)
- They are asked to design an FPGA that fits on a device (xc7vx1140tflg1930-1)
- We provided list of restructured HLS codes for sub modules of FFT and sorting.
- We asked to follow a structure.
- Group A was supposed to design sorting hardware for sizes (1024, 16384)
- Group B was supposed to design FFT hardware for sizes (64, 1024)
- Duration of assignment was a week.

Table 5.6. User Study-2

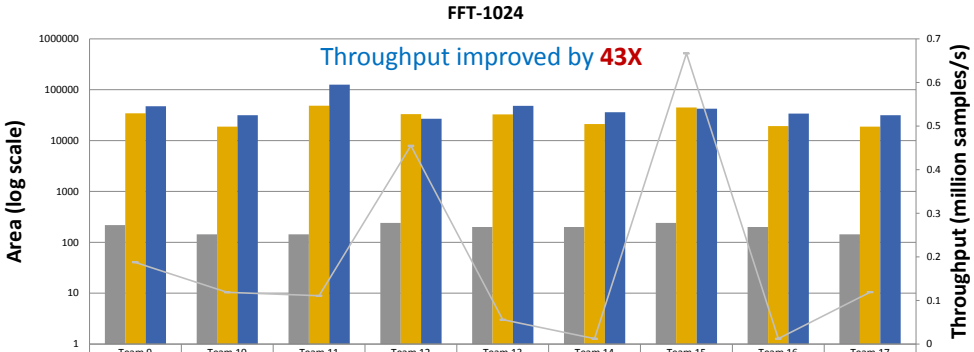
	Size of Team		Team
	1 individual	2 individuals	Total teams
Sorting (Group A)	3	5	8
FFT (Group B)	3	6	9

Figure 5.30, Figure 5.31, Figure 5.32 and Figure 5.33 provide final hardware area and throughput results for FFT and sorting. For FFT-64 average throughput increased by 2.5X, and for FFT-1024 average throughput increased by 43X. In this study, teams achieved better throughput and area results in a shorter time than study 1. The same kind of trend is also can be observed from Figure 5.32 and Figure 5.33 for sorting. Sorting-1024 and sorting-16384 throughput results improved by 2.6X and 16X.



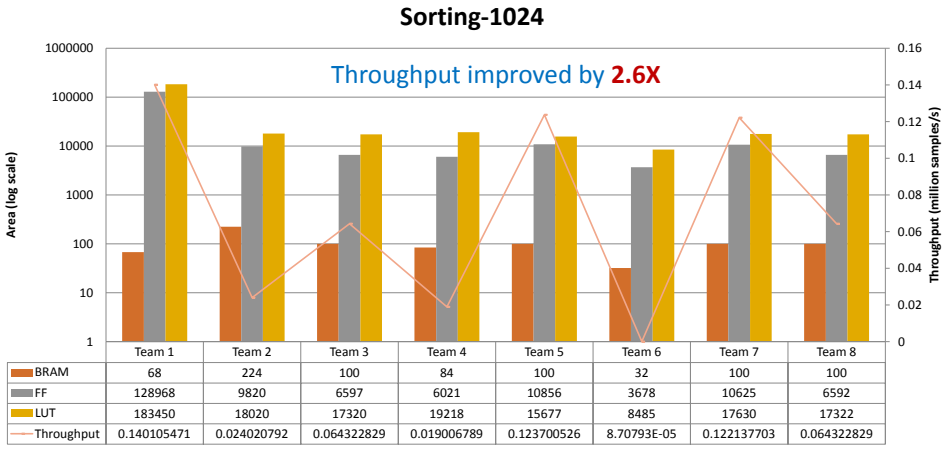
	Team 9	Team 10	Team 11	Team 12	Team 13	Team 14	Team 15	Team 16	Team 17
DSP48E	138	96	138	144	96	129	138	120	96
FF	13969	11637	45187	19576	32449	12134	136069	11390	11637
LUT	22452	19313	74807	15545	38353	20970	82073	20082	19313
Throughput	2.472187886	1.489269811	1.511875784	4.164410944	3.583715596	0.213857084	10.72846261	0.040360924	1.489

Figure 5.30. Hardware area and performance (throughput) results for FFT-64 after providing templates



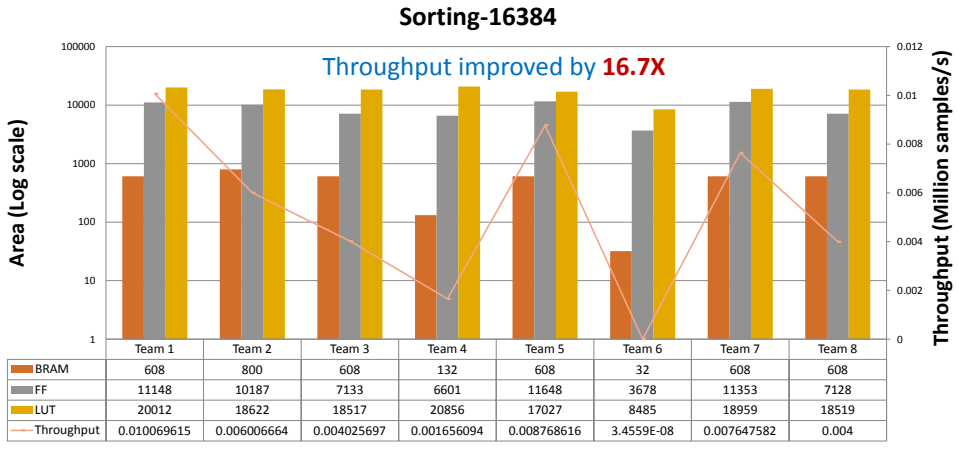
	Team 9	Team 10	Team 11	Team 12	Team 13	Team 14	Team 15	Team 16	Team 17
DSP48E	218	144	144	240	200	200	240	200	144
FF	34462	18845	48642	33156	32820	21140	44703	19314	18845
LUT	47226	31472	126018	26961	48171	36114	42345	33989	31472
Throughput	0.187286961	0.118513321	0.110797554	0.455000455	0.055940926	0.012440757	0.666560017	0.01245504	0.119

Figure 5.31. Hardware area and performance (throughput) results for FFT-1024 after providing templates



	Team 1	Team 2	Team 3	Team 4	Team 5	Team 6	Team 7	Team 8
BRAM	68	224	100	84	100	32	100	100
FF	128968	9820	6597	6021	10856	3678	10625	6592
LUT	183450	18020	17320	19218	15677	8485	17630	17322
Throughput	0.140105471	0.024020792	0.064322829	0.019006789	0.123700526	8.70793E-05	0.122137703	0.064322829

Figure 5.32. Hardware area and performance (throughput) results for Sorting-1024 after providing templates



	Team 1	Team 2	Team 3	Team 4	Team 5	Team 6	Team 7	Team 8
BRAM	608	800	608	132	608	32	608	608
FF	11148	10187	7133	6601	11648	3678	11353	7128
LUT	20012	18622	18517	20856	17027	8485	18959	18519
Throughput	0.010069615	0.006006664	0.004025697	0.001656094	0.008768616	3.4559E-08	0.007647582	0.004

Figure 5.33. Hardware area and performance (throughput) results for Sorting-16384 after providing templates

All in all, restructured design plays an important role in high-level synthesis in order to generate efficient hardware. Writing a restructured code requires substantial hardware expertise. We must consider restructured design both in instruction level and task level. Instruction level restructuring is done to remove (e.g., RAW dependency) while task level restructuring is done to allow tasks to run in parallel. Quality of hardware design generated by high-level synthesis improves when 1) high-level synthesis tool is

given restructured code for sub modules which take care of instruction level parallelism, 2) high-level synthesis tool is given structure of computation which hints overall structure of computation. Next, we address some challenges about generating restructured code and purpose a solution to make the hardware design process with high-level synthesis easy for software programmers and domain experts.

5.4 Challenges

Today's HLS tools are close to overcoming challenges of manual hardware (HDL) design. This is due to result of more than three decades of research. Despite this, HLS tools are still the domain of hardware experts. In this chapter, we studied code restructuring for different kinds of kernels (regular and irregular). We also studied code restructuring in different levels (instruction level and task level). Most software engineers are not familiar with HLS coding styles presented in Section 5.2 because it requires developers to write restructured code targeting a specific implementation and knowledge of the underlying FPGA. One way to make the restructuring easier for software programmers is through the use of *automated compiler techniques* and *domain specific HLS templates*. Automated compiler equipped with automatic parallelization and memory optimization techniques such as Polyhedral models promise to efficiently generate optimized HLS code from software code [140, 42, 14]. These kinds of models work well for generating restructured code in instruction level. However, automatic parallelization alone is not enough since some kernels require creation of efficient hardware architecture. Next, we will discuss challenges and possible solutions that make code restructuring easier for domain experts.

5.4.1 Restructured Code Generation: Instruction level

Instruction level restructuring as discussed in Section 5.2.1 poses challenges to achieve $II = 1$. One way to solve instruction level code restructuring is by designing source-to-source translator. Before synthesizing the code with high-level synthesis, we can run the code via source-to-source (S2S). One problem with this approach is that each HLS tool (vendor) has to write its own source-to-source translator. However, these source-to-source translators share a set of common rules. For example, we can make general rule for prefix sum (as following to algorithmic codes show) in the algorithmic level. Then, HLS designers must make sure their tool follows these pre-defined *rules* on the static source code and generates restructured HLS code before doing scheduling and binding.

Algorithm 5: an example algo 1

```
1 s_0 ← x_0
2 forall the i ← 1 to k do
3 |   s_i ← s_{i−1} + x_i
4 end
5 ;
6 ;
```

Algorithm 6: an example algo 1

```
1 sum ← x_0
2 s_0 ← sum
3 forall the i ← 1 to k do
4 |   sum ← sum + x_i
  |   s_i ← sum
5 end
```

5.4.2 Restructured Code Generation: Task level

Task level code restructuring is much more difficult than instruction level code restructuring for a number of reasons. First, it requires domain expertise. Designer must know his/her application well in order to extract parallelizable tasks. (e.g., FFT). Second, it requires hardware expertise. Designer must understand trade-offs between different

hardware design choices. These hardware choices are running those tasks in sequential way, running those tasks in parallel way with memory between them, or running those tasks in parallel as streaming way. Since task level code restructuring requires both domain expertise and hardware design expertise, we propose a solution that is generally applicable. In our approach, we propose to study different application domains; this study has two goals. First is to extract core kernels (computations) shared by higher level applications in that domain. Second is to identify computational structures among those core kernels in each application domain. If we know core kernels and computational structure, then it is easier to write restructured code as shown in our HLS user study. Assuming, we know the computational structure and core kernels, we can suggest a tool chain as below. In Figure 5.34, we show the overview of a proposed tool chain to allow software developers to use HLS tools more easily. In this section, we discuss some challenges to be solved in order to realize this flow.

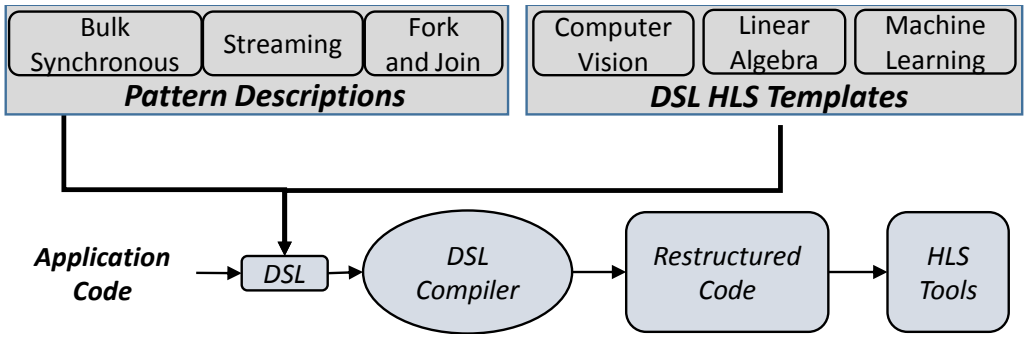


Figure 5.34. Design flow for software programmers using HLS templates (restructured code) and parallel programming patterns

We propose domain specific HLS templates to ease generation of restructured code for common kernels. Domain specific HLS templates *define an efficient hardware architecture* for certain classes of common kernels that have the same or similar computational patterns. Common kernels with the same or similar computational patterns are very prevalent in real world applications, and some research has done to classify ker-

nels according to computational patterns [16]. Current classification techniques mostly target general parallel programming practices (e.g., multi-core CPU). We can classify frequently used kernels in FPGA applications according to hardware architecture. For example, sliding window is a common architectural pattern shared both by Sobel and Gaussian filters and both map to the same hardware architecture, convolution, which can be implemented as an HLS template.

While there is no universal way to classify/extract kernels according to their computational/communicational patterns, we purpose a solution with three steps:

1. Identify common kernels from applications from a variety of applications
2. Classify these kernels according to their efficient hardware architectures.
3. Make domain specific HLS templates for those kernels based on their class of hardware architecture.

While having domain specific HLS template for every kernel is not possible, having domain specific HLS templates for the common kernels will ease the use of FPGA by software programmers. Further research is needed to identify, classify and making domain specific templates for the most common kernels for HLS tools. These domain specific templates must by tool independent and define underlying hardware architecture in an efficient way. Domain specific HLS templates are incorporated into the design flow to design kernels as shown in Figure 5.34.

5.4.3 Complex Application Design

Real world applications are complex, and almost universally contain several computational kernels. Therefore, software programmers must be able to connect and map multiple kernels of an application on an FPGA. The main challenge here is: *What is the best way connect the kernels designed with domain specific templates in HLS to*

facilitate task level parallelism? State-of-the-art HLS tools provide interface directives such as *ap_fifo* to specify a port as a FIFO, or *ap_memory* to specify a memory port. However, these kind of low level interface optimizations require hardware domain expertise. In general FPGA systems for applications such as video processing, digital signal processing, wireless systems, and data analysis rely on dataflow streaming architectures [100]. The programming model for dataflow streaming programs differ significantly from traditional processor (both CPU and GPU) implementations for software programmers. An easy programming model is needed to allow software programmers to exploit dataflow streaming architectures. In this work, we propose an approach to provide communication among kernels efficiently according to Pattern Description. *Pattern Description* represents common programming models such as streaming dataflow and bulk synchronous (CUDA programming model) based on common parallel programming patterns such as Fork/Join, Partition which are known to software programmers.

5.5 Conclusion

In this chapter, we demonstrated that HLS tools can generate high quality hardware (high performance and low area). However, this requires writing the input code in a way that reflects low level architectural hardware knowledge, which we call restructured code. In this chapter, we formally defined code restructuring and give several examples of code restructuring for regular and irregular kernels. As of today, code restructuring still remains the HLS developer's task and requires hardware expertise and domain knowledge. Code restructuring promises accessibility of FPGAs for domain experts. In next chapter, we presented an approach that promise to ease the usage of high-level synthesis by domain experts.

Acknowledgements

This chapter contains materials published in First International Workshop on FPGAs for Software Programmers (FSP 2014). Matai, Janarbek; Richmond, Dustin; Lee, Dajung; Kastner, Ryan. The chapter also contains additional materials that was added during writing of this thesis. The dissertation author is the primary investigator and author of this work.

Chapter 6

Composable, Parameterizable Templates for High-Level Synthesis

6.1 Introduction

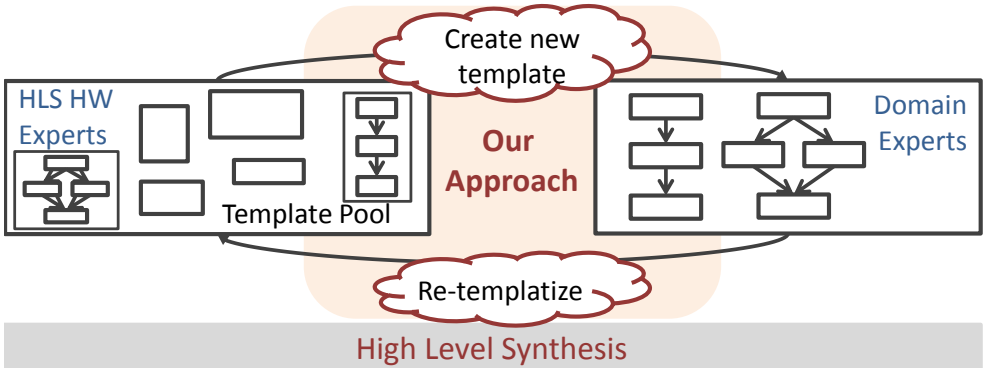


Figure 6.1. An abstraction layer that separates domain knowledge from hardware skills.

In this chapter, we present an approach to help separate domain knowledge from hardware expertise in order to create more efficient implementation of an application on an FPGA. Our approach is based on our experience building various restructured codes over 3 years of time. The importance of restructured code in high-level synthesis is presented in Chapter 5. We have seen that each domain has a number of basic kernels that share the same or similar computational primitives. This indicates that these kernels can and should be built using a highly optimized restructured code that is efficiently

translated into efficient hardware in HLS. These restructured code are developed by hardware designers that have intimate knowledge of both the domain, hardware design, and the HLS tools.

The basic building block of our approach is a *composable, parameterizable template* which are modular entities written in a restructured code. These templates are easily composed to create new templates that are automatically optimized for efficient synthesis by HLS tools. This is enabled by utilizing existing templates that follow pre described rules and common data access patterns. These composed templates are added to the template pool and can be later used to compose more complex templates. In this way, domain experts simply need to use an existing template, or form a new template for their specific applications.

The key to our approach is the ability to efficiently create hardware implementations for complex applications through hierarchical composition using a small number of highly optimized templates. These templates are composable and parameterizable, which enable them to easily, and in most cases, automatically build more complex templates. Similar to platform - based design [31], our approach is a structured methodology (based on rules and functions).

> Platform-based design theoretically limits the space of exploration, yet still achieves superior results in the fixed time constraints of the design [31].

Hardware design expertise is required at the initial stage of the process to contribute primitive templates for composition. However, we show that the number of primitive templates is small for many applications across several domains. We also show that it is possible to automatically generate efficient, high performance hardware implementations through the careful use of composable, parameterizable templates. Our method targets towards application programmers who have little hardware design exper-

tise and HLS expertise. It is also useful for HLS and/or hardware design experts as they can use or methods easily develop applications and perform design space exploration. This general process is shown in Figure 6.1.

The major contributions of this chapter are:

1. A novel approach based on composable, parameterizable templates that enables the design of applications on an FPGA by separating the domain knowledge from hardware design skill.

2. A theoretical treatment of the composibility and parameterization of templates in order to combine basic templates into more complex ones.

3. The development of basic templates across application domains, and case studies of how to compose these templates into more complex applications that meet demanding performance constraints.

The rest of this chapter is organized as follows. We continue with a discussion of motivating example. Section 6.3 and Section 6.4 formalizes the notation of a template and composition of templates. Section 6.5 presents experimental results. Section 6.6 discusses related work. We conclude in Section 6.7.

6.2 Motivating Example: Sorting

The goal of this section is to motivate the research by stepping through an example that demonstrates how small number of basic templates can be composed to create a hardware implementation that efficiently sorts large numbers. We show how two basic optimized templates, prefix sum and histogram, can be combined in a hierarchical manner to create highly efficient implementations of different sorting algorithms. First, we combine the prefix sum and histogram templates to create a counting sort template.

This in turn will be used to develop several parameterized implementations of a radix sort template. We use two data access patterns (bulk-synchonous and fork/join) to compose these basic templates into more complex templates.

We will quickly and briefly discuss the basics of the counting and radix algorithms. Counting sort has three primary kernels which are ripe for basic templates. These code blocks are: histogram, prefixsum, and another histogram operation. Figure 6.2 a) and b) shows how these histogram and prefix sum templates are used to build a counting sort template. Creating an efficient counting sort template requires functional pipelining between the three templates using a data access pattern that we call *bulk-synchronous*. We will later argue, and show in a number of examples, that this sort of functional pipelining is generalizable to a large range of applications.

It must be mentioned that it is quite important that the initial templates are optimized in a manner that enables them to be efficiently composed. While we do not have space to describe such optimizations for histogram and prefix sum, it is not simply creating a functionally correct implementation. Care must be taken to insure that they can be composed efficiently. This largely boils down to insuring that each template can be suitably pipelined. Details on how to make these subtle, but extremely important transformations can be found in [87].

Radix sort performs counting sort n times for n radix (digits). Therefore, we can compose a radix sort template using counting sort templates. Again, to develop an efficient implementation, we require functional pipelining of the counting sort templates. In this case, we can build multiple implementations of radix sort, as depicted in Figure 6.2 c) and d). Figure 6.2 c) depicts a bulk-synchronous implementation using three counting sort templates. This hierarchical composition provides on good radix sort implementation that can be added into the template library. Figure 6.2 d) provides another implementation of radix sort. Here we combine two of the previously developed radix sort templates

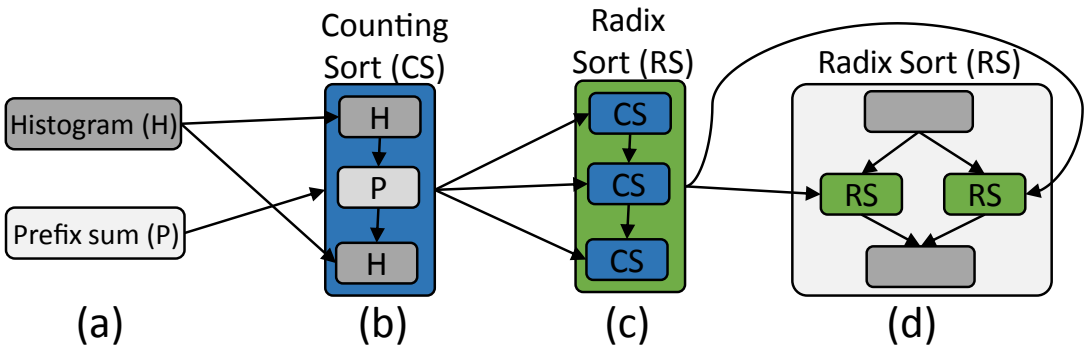


Figure 6.2. a) Initial templates for histogram and prefix sum. b) A counting sort template built using histogram and prefix sum templates. c) Radix sort built with counting sort templates using a bulk-synchronous data access pattern. d) A different implementation of radix sort composed using a fork/join data access pattern on two radix sort templates.

hierarchically using a fork-join data access pattern. This provides another option in the design space with different performance and area. Both of these can be added into the template library and later be used to compose more complex templates.

Based upon this example and our experience developing applications on FPGAs, we argue that domain programmers can efficiently build hardware implementations if they are provided with templates that are easily composable. We aid the composability by using a number of common data access patterns, e.g., bulk-synchronous, fork/join (mapreduce). By providing some optimized basic templates, and methods for automatically composing new templates in an efficient manner using these data access patterns, we show that it is possible to create highly efficient hardware implementations across a wide swath of application domains. In the following sections, we discuss the theory and algorithms behind automatic composition of templates.

6.3 Templates and Compositions

In this section, we present a theory behind templates and necessary conditions for their composition. The template compositions are done on a meta-model named *abstract templates*. The *abstract templates* is a superclass of certain templates which

have the same functionality, but with different architectures. Let us start by defining necessary properties for the *abstract templates*. We define three universal sets: T, P, and I. T is a set of the *abstract templates*, $\{t_1, t_2, ..t_n\}$. P is a set of ports for the templates where $P = \{p_1, p_2, ..p_m\}$; P_i and P_o are sets of input ports and output ports, respectively, where $P = P_i \cup P_o$ and $P_i \cap P_o = \emptyset$. I is a set of template interfaces applicable to ports, $\{i_1, i_2, ..i_n\}$.

Definition 1 *A port is a tuple $p = (d, i) \in P$, where d is a direction of in/our port and $i \in I$. We use $d(p)$ and $i(p)$ to represent the direction and interface of the port p.*

Templates are composed based upon their *port* properties and allowed structural patterns. Patterns will be discussed later. The *Definition 2* defines port properties which are defined by forward/backward (FC/BC) compatibility which are defined below. The *Definition 3* defines templates and their composability properties based upon forward/backward compatibility.

Definition 2 *(Forward/Backward Compatibility) For $\forall p_1, p_2 \in P$, if $d(p_1) == out \wedge d(p_2) == in \wedge i(p_1) == i(p_2) \Rightarrow FC\ (p_1, p_2) = 1$. If $d(p_1) == in \wedge d(p_2) == out \wedge i(p_1) == i(p_2) \Rightarrow BC(p_1, p_2) = 1$.*

Definition 3 *An abstract template is a tuple $t = (IN, OUT, f) \in T$ with following properties: **1)** $IN \subset P_i \wedge OUT \subset P_o$, **2)** $|IN| \geq 1 \wedge |OUT| \geq 1$, **3)** $f(IN) = OUT$.*

$IN(t)$, $OUT(t)$, and $f(t)$ represent a set of input ports, a set of output ports, and the functionality of t, respectively. Abstract template is an *abstract class* used for composition purposes. Actual architectures are represented by *optimized architectural instance templates* which are subclasses of abstract template. The *abstract template* $t \imath n T$ can have multiple variants of *optimized architectural instance templates* which has the same functionality and ports with the abstract template t. We use a set $A_I^{t_i} =$

$\{t_{ij}|t_{ij}$ *is instance templates* $\}$ where $i = 1,..k$ to denote k instance templates of an abstract template t_i. In the rest of the paper, we use *instance template* to refer optimized architectural instance template and *template* to refer both abstract and instance template. An instance template t_{ij} is a tuple $t_{ij} = \{II, a\}$ where II is the throughput of t_{ij} and a is the area of t_{ij}. We use $II(t_{ij})$ and $a(t_{ij})$ to represent throughput and area of instance template t_{ij}.

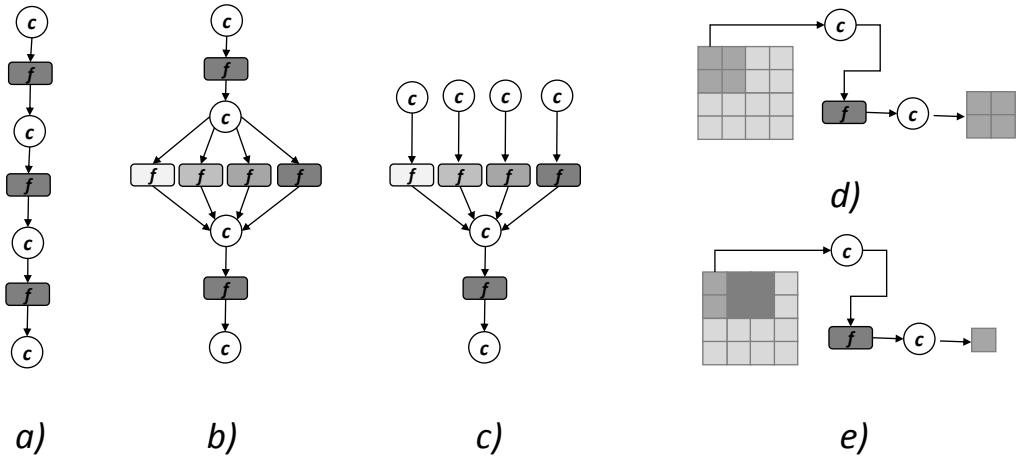


Figure 6.3. Patterns. a) Bulk synchronous, b) Fork/Join, c) Merge, d) Tiled computation, e) Sliding window. Symbol *c* refers to channel. Channel is a communication medium such as a fifo or a memory.

Now we define rules and functions that must hold in order to compose two or more templates to form a new template. Template composition is done using patterns. In this work, we consider patterns shown in Figure 6.3. Tiled computation and sliding window are explained in Chapter 5 and they are defined as computational patterns. Merge, Bulk Synchronous and Fork/Join are structural patterns. Merge pattern follows from fork/join pattern. Thus, we will give formal rules for bulk synchronous pattern and fork/join pattern. We use $BSComposability$ and $FJComposability$ functions for bulk synchronous and fork/join patterns. $BSComposability$ and $FJComposability$ functions are main blocks to check if two or more templates can be composed to form a new

template based on bulk-synchronous or fork/join data access patterns, respectively. In the following we will define these two functions. In order to define bulk-synchronous composability of $t_1, t_2, ..t_n \in T$, we check the composability of every consecutive pair of templates using a binary composability property, *BinaryComposability*, which is defined below. Based on *BinaryComposability*, we define the *BSComposability* function.

Rule 1 *A binary composability function is defined for* $\forall t_1, t_2 \in T$ *if 1)* $|OUT(t_1)| = |IN(t2)|$*, 2)* $FC(p_i, p_j) = 1$ *for* $\forall p_i \in OUT(t_1)$ *there exist only one* p_j *s.t* $p_j \in IN(t_2)$*, then we say that* $BinaryComposability(t_1, t_2) = 1$

Rule 2 *The function BSComposability is defined as* $BSComposability(t_1, t_2, t_3, ..t_n) = 1$ *if* $\forall\ t_i, t_{i+1} \in T$ $BinaryComposability(t_i, t_{i+1}) = 1$ *where* $i = 1, n-1$

In order to define fork/join composability of $t_1, t_2, ..t_n \in T$, there are two constraints. We assume t_1 is the fork, and t_n is the join. Then the fork join composability function, *FJComposability*, is defined by checking *ForkComposability* and then *JoinComposability* rules.

Rule 3 *The function ForkComposability is defined as* $ForkComposability(t_1, t_2, ..t_n) = 1$ *where* $\{t_1, t_2, t_3, ..tn\} \in T$ ***if 1)*** $|IN(t_1)| == 1 \wedge |OUT(t_1)| == n-1$, ***2)*** $|IN(t_k)| = |OUT(t_k)|$, $\forall k = 2, n$, ***3)*** $\forall p_i \in OUT(t_1)$ *if* $\exists p_k \in IN(t_k)$ *such that following conditions hold:*

$$ForwardCompatibility(p_i, p_k) = 1 \wedge BackwardCompatibility(p_k, p_i) = 1$$

Rule 4 *The function JoinComposability is defined as* $JoinComposability(t_1, t_2, ..t_n) = 1$ *where* $\{t_1, ..tn\} \in T$ ***if 1)*** $|IN(t_i)| == 1 \wedge |OUT(t_i)| == 1$, $\forall i = 1, n-1$ $|IN(t_n)| == n-1$, ***2)*** $\forall p_k \in OUT(t_i)$ *for* $i = 1, n-1$ *if* $\exists p_j \in IN(t_n)$ *s.t.* $ForwardCompatibility(p_k, p_j) = 1 \wedge BackwardCompatibility(p_j, p_k) = 1$

Rule 5 *This rule states that* $FJComposability(t_1,t_2,t_3,..t_n) = 1$ *is where* $\{t_1,t_2,t_3,..t_n\} \in T$ *if* $ForkComposability(t_1,t_2,..t_{n-1}) == 1 \wedge JoinComposability(t_2,t_3..t_n) == 1$. *Next, we define sub rule of FJComposability. For a template t, if* $IN == IN_1 \cup IN_2,..IN_n$, $OUT == OUT_1 \cup OUT_2,..OUT_n$ *and* $\exists f(t)$ *such that* $f(IN1) == OUT1, f(IN2) == OUT2,..f(IN_n) == OUT_n$, *then t is n-way FJ composable.*

Based on definitions and rules above, we define template composition functions. Template composition is a way of structurally composing existing templates from T based on data access patterns such as bulk-synchronous and fork/join. We omit the proofs here for the sake of brevity.

Lemma 1 *if* $BSComposability(t_1,t_2,..t_n) == 1$ *for* $\{t_1,..tn\} \in T$, *then* $\Rightarrow BS(t_1,t_2,..t_n)$ *maps to a new template* t_{new} *where* t_{new} *has all properties defined in Definition 3.*

Lemma 2 *if* $FJComposability(t_1,t_2,..t_n) == 1$ *for* $\{t_1,..tn\} \in T$, *then* $\Rightarrow FJ(t_1,t_2,..t_n)$ *maps to a new template* t_{new} *where* t_{new} *has all properties defined in Definition 3.*

BS and *FJ* functions are ways of constructing a new template t_{new} for a new functionality, which later can be used as an existing template. We also define a *general strategy* to parameterize any code block. If there is a functionality which does not have corresponding template in T, we rely on users making a new template and contributing it to our system. The users add the new template to the system by defining the abstract properties of the template. The properties are derined in a standard xml format to be used for automation. Loop optimization works at [34, 140, 64] can be used here. This area of research needs further investigation. We believe that our approach, as in Figure 6.1, eventually fills this gap by producing more and more templates in a disciplined manner.

6.4 Template Parameterization

In this section, we describe how to compose templates in a highly optimized manner and provide trade-offs on performance and area. When composing new templates based on the rules defined in previous section, we have two constraints: 1) composition algorithm, 2) parameterizable architecture generation.

6.4.1 Composition Algorithm

A domain expert is designing an application A with n kernels, i.e., $A = \{k_1, ..k_n\}$. Assume that there exists at least one template that can be used to implement each k_i. The input to the algorithm is a set of templates T, user input data, and an optional user constraints UC. UC is a tuple $UC = (f_u, II_u, a_u)$ where f_u is frequency, II_u is throughput, and a_u is area. The area, a_u, is considered as a weighted combination of BRAMs and LUT/FFs. Next, we present an algorithm for constructing a new template using bulk-synchronous function (*BS*) in Algorithm 7. Due to limited space, we only present an algorithm for *BS*. The same principle and algorithm applies to FJ function using Rule 5.

The algorithm has four sub routines. The $FindInstances$ calls $GetAllInstance$ sub routine for each abstract template t_i. The $GetAllInstance$ returns a set M_A containing optimized instance templates. As disscuessed in previous section, abstract template is a black box, and each abstrat template has a number of instance templates. This is because in our framework, we want to seperate functionality from the underlying microachiectural hardware, and letting our framework choose the one based on user constraints. For example, as shown in Figure 6.4, matrix multiplication abstract template has a number of instances. Each instance is implemented in different microarchiecture (streaming, 1 processing element (PE), 4 PE with streaming) having different performance and area based on user constraint. Based on user constraint (e.g., input data size), the algorithm

selects different instance templates. This is importan because some applications have a intersection points between different instance templates where certain instance template is better than another around that point. We call it performance breaking point. This will be disscussed in more detail experimental section.

After *FindInstances* routine, M_A set contains all instance templates necessary to implement an application A. M_A has a matrix-like structure where column i represents the same class of templates that can be used to implement kernel k_i. To illustrate this process better, we give an example in Figure 6.4 (a). The k_1, k_2, k_3, and k_4 are kernels which can be implement by *abstract templates* t_1, t_2, t_3, and t_4. In the next step, we call subroutine *ComposabilityCheck* which returns a set of graphs (each graph contains a set of tempaltes composable based on *BS* model). The routine checks Rule 1 for each t_{ij}, t_{ij+1} pair, and Rule 2 for the selected set of t_{ij}. In the case of fork/join, we use Rule 5 to check composability. After this step, we obtain one or more sub-graphs of M_A as shown in the Figure 6.4. The optimal algorithm (maximizes throughput) to find instance templates runs checks all possible paths in each graph. The optimal algorithm runs $O(n \times k \times k)$. Currently, we do greedy algorithm which selects a graph that has t_{1i} where the $II(t_{1i})$ is minimum. The result of this algorithm returns a graph G which starts from a selected t_{1i} as a source. The next step, *ConstructBulkSynchronous*, accepts input G and outputs a path from a source of G to a sink of G. This procedure returns path that contains a set of instance templates for the given application A based on *BS*. In this process, we consider two cases; When $UC = \emptyset$, the algorithm selects each next instance template greedily which maximizes throughput. If $UC \neq \emptyset$, then we model the selection as a a cost function using *closest point problem* [115] between UC and a set of candidate instance templetes. The function *GetAllComposableTo* returns all composable templates from the current vertex v_i. For example, in Figure 6.4, if we are on t_{14} of G_2, then *GetAllComposableTo* retuns t_{22} and t_{24} The *CalculateClosestPair* function calcualtes

cost from the current vertex v_i to all other vertices returned by *GetAllComposableTo*. The next instance template t_{ik} is selected based on a value of closest point between pair of (II_a, a_u) and a $II(t_{ij}), a(t_{ij})$ where $t_{ij}$is a set of all candidate instance templetes. This process is performed in *VertexWithMinCost* function. Based on UC, if a certain template fall to meet II_u, we apply *parameterized template generation and selection*, which will be discussed in Section 6.4.2. The final step, *CodeGeneratorBS*, generates optimized HLS code. This will be discussed in Section 6.5.

6.4.2 Parameterization

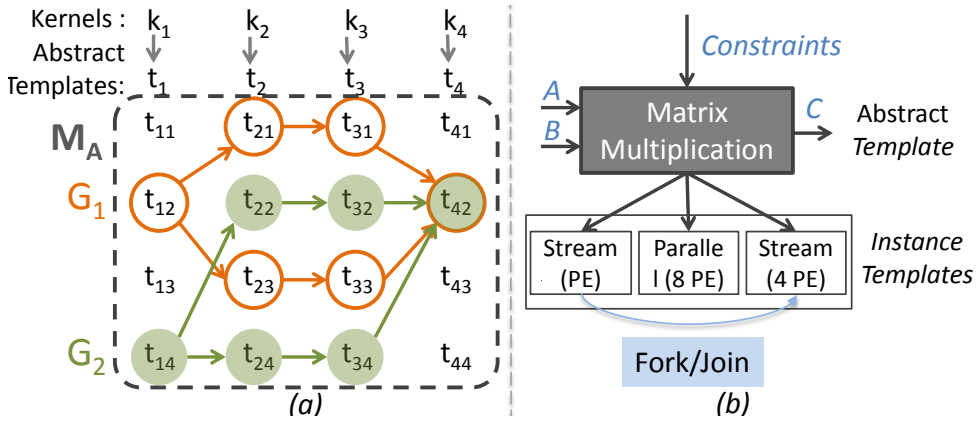


Figure 6.4. Composition example. a) Bulk-synchronous model, b) An example relationship between abstrat and instance templates.

Instance templates allow users to have parameterizable architectures. This enables instance templates to provide flexibility that leverages area and performance trade-off by providing different instances of an abstract template. The flexibility of instance templates provides two benefits: 1) adjusting throughput to global throughput or to user specified constraints when composing template, 2) template selection based on user constraints or user input data size. Both of these benefits are crucial when composing templates. The former benefit provides easy way to achieve throughput increase/decrease based on user constraints. We name it *parameterizable speed-up* in this paper. The latter benefit

of instance template is essential to provide optimized architecture for the user, which selects certain architecture based on user constraint or user input data. For example, *RS*1 and *RS*2 in Figure 6.4 (b) are the same instance templates for radix sort that represent the architecture as in Figure 6.2 (c). Based on different parameters and user input data size, *RS*1 and *RS*2 has different throughputs for a given data size as shown in Figure 6.4 (b). We will discuss this example in detail in Section 6.5. Currently, this process of selecting optimized architecture for specific user constrain is being done manually by HLS experts. With our approach, we will automate this process by leveraging user constraints and analysing user input data. Since templates have pre-defined high-level structures, throughput, $II(t_{ij})$, and area, $a(t_ij)$, are linear functions of input data size. They can be determined differently for regular and irregular programs; for regular programs, $II(t_{ij})$ and $a(t_ij)$ are defined by exploiting the user input data and instance template structures. For irregular programs such as sparse matrix vector multiplication, we rely on design space exploration to determine $II(t_{ij})$ and $a(t_ij)$.

6.5 Experimental Results

We used Vivado HLS 2014.1 as a back-end HLS tool. Our approach can be easily modified for use with another HLS tool. Here, we present an approach that shows composing templates based on an abstract template. This kind of composition can be easily done using existing scripting languages such as Python. Each instance template is a Python class inherited from an abstract template class. For example, *prefixsum* abstract class identifies the functionality of prefix sum. Implementation classes inherit from abstract prefix sum class and implements different versions of prefix sum hardware architecture. An abstract template class can consists of fields to model ports, interfaces, and functionality for its child classes. In this work, we define interfaces based on Vivado HLS interface specification [10]. The abstract template implements the

HLSCodeGenerator function which writes HLS code using pre-defined structure. Each instance template inherits this method and calls it with template specific parameters, e.g., optimization parameters, bit width, size, number of functional units, etc. By default all templates include the dataflow directive. When a template is composed using existing templates, the *HLSCodeGenerator* subroutine traverses each instance template and generates separate HLS code for each of them. Then it generates a top-level function which calls the generated instance templates. The dataflow directive is placed in top level function which functional pipelining in both *BS* and *FJ* data access patterns.

Next, we present area and performance results of different primitive templates. We use *Template* and *OSC* to indicate code generated from our method and *optimized synthesizable code* for HLS, respectively. OSC is a HLS code *highly optimized by HLS expert using HLS #pragmas*. This codes is not rewritten to target low level architectural features. It is code written optimized for software and using HLS pragmas. *It is how a good software programmer would write the code and use the HLS tools based on our experience in teaching HLS over three years of a graduate student class*. On the other hand, *Template* is generated using our parameterizable templates. In the following, we start by comparing *Template* against *OSC*. Then, we present two examples of achieving parameterizable speed-ups using templates. Finally, we use several of those highly parameterizable templates to design five large applications. Due to space constraints, we present only few results of applications designed using templates. The applications are Canny edge detection and different kinds of matrix inversion. All results are obtained from place and route. The target device is Zynq.

Template vs. OSC: Figure 6.5 shows throughput of *Template* and *OSC* designs for various templates designed. Level 0 is a primitive template and includes the *0prefsum, 0histo, 0gaussian, 0conv, 0hufftre, 0thrh, 0Imgadjt, 0bicubic, 0dilation, 0erosion, 0bit_rev, 0butter, 0SpMV_0* kernels. For templates *0dilation, 0erosion*, and *0thrh* kernel, *Template*

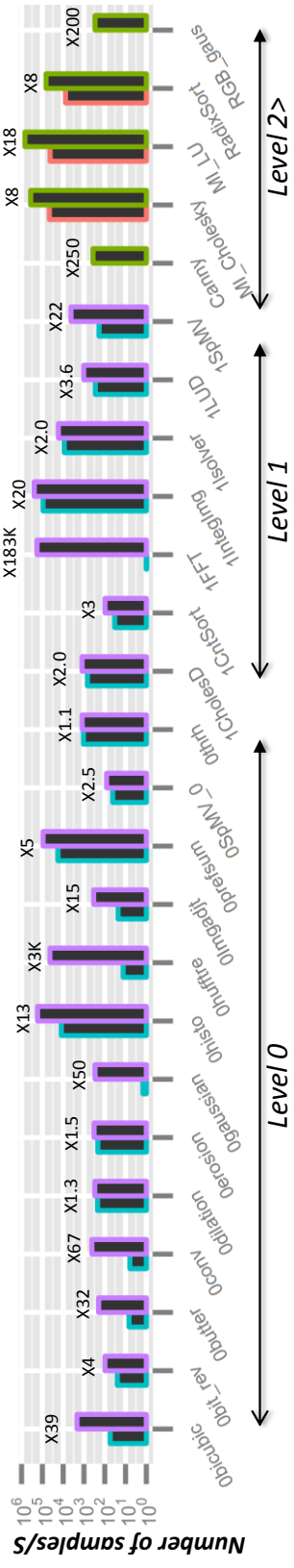


Figure 6.5. The OSC results are always on the left and the Template results on the right. In each application we give the actual speedup above the bars. The Template approach is always better than OSC. The "level" indicates the maximum amount of hierarchy used for the templates. Level 0 is a primitive template. Level 1 is composed of at least one primitive template. Level 2 is composed of at least one Level 1 (or higher) template.

is better than *OSC* by around $1.1 - 1.5X$. This is because those templates can be highly optimized using only HLS only directives. For kernels *0gaussian, 0conv, 0bicubic, 0bit_rev, 0butter* and *0hufftre*, we see several orders of magnitude of improvement. This is because these templates require low-level micro-architectural knowledge in order to generate efficient hardware. The second level kernels are designed using the templates from the *Level 0*. For example, *1CntSort* is built on using *0histo* and *0prefsum*. *FFT* is built using *0bit_rev* and *0butter*. *1SpMV* is built by composing several of *0SpMV_0* templates using the fork join data access pattern. Several templates in Level 1 use linear algebra primitives such as vector-vector multiplication. Since linear algebra primitives are easier to design in HLS than others, we omitted their results from Figure 6.5. For example, *1IntegImg* is built by composing *0conv* and vector subtraction. Level 2 are five applications composed using existing templates from Level 0 and Level 1. We will discuss applications later. Next, we will elaborate parameterizability of our templates.

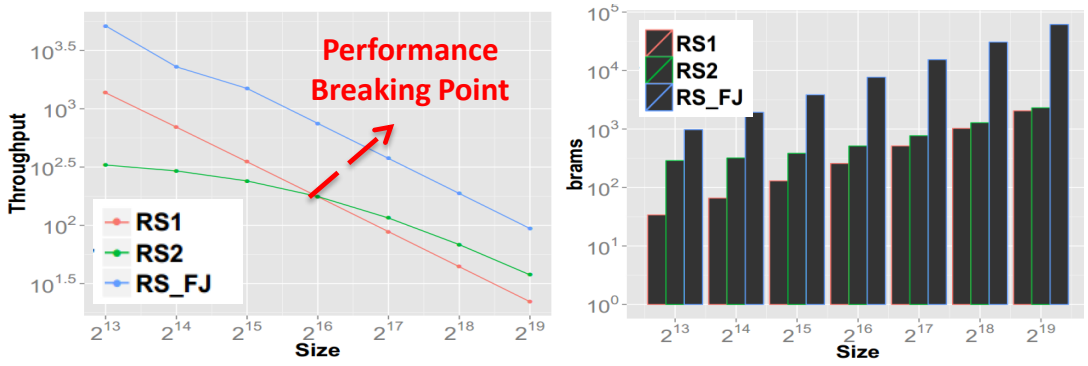


Figure 6.6. (a) Performance graph showing performance breaking point (PBP). Templates provide a means to select right tempalte based on PBP. (b) Area

Parameterization: Parameterization plays a vital role in Algorithm 7 when composing templates to meet a throughput requirements. Here we describe the parameterizable templates for prefix sum and histogram. The result is shown in Figure 6.7. First, we optimized both of them targeting low level hardware architecture by removing data and read after write dependencies. This is same as the template in Level 0 in Figure 6.5.

We call this *Lev*0. Then using these *Lev*0 designs, we applied different combinations of parameterizable speed-up factors using FJ and BS data access patterns. The prefix sum is composed based on FP pattern while histogram is composed based on FJ pattern. The speed-up factor (shown as *Factor* in the Figure 6.7) is the unrolling factor for OSC and *Lev*0 designs. $OSC-X$ means speed-up factors of X. The same convention follows for *Lev*0 designs. For BS and FJ designs, the speed-up factor is the task level parallelism factor. In both cases, OSC designs does not give the desired throughput regardless of unrolling factor. In the *Lev*0 designs, the throughput does increase, but it does not scale as expected. This is because the clock frequency is also increasing with higher speed-up factor. BS and FJ, both designs perform and scale as expected according (4, 8, 16) to speed-up factor. Algorithm 7 selects the *histogram template with fifo* interfaces as parallel task of FJ.

Next we present three different radix sort templates in Figure 6.6. $RS1$, $RS2$ are templates composed as in Figure 6.2 (c) with different parameters, and RS_FJ is a template as in Figure 6.2 (d). Sorting algorithms use less slices, and usaully BRAM is important area metric. Thus, in the Figure 6.6 (b), we presented throughput and BRAM utilization. $RS1$ and $RS2$ have similar area usage, and RS_FJ has larger area (2-8 times) usage than $RS1$ and $RS2$ due to higher parallelism. In this case if UC is maximizing throughput with minimum area. Our algorithm transparently selects an archeicture based on user constraint balancing performance breaking point. Algorithm 7 selects $RS1$ for input data size $2^{13}-2^{15}$ based on performance breaking point, and it selects $RS2$ for input data size $2^{16}-2^{19}$ as shown in the Figure 6.4. If UC is empty or maximum throughput, the algorithm selects RS_FJ.

Canny Edge Detection: Next, we argue that the hardware generated from our approach has competitive area and performance results. We compare area and performance of applications composed with templates with other published work. We use two

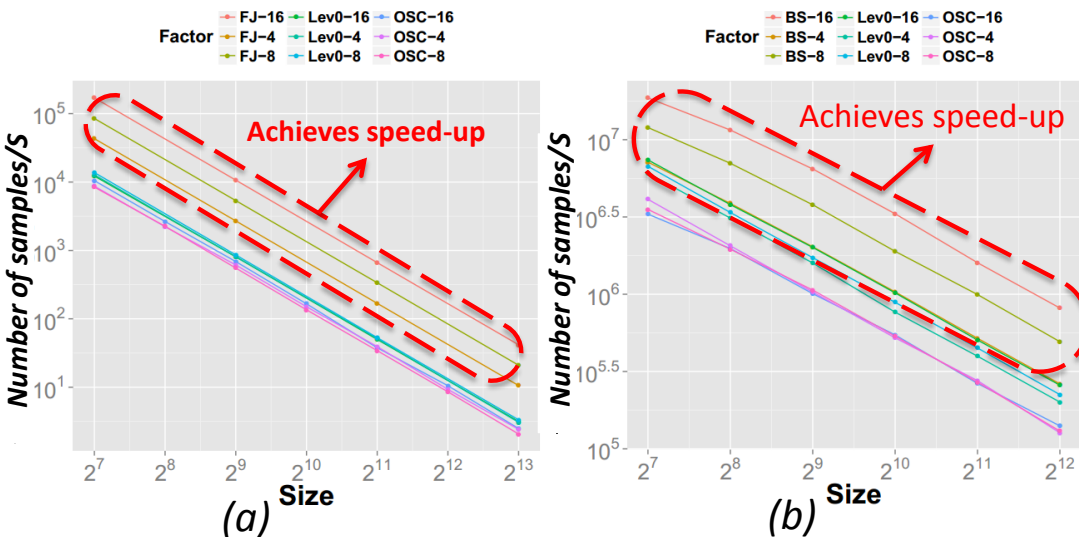


Figure 6.7. Developing parameterizable templates for a) Histogram, b) Prefixsum

cases: Canny edge detection and matrix inversion. The Canny edge detection algorithm is divided into four stages, Gaussian smoothing, edge strength identification, non maxima suppression, and double thresholding. All four stages can be designed using highly parameterizable convolution and histogram templates from our template pool. We composed a new template for Canny edge using *BS* function. The generated code of Canny edge runs for Q/VGA sizes. Table 6.1 shows the throughput as a frames per second and hardware utilization for our designs and previous work. Our results are comparable to these published application papers.

Matrix Inversion: Three matrix inversion templates are composed using *FJ* LU and Cholesky decomposition, and linear algebra templates. Table 6.1 shows a comparison between our results and previously published works [70, 74] for 4×4 matrix inversion. For the sake for comparison, we implemented our designs to a Virtex4 device. In general, our performance is 7-18X better than [70, 74] but our area is 2-7X larger than those works. Our design uses 32-bit numbers while [70] is using 20 bit numbers.

6.6 Related Work

Several HLS vendors provide libraries, e.g., OpenCV and linear algebra from Xilinx. They provide a first step towards making FPGA designs more accessible. While experienced HLS users find these libraries useful, it is difficult for a domain programmers to use them since they require low level hardware expertise. Our technique goes further than just static libraries; these libraries are typically not composable or parameterizable. In fact, we can use the functions in these libraries as basic templates. The work [39, 90] present similar approaches to this work terms of facilitating composability of reusable components. The work [39] defines accelerator building block as a service for hardware blocks and the work[90] presented a study of IP core composition. Both of these works compose low level IP cores. Therefore, our approach can provide functionality to them by generating composable IP cores or accelerator blocks. Several other works such as Chisel [18], FCUDA [105] and others [57, 121] present domain-specific language based approach to design an FPGA system. This work is orthogonal to our research. For example, we can use HLS code generated from [57, 121] to make new templates and use them in our framework.

BALBOA [51] and subsequent works [123, 117] present component composition and theoretical framework for the system-level design. These works focus on much wider compositional framework than our approach. In our work, we restrict the design space exploration with known best design practice templates and patterns. We believe restricting the design space exploration with known templates and patterns eventually allow to generate efficient hardware from high-level synthesis.

System level design automation [8] and compositional high-level synthesis [48] present an approach to select hardware components while doing inter/intro optimizations among components. The main building blocks (or assumptions) of these works

are existing components. These components, in fact, can be modelled as composable parameterizable templates. Thus our work can be used as a components for [8, 48]. Different than these works, we assume the users of our work will be pure software programmers without any hardware knowledge. Thus, our work provides higher-level of abstraction by composable parameterizable templates. Our work also does not provide direct component descriptions in (VHDL, Verilog, SystemC, or C/C++), we provide meta description of templates which are flexible to apply different compositional (FJ, BS) functions. This abstraction allows our work to be part of existing works such as [8, 48], or use the existing works to compose hierarchical templates.

Limitations: The success of this work is dependent on building enough primitive templates to cover all applications. Ideally, users contribute to the system by providing their templates. Instead of relying on users, we take a domain-specific approach where we studied certain domains such as database operations, image processing, linear algebra, and compression. For example, there are a handful of primitive database operations that are needed as templates. Sorting is a one such primitive. There are many methods for sorting including sorting networks, bitonic sort, and fifo-based merge sort. When we have a composable templates for these sorting primitives, we can compose complex hierarchical sorting. These composed sorting primitives can be used to build median operator, and eventually allow us to cover almost all database operations.

In total, we have developed over 40 templates; Using these templates, we implemented the following applications: Sorting (Merge, Insertion, Hybrid, and Radix), linear algebra (matrix multiplication, LU/QR/Cholesky decomposition, matrix inversion, Gaussian elimination), data compression (Huffman encoding), image processing (Gaussian blur, Susan corner detection, Harris corner detection, face recognition, face detection, lane detection,). Unfortunately, we do not have the space to describe all of these applications, but these templates available for general public through an open

source github repository (link not provided to maintain blind review).

6.7 Conclusion

In this chapter, we described a theoretical framework for parameterizable and composable HLS templates. Based on this theoretical framework, a new composable template can be built using existing composable templates hierarchically based on certain patterns. This new template will have new functionality and will be added to the template pool for later usage by domain experts. We built a highly optimized library of basic parameterizable templates and showed how to compose them to create a number of large applications from various domains. These designs were highly optimized and easily developed using our framework. Next, we will present a framework ”Resolve” which uses existing sorting templates and decorator pattern to generate customized sorting archiecture for a user given parameters.

Acknowledgements

This chapter contains materials from our current work. This work is to be submitted to relevant conference in March, 2015. Matai, Janarbek; Lee, Dajung; Alric Althoff; Kastner, Ryan. The dissertation author is the primary investigator and author of this work.

Algorithm 7: Procedures of BS construction algorithm

```
1  Procedure BulkSynchronous()
       Data: UC = {f_u, II_u, a_u}, D = data T = {t_1, t_2, ..t_n}
2      //Call the subroutines here
3  Procedure FindInstances()
       Data: T = {t_1, t_2, ..t_n}
       Result: M_A=set of instance templates for each t_i
4      forall the t_i ∈ T do
5          in=GetAllInstancesOf(t_i, D), i = 1, ..n
6      end
7      return M_A
8  Procedure ComposabilityCheck()
       Data: M_A
       Result: G=A set of graphs of composable templates
9      return G
10 Procedure ConstructBulkSynchronous()
       Data: G(V, G)
       Result: BS= BSComposability set of templates
11     currentVertex = v_0
12     if UC is ∅ then
13         while i < n do
14             Select next v ∈ V s.t. v(II) is minimum BS.AddToBS(v)
15             currentVertex = selected(v)
16         end
17     end
18     else
19         foreach (v_i) do
20             templates = GetAllComposableTo(v_i) BS.AddToBS(v)
21             foreach (templates) do
22                 cost = CalculateClosestPair(v_i, templates)
23             end
24             currentVertex = VertexWithMinCost()
               BS.AddToBS(currentVertex )
25         end
26     end
27     return BS
28 Procedure CodeGeneratorBS()
29     //Omitted for brevity
```

Table 6.1. Hardware area and performance results.

	Canny Edge			Matrix Inversion		
	QVGA/VGA	Ref [67]	Ref [55]	MI	Ref [70]	Ref [74]
Throughput	464/134	400	240	2.44	0.33	0.13
Size	*	360x280	512x512	4x4	4x4	4x4
Frequency	121/140	27	120	100	-	115
BRAM	9/9	-	-	10	1	9
Width	8/8	-	-	32	20	20

Chapter 7

Resolve: Automatic Generation of a Sorting Architecture

7.1 Introduction

In this chapter, we will demonstrate a case to compose templates to build highly optimized custom applications. To that end, we introduce Resolve: A framework that generates high performance sorting architectures by composing basic sorting architectures implemented with optimized HLS primitive templates. We built templates for basic sorting algorithms. Using these templates and common patterns, we presented a small domain-specific language which generates customized sorting architecture for a user requirement.

This concept is shown in Figure 7.1. We note that this is similar to *std* :: *sort* routine found in Standard Template Library (STL), which selects a specific sorting algorithm from a pool of sorting algorithms. For example, insertion sort is selected for lists smaller than 15 elements and then switches to merge sort for larger lists. We believe a routine like *std* :: *sort* for HLS is important to facilitate FPGA designs for non-hardware experts. Our framework is designed to integrate into a heterogeneous CPU/FPGA system using RIFFA [72]. This allows application programmers to quickly and easily create fully functional, heterogeneous CPU/FPGA sorting systems. We focused sorting application

domain because sorting is a widespread and fundamental data processing task especially in the age of big data.

Sorting is widely studied algorithmic problem [76] that is applicable to nearly every field of computation: data processing and databases [94, 32, 62], data compression [26], distributed computing (MapReduce) [46], image processing, and computer graphics [25, 79]. Each application domain has unique requirements. For example, text data compression typically requires sorting arrays with less than 704 elements [11]; MapReduce sorts millions of elements; databases must sort both large and small size arrays.

The major contributions of this chapter are:

1. The design and implementation of highly optimized basic sorting architectures, using sorting primitives written in a high-level synthesis language

2. A framework to generate hybrid sorting architectures by composing basic algorithms

3. A comparison of these generated sorting architectures with other sorting architectures implemented on a FPGA

The rest of this chapter is organized as follows. Section 7.3 describes the optimization of standard sorting primitives, and how to use them to create efficient architectures for ten basic sorting algorithms. Section 7.4 presents our framework. Section 7.5 provides experimental results. Section 7.6 discusses related work, and we conclude in Section 7.7.

7.2 High-Level Synthesis Optimizations

We present results using the Vivado HLS tool from Xilinx. Vivado HLS is a state-of-the art tool used by both academia and industry. While the coding techniques and optimizations that we use are specific to Vivado HLS, these general ideas can be applied to other tools. We refer the reader to [10] for available optimizations.

Most modern high-level synthesis tools from academia [30, 105] and industry [1, 10] provide optimizations that are typically embedded in input source code as a *pragma*. Two common optimizations are pipeline, which exploits instruction level parallelism, and unroll, which vectorizes loops. Unfortunately, designers must often write special code (i.e., restructured code) to generate good hardware. This restructured code requires substantial hardware expertise [56, 89].

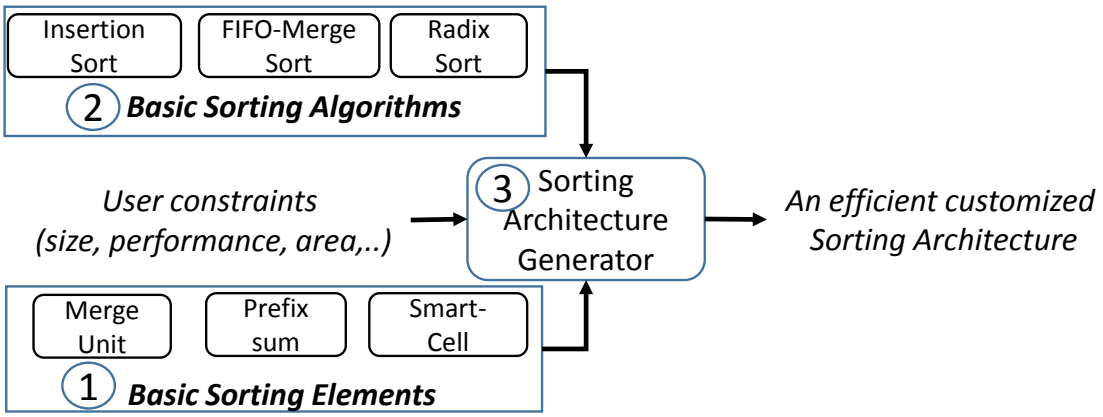


Figure 7.1. The sorting framework.

7.3 Hardware Sorting

This section presents our sorting primitives, their hardware implementations using HLS, and the implementation of ten basic sorting algorithms using these primitives. Figure 7.1 shows the structure of our framework. It has three components: 1) Block ① is a library of parameterizable sorting primitives. These sorting primitives are the building blocks of our framework. They are implemented in HLS and optimized to generate efficient hardware, as demonstrated in 7.2. Block ② represents our basic sorting algorithms. The algorithms use the sorting primitives to implement all the basic sorting algorithms on an FPGA using high-level synthesis. Block ③ is the sorting architecture generator. Here we use the sorting primitives and basic algorithms to generate optimized

Table 7.1. Sorting Algorithms evaluations when implementing them using HLS. n=number of elements to sort. *n=number of insertion sort cells, t*= number of compare-swap elements

		Parallel HLS Implementation			
Algorithm name	SW Complexity	Tasks	Complexity	Storage	Main Sorting Primitives
Selection sort	$O(n^2)$	2	$O(n^2/2)$	$O(2 \times n)$	Compare-swap
Rank sort	$O(n^2)$	n	O(n)	$O(n^2)$	Histogram, Compare-swap
Bubble sort	$O(n^2)$	2	$O(2 \times n^2)$	$O(2 \times n)$	Compare-swap
Insertion sort	$O(n^2)$	-	n	n*	Compare-swap, insertion-cell
Merge sort	$O(n \log n)$	-	$O(n)$	$O(2 \times \sum logn)$	Merge Unit
Quick sort	$O(n \log n)$ or $O(n^2)$	t	$O(n/t \log n/t)$	$O(n \times t)$	Prefix sum
Counting sort	$O(n \times k)$ (k=3)	3	n	(k-1)n	Prefix sum, Histogram
Radix sort	$O(n \times k)$ (k=4)	4	n	(k-1)n	Prefix sum, Histogram, Counting Sort
Bitonic sort	-	t	$log^2 n$	$O(n \times t)$	Compare-swap
Odd-even trans sort	$O(n^2)$	t*	$O(n^2/t^*)$	$O(t*)$	Compare-swap

hybrid sorting architectures to meet the user constraints.

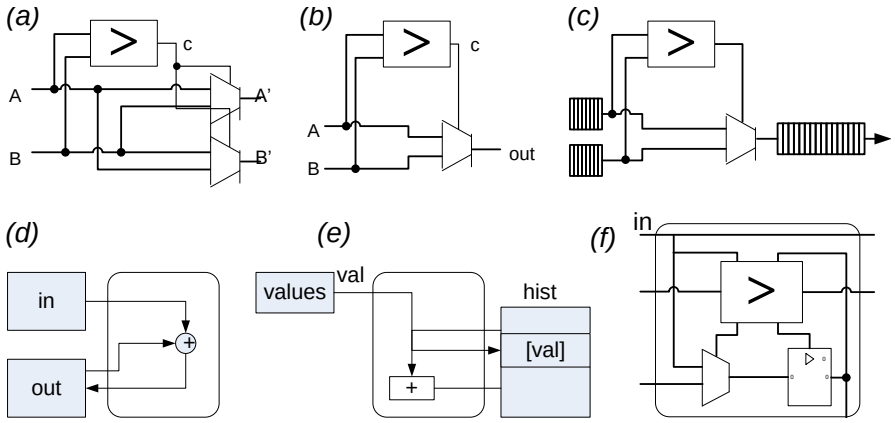


Figure 7.2. Initial hardware architecture of sorting primitives generated from HLS. a) compare-swap, b) select-value element, c) merge, d) prefix-sum, e) histogram, f) insertion cell

7.3.1 Sorting Primitives

This section presents optimized HLS implementations of our sorting primitives. Previous works presented a list of several common sorting primitives, e.g., compare-swap, select-value, and a merge unit [77]. After analyzing more common sorting algorithms, we added three more primitives to this list: prefix-sum, histogram, and insertion-cell. Our basic sorting algorithms (presented in Section 7.3.2) are implemented efficiently in hardware using these six sorting primitives. Figure 7.2 shows the initial hardware architectures generated from HLS code for our sorting primitives.

Section 7.2 described how *restructured* HLS code is necessary to generate efficient hardware. In this section, we will present the optimization of prefix sum, merge primitive and insertion-cell.

```
1 #pragma HLS DATAFLOW
2 //omitted partition
3 //pragmas
4 stage1(IN, TEMP);
5 ...
6 stage(TEMP, OUT);
```

```
7 }
```

Listing 7.1. Prefix sum dataflow

```
1 stage1(in, t) {
2   for(i=0; i<SIZE; ++i) {
3     #pragma HLS UNROLL factor=4
4     #pragma HLS PIPELINE
5     t[i] = in[i-1]+in[i];
6   }}
```

Listing 7.2. Prefix sum stages

In Chapter 5, we deeply covered design and implementation of prefix sum. As an additional example, we present another optimized HLS block for prefix sum which implements the reduction pattern in [66]. The reduction pattern uses $\log(n)$ parallel stages to compute a prefix sum of size n in parallel. The individual stages do not have the data dependency seen in the previous example. Listing 7.1 shows a high-level prefix sum implementation using a reduction pattern. The *stage* functions are implementations of the parallel stages without the data dependency. The Listing 7.2 shows the code for the first *stage* function. Since there is no data dependency, it is straightforward to get a speed up of 4 or more by unrolling and cyclically partitioning as in Listing 7.2.

When possible, we implemented multiple versions of optimized sorting primitives to facilitate further design space exploration. For example, the prefix sum presented in Chapter 5 achieves the desired unrolling factor with reduced frequency, while the prefix sum in Listing 7.1 with the same unrolling factor achieves higher frequency.

```
1 void MergeUnit(hls::stream<int> &IN1, hls::stream<int> &IN2, hls::stream<int> &OUT){
2   int a,b;
3   //read IN1 and IN2 into a,b
4   for(int i=0;i<n;i++) {
5   #pragma HLS PIPELINE
6     if(a<=b) {
7       OUT.write(a);
8       if(!IN1.empty())
9         IN1.read(a);
```

```
10        else
11          a=LARGE_NUMBER;
12      }else{
13        OUT.write(b);
14        if(!IN2.empty())
15          IN2.read(b);
16        else
17          b=LARGE_NUMBER;
18      }
19    }
20  }
```

Listing 7.3. FIFO based streaming merge primitive

The merge primitive merges two sorted $n/2$ size arrays into an array of size n. Figure 7.2 (c) architecture of merge primitive. We implemented merge primitive using C arrays and using streams. Here we present implementation of merge primitive using stream in HLS. Listing 7.3 shows the HLS implementation of streaming fifo-based merge unit. Here the $IN1$ and $IN2$ are two sorted arrays and OUT is the merged output. The loop in line 4 runs n times where $n/2$ is the size of $IN1$ and $IN2$. We pipelined this loop to get an initiation interval of 1, reading one element from $IN1$ or from $In2$ into a or b in a clock cycle. Next in lines 6-9 and 12-15, we write the smaller of a or b into the output. The *else* part of the logic on line 11 and 17 is used to get all the input array elements out of fifo by sending largest number. Input fifos can be modelled using an array or using $hls :: stream <>$ interface. We found out that $hls :: stream <>$ is the best suited to implement (for performance) fifo-based merge sorter than using an array.

The insertion-cell (Figure 7.2 (f)) is a sorting primitive used in insertion sort to design hardware-oriented insertion sort. It has an input, output, a comparator and a register. The insertion-cell compares the current input with the current value in current register. The smaller value between the current register and the current input is given as an output if the sorting direction is ascending. We will give an example of particular implementation of insertion-cell in Section 7.3.2 when we present implementation of

insertion sort. Next, we present basic sorting algorithms using these optimized sorting primitives.

7.3.2 Sorting Algorithms

In this section, we elaborate on the HLS implementation of our selected sorting algorithms. These algorithms are classified into four categories based on their structure: nested loop, recursive, non-comparison, and sorting network. These categories help us to identify HLS optimizations. Table 7.1 summarizes the results of our HLS implementations.

Nested Loop Sorting Algorithms

The *Selection Sort* algorithm finds the minimum or maximum element in an array and swaps it with the first or last element until the list is sorted. This algorithm runs in $O(n^2)$, where n is the number of array elements. In HLS, we can pipeline the inner loop to get $II = 1$, which still gives us $O(n^2)$ time. In HLS, we can implement better design by finding the minimum and maximum elements in parallel, which reduces the number of iterations in the outer loop by a factor of 2. This gives us $O(n^2/2)$ time. In general, selection sort does not translate into high performance hardware using HLS. However, it can produce an area-efficient sorting algorithm implementation.

Rank sort algorithm sorts by computing the rank of each element in an array, and then inserting them at their rank index. The rank is the total number of elements greater than or less than the element to be sorted. Sequential rank sort has a complexity of $O(n^2)$. The rank sort algorithm can also be fully parallelized in HLS: To sort an array of size n, there are n units operating in parallel computing the rank of each element. However, this process uses $2 \times n^2$ storage (BRAM) to sort the array of size n. Rank sort can be useful when designing sorting hardware in HLS because it is a good algorithm for exploring

area and performance trade-offs.

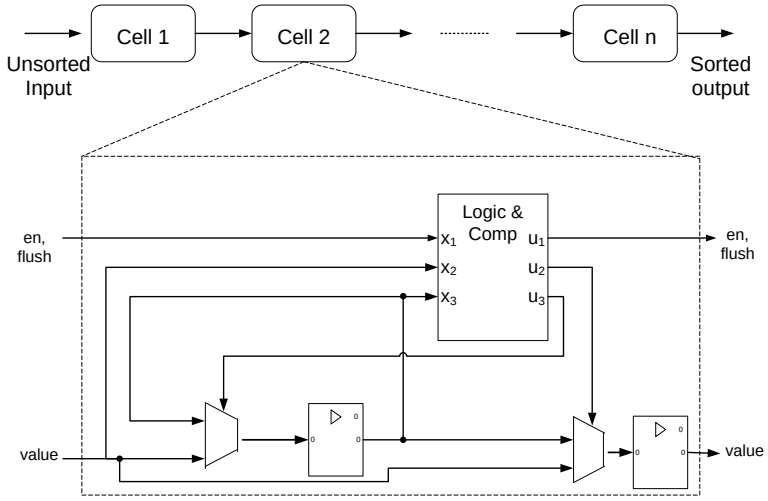


Figure 7.3. Hardware architecture of linear insertion sort

Insertion sort iterates through an input array maintaining sorted order for every element that it has seen. Insertion sort has a complexity of $O(n^2)$. Listing 7.4 shows a software-centric HLS implementation of insertion sort. We attempted to pipelined the inner *while* loop in line 5, however, because of loop exit test in line 5 on variable $j =$ and the update of j an initiation interval of 1 is not possible. This makes the software-centric HLS implementation of insertion sort (as it is in Listing 7.4) worse than the HLS implementations of selection sort or rank sort.

```
1 void InsertionSort (array) {
2 for (i=1; i < SIZE; i++){
3 index = array[i];
4 j = i;
5 while ((j > 0)&& (array[j-1] > index) ){
6   #pragma HLS PIPELINE II=1 // II>1
7   array[j] = array[j-1];
8   j = j - 1;
9   }
10 array[j] = index;
11 }}
```

Listing 7.4. Insertion Sort code for HLS design

However, insertion sort can be implemented in hardware efficiently using linear array of insertion-cells [104, 20, 84, 15] or using sorting networks [96]. We will discuss sorting network implementation in HLS later. Now we present insertion sort implementation based on a recent work [15] which implements linear insertion sort. Figure 7.3 shows architecture from the work [15]. In this architecture a series of cells (insertion-cell primitives) operate in parallel to sort a given array. It compares the current input (IN) with the current value in current register (CURR_REG). The smaller of current register and the current input is given as an output to OUT. The Listing 7.5 shows the source code that represents the hardware architecture in Figure 7.3. The lines 1-12 shows code that represents a insertion-cell architecture. The cell function must save its current value between calls; This is done with a *static* variable. A cascade of insertion-cells are modelled in HLS code using dataflow pragma (see [10]) and series of calls to cell functions as shown in 12-19 lines of Listing 7.5. Since we have to store the values in (CURR_REG) of each cell, we created n copies of the cell function. With this HLS implementation, we achieve $O(n)$ time complexity to sort an array of size n.

```
1 T cell0(hls::stream<T> &IN, hls::stream<T> &OUT){
2   static T CURR_REG=0;
3   T IN_A=IN.read();
4   if(IN_A>CURR_REG) {
5     OUT.write(CURR_REG);
6     CURR_REG = IN_A;
7   } else
8     OUT.write(IN_A);
9   return CURR_REG;
10 }
11 ...
12 void InsertionSort(hls::stream<T> &IN, hls::stream<T> &OUT){
13   #pragma HLS DATAFLOW
14   // Function calls;
15   cell0(IN, out1);
16   cell1(out1, out2);
17   ...
18   cell7(out7, OUT);
19 }
```

Listing 7.5. Insertion Sort code for HLS design based on the hardware architecture in Figure 7.3.

Recursive Algorithms

Merge sort and *Quick sort* is not implementable in HLS because of recursive functions. In the software implementation of merge sort, there are two tasks. One task partitions the array into individual elements, and the second merges those units. Thus main work is done in the merging unit, which is implemented with merge primitive.

Merge sort can be implemented in HLS using merge sorter tree [77] or using odd-even merge sort. The Listing 7.6 shows skeleton code for streaming merge sorter tree. In this code, $IN1$, $IN2$, $IN3$ and $IN4$ are $n/4$ size input and OUT is n-size output. $MergePrimitive1$ and $MergePrimitive2$ respectively merges two sorted list of array size $n/4$ and $n/2$. Using dataflow pragma, we can functional pipeline all the functions in the code. Merge sort based on odd-even merge also uses merge sorting primitive to sort a given *n* size array in *n* time complexity parallelizing *nlogn* tasks.

```
1 void CascadeMergeSort(hls::stream<int> &IN1,
2       hls::stream<int> &IN2, hls::stream<int> &IN3,
3       hls::stream<int> &IN4, hls::stream<int> &OUT){
4   #pragma HLS DATAFLOW
5   #pragma HLS stream depth=4 variable=IN1
6   for(int i=0;i<SIZE/4;i++) {
7     //read input data
8   }
9   MergePrimitve1(IN1, IN2, TEMP1);
10  MergePrimitve1(IN3, IN4, TEMP2);
11  MergePrimitve2(TEMP1, TEMP2, OUT);
12 }
```

Listing 7.6. FIFO based streaming merge sorter tree

Quick sort uses a randomly selected pivot to split an array into elements that are larger and smaller than the pivot. After selecting pivot, a function named $pivot\_function$

does the following. In $pivot\_function$, all elements smaller than pivot will be placed on the left side of it in the array, and all equal or greater elements come after that. This process will be repeated for the smaller portion and greater portion separately. The software complexity of this algorithm is n^2 in the worst case and $nlogn$ in the best case. We implemented non-recursive version of quick sort in HLS by selecting a pivot in a constant time. The main work is done in $pivot\_function$ and we use prefix sum to optimize the quick sort. To implement the quick sort in HLS, we can run t tasks to divide the work of $pivot\_function$ to sort n size array into n/t. The integration of t results from tasks can be done using prefix sum.

Non-comparison based

Counting sort has three stages. An example of counting sort for 8-element input array is shown in Figure 7.4 (a). In the first stage, the counting sort computes the histogram of elements from the unsorted input array. In the second stage, it finds prefix sum of previous stage. In the given example, the $0+3=3$ is written in the index 1 of this stage. The final stage puts the array in sorted order. First it reads the value from the unsorted input array. Then it finds the first index of that element from the prefix sum stage and writes it to the output array. Then it increments the index in the prefix sum (again histogram) by one. Parallel counting sort can be designed using dataflow pipelining of three stages (histogram, prefix sum, histogram), and it runs in n time using $n \times k$ (k is usually 3) memory storage. The $3n$ storage is used between intermediate stages of dataflow pipelining.

Radix sort sorts input by applying counting sort by k times for each radix one after another. Usually, counting sorts are applied to the input array in bit radices. For example, to sort 32-bit integers, we can apply counting sort four times to each of the hexadecimal digits. We can implement fully parallel radix sort in HLS by dataflow pipelining of each

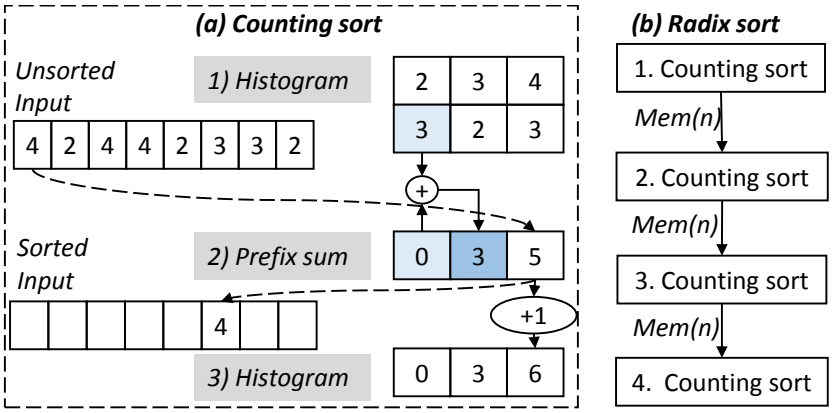


Figure 7.4. An example hardware architectures for counting sort and radix sort

counting sort. Each counting sort has a throughput of n, thus fully parallel radix sort will have a throughput of n. To store the outputs of intermediate stages, we need $n \times k$ storage. Here k is usually 4 for 32-bit number or 8 for 64-bit number. Thus to sort 32-bit number in parallel, we use $3 \times n$ storage (3 intermediate memory storage) as shown in Figure 7.4 (b). Our empirical results suggest that radix sort is good for small to medium size (10^{12}) arrays.

Sorting networks

Sorting networks are [96] popular parallel algorithm sorting network is a network of compare-swap primitives connected by wires. *Bubble sort* is a particular algorithm that can be implemented this way. The minimum of the inputs x_0 and x_1 is assigned to the upper wire and the maximum goes to the lower wire. Figure 7.5 (a) shows an example of compare-swap element used in sorting networks. Figure 7.5 (b), (c), (d) show the networks generated by bubble sort, bitonic and odd-even transposition sorting algorithms. Due to parallel nature of sorting networks, they are easier to implement in HLS but does not scale well. This requires balancing the parallelism and area in HLS and will be discussed later. For example using parallel n compare-swap elements, odd-even transposition sort can sort an n size array in $O(n)$. We do not cover the HLS code used

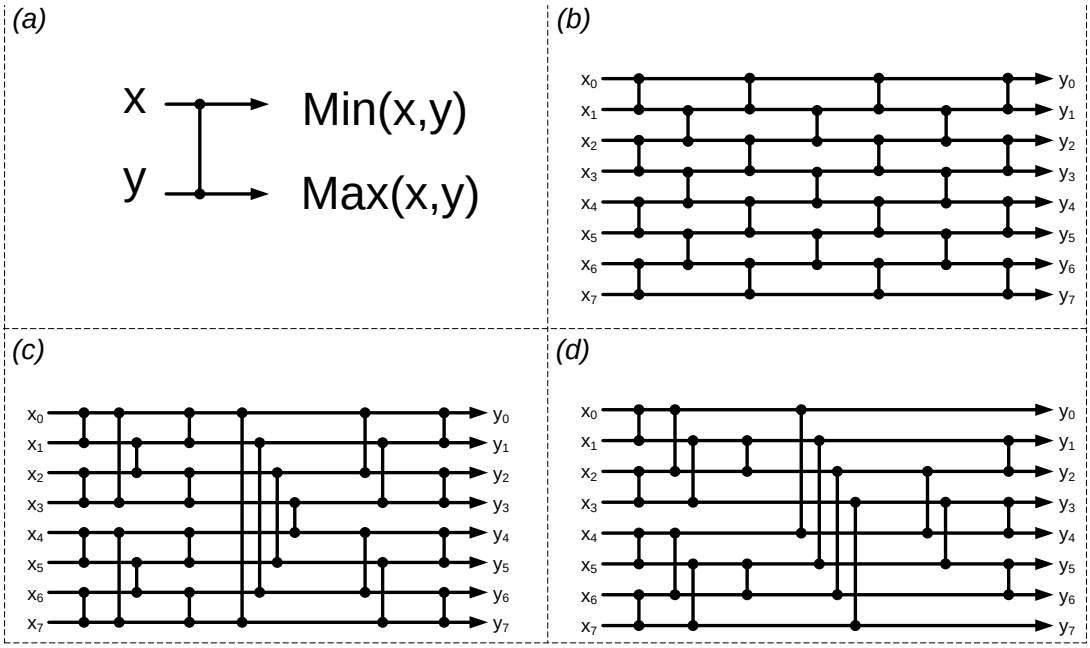


Figure 7.5. Sorting networks. a) Odd-even trans sort (Bubble sort), b) Bitonic sort, b) Odd-even transposition sort

for sorting networks because they are straightforward to write.

7.4 Sorting Architecture Generator

In this section, we will present sorting architecture generator from the user's perspective. Users can use our framework in two different ways. The first way for doing design space exploration of particular sorting algorithm for a range of user parameters. The second way is relying on a sorting framework to generate customized sorting architecture for a given user constraints. In the rest of this section, we focus on the second way of using the framework. The flow of using our sorting framework is shown in Figure 7.7. We define user constraint as a tuple $UC(T,S,B,F,N)$ where T, S, B, F and N are throughput, number of range of slices, number of range of block rams, frequency for a sorting design of size N. We define V as a set of sorting designs that can do sorting of N size array. Sorting architecture generation is a problem to find a design D of the form

$D(T,S,B,F,N)$ that satisfies UC.

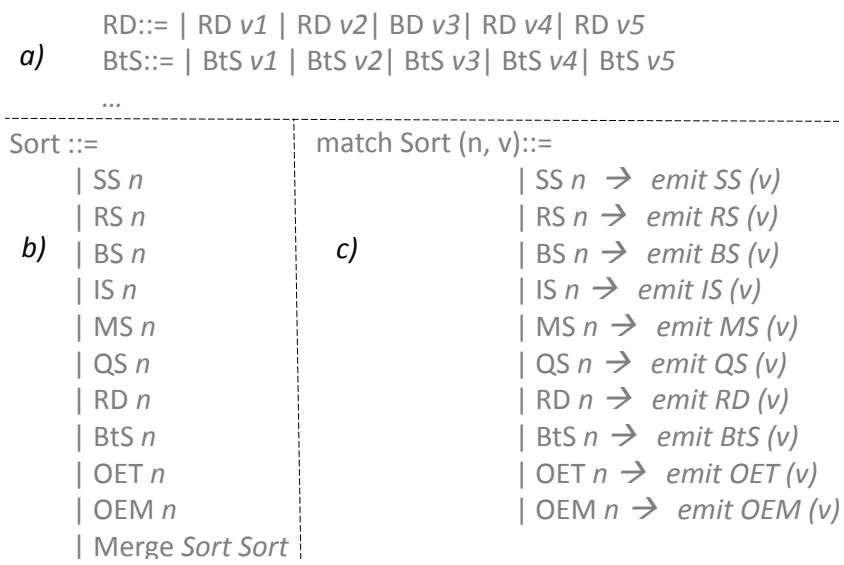


Figure 7.6. SS=Selection sort, RS = Rank sort, BS=Bubble sort, IS=Insertion sort, MS=Merge sort, QS=Quick sort, RD=Radox sort, BtS=Bitonic sort, OET=Odd-even transposition sort, OEM=Odd even merge sort. a) Sorting architecture variants for particular algorithm, b) Sort function grammar, c) Code generator

The framework is written in Python and implemented as a small domain-specific language. Figure 7.6 shows simplified grammar of the language. Sorting architectures defined in previous section are defined by types for instance, *RD* and *IS*. Each sorting algorithm has a number of different implementations, called *variants*. For example radix sort, *RD*, has five variants: $RD\_v1, RD\_v2, RD\_v3, RD\_v4, RD\_v5,$. The *sort* function can use any sorting algorithm, or hybrid composition of one or more algorithms. For example, to sort an array of size *n*, *sort* can be any of followings:

```
sort::=SS, RS, BS, IS, MS, QS, RD, BtS, OET, OEM
or
sort::= Merge(Sortn/2, Sortn/2)
or
sort::=Merge (Merge(Sort n/4, Sort n/4), Merge(Sort n/4, Sort n/4))
```

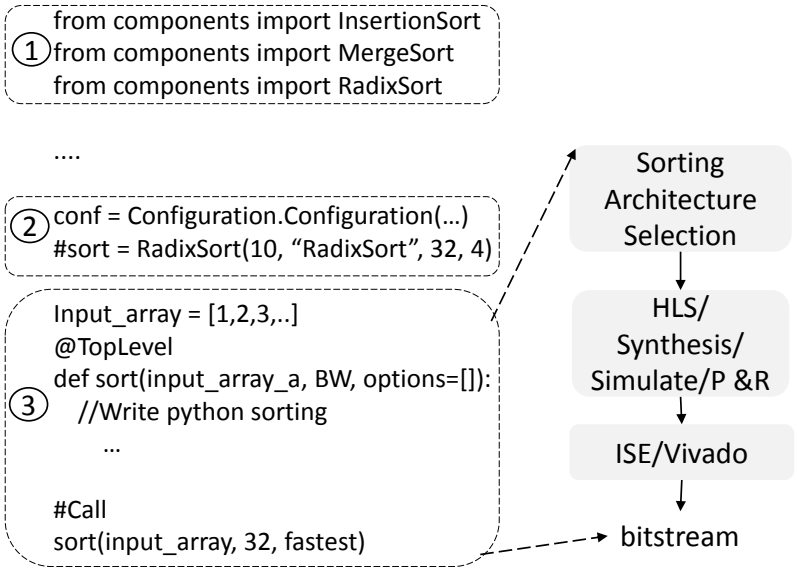


Figure 7.7. Sorting architecture generation

Based on the *sort* function (primitive or hybrid merge), the framework generates specific variant of sorting architecture. Thus our framework completely abstracts the underlying architectural details from the user, and allows the user to generate an optimized architecture in a matter of minutes.

To use the framework, the user writes python code as in Figure 7.7. It has three components: ① is a library of the template generator classes for existing sorting algorithms (e.g., InsertionSort, MergeSort). There are currently eleven classes, some with multiple architecture variants. All these classes inherit from base class called *Sorting*. The *Sorting* class provides common class methods and members (e.g., size, bit width) for all the sorting algorithms. Each class provides parameterizable functions tailored to specific sorting algorithm. For example, $RadixSort.optimized\_II1(size, bit-width)$ generates optimized Radix sort with $II = 1$, while $functional\_pipelining(size, bit\_width)$ generates dataflow pipelined radix sort for a given parameters. ② is HLS project generator and configuration class. The configuration class accepts four parameters. These are FPGA device, frequency, clock period, simulate_true, implement_true, and name of the module.

If $simulate\_true = 1$ then the generated design is simulated and verified in modelsim. If the $implement\_true = 1$, then the design is physically implemented.

Algorithm 8: Customized Sorting Architecture Generation

```
   Data: UC={T,S,B,F,N}, V={V_1,V_2,..V_m}, P={N/2,N/4..}
   Result: D=architecture for UC, R=performance area results
1  if UC is 1 then
2  |  [D,R]=SorterGenerator(V, N)
3  end
4  else
5  |  foreach (P) do
6  |  |  [D,R]=SorterGenerator(V, P)
7  |  |  if R(II) < N then
8  |  |  |  emitMerge(D, P)
9  |  |  |  if sim/impl is 1 then
10 |  |  |  |  R = Simulate D R =Implement D
11 |  |  |  end
12 |  |  end
13 |  end
14 |  Procedure SorterGenerator()
   |  |  Data: V,N
   |  |  Result: D,R
15 |  |  TS(1,2,..,m)=CalculateThroughput(V,N)
16 |  |  S = min(V_1(t),V_2(t),..V_m(t))
17 |  |  [D,R]=emitCode S
18 |  |  if sim/impl is 1 then
19 |  |  |  Simulate D Implement D
20 |  |  end
21 end
```

Block ③ is where the users write their *TopLevel* function that calls the sorting routine. The *TopLevel* is a python *decarator* which allows us to add additional functionality to the existing python code. Once *TopLevel* decorator starts executing, it does several things: First, it generates customized sorting architecture tailored to user provided parameters using Algorithm 8. Here V is a set of all different variants of existing sorting architectures, D and R are returned sorting design and respective simulation/ implementations results. User provides UC. UC must contain at least one

element which is size of array to sort (N). If UC is one, then sorter generates a design from existing designs which has the highest throughput using *SorterGenerator* function. The *SorterGenerator* function uses initial II of each variant to calculate throughput of them (TS). We assume II of each variant is known.For example, we know linear sort (LIS) has $II = 1$, so the $TS(LIS) = 1 \times N$. Then it generates design D and returns report R. In the case of $|UC| > 1$, we must satisfy these conditions: $UC(T) > D(T)$, $UC(S) < D(S)$, $UC(B) < D(B)$, $UC(F) > D(F)$. We presented a heuristic approach in line 5-13 for $UC(T) > D(T)$. Procedure for conditions can be written in similar way. Here it heuristically finds a design D that has a throughput $R(II) < N$ because throughput of sorting design is limited by N.

7.5 Experimental Results

We present the performance and area results for a representative sample of architectures generated by our framework. We also compare our HLS designs with existing RTL implementations of sorting hardware architectures. All designs implemented targeting xc7vx485tffg1761-2 using Xilinx's Vivado HLS 2014.4 tool. Performance results are presented in terms of million samples per second (MSPS) or Megabytes per second (MB/s). Instead of competing for best results, we have opted to show a broad set of architectures to highlight the composability of our framework.

In our experiments we first implemented seven different sorting algorithms for three different problem sizes (32, 1024, and 16384). These sorting architectures are used later when we compose algorithms. These are Selection Sort, Rank Sort, Linear Insertion Sort, Merge Sort (two variants), Radix Sort (two variants), Bitonic Sort, and Transposition Sort (two variants). The results are tabulated in Table 7.2

For single-algorithm architectures, selection sort and rank sort have the worst throughput/area trade-off. In HLS implementations of both of these algorithms, we

created an architecture where two processing units run in parallel as discussed above. Rank sort provides 508 MB/s because of achievable parallelism for a small (32) size. For a larger n, their respective throughput decreases significantly due to required amount of compare-swap element needed to make them faster (embarrassingly parallel). Linear insertion sort operates best when $n <= 1024$, and suffers as n increases. The number of slices increases linearly as the number of insertion cells increases. Designs, *Mergesort* is based on (cascade of merge units) odd-even merge sort algorithm [76]. At this moment, full merge sort is using local memories (BRAMs) as a intermediate storage between stages.

*Radixsort*8 and *Radixsort*4 sort using 8-bit and 4-bit radices respectively. In $8-bit$ variant of radix sort there are 4 parallel task and in $4-bit$ variant there are 8 parallel tasks. Radix sort has a good area-throughput trade-off; In the 4-bit implementation, doubling the area produces a 4-times speed-up with higher frequency. This makes it suitable for medium size (1K-8K) size array sorting in for high frequency. Bitonic sort achieves highest throughput for arrays smaller than 1024 elements, however it is not scalable.

After the break, in Table 7.2 we present four hybrid sorting architectures. These are $Merge(Stream)$ and $Merge4$, $Merge8$, and $Merge16+Radix$. $Merge(Stream)$ is a streaming merge sort that operates on pre-sorted inputs. Thus, $Merge(Stream)$ is suitable for in scenario where sorting is done on CPU-FPGA heterogeneous systems. $Merge4+Radix$ uses Merge sort to combine four 4096-element Radix sorts, which gives the highest throughput design with less than 170 Block RAMs. $Merge8+Radix$ and $Merge16+Radix$ are similar designs as with $Merge4+Radix$. $Merge4+Bitonic$ is hybrid of bitonic and merge primitive. Generation of these architectures can be automated in our framework based on Algorithm 8. For example, if $UC(T=H, n=16384, B<170)$, then $Merge4+Radix$ should be selected based on Algorithm 8.

Table 7.2. HLS Implementation of different sorting architectures. Tasks=number of parallel sorting processes.

		32			1024			16384		
Algorithm name	Tasks	Slices / BRAM	Frequency	MB/s	Slices/BRAM	Frequency	MB/s	Slices/BRAM	Frequency	MB/s
Selection sort	2	26 / 0	266	50	410 / 12	232	3.5	599 / 192	171	< 10
Rank sort	2	119 / 4	389	508	162 /16	419	4	504 / 256	348	< 10
Linear Insertion sort	n	374 / 0	345	1380	12046 / 0	310	1243	- / -	-	-
Merge sort	nlogn	666 / 18	180	550	1268 / 40	281	899	2474/ 832	177	567
Radix sort 8	4	1420 / 19	227	42	1500 / 36	230	202	1743 / 456	222	220
Radix sort 4	8	2146 / 30	353	223	2470 / 60	362	356	3352 / 960	289	289
Bitonic sort	-	6412 / 0	550	2197	11022 / 216	330	1297	-	-	-
Odd-even trans	8*2	929 / 33	342	96	1254 / 36	301	15 1361 / 128	225	0.8	
Odd-even trans	16*2	1326 /0	323	70	2209 / 68	270	29	2370 / 128	212	1.64
Merge (Stream)	-	221 / 0	395	1407	231 / 0	374	1490	255 / 0	368	1474
Merge4 + Radix	-	- / -	-	-	- / - /	-	-	1010 / 168	244	411
Merge8 + Radix	-	- / -	-	-	- / - /	-	-	2584 / 240	245	782
Merge16 + Radix	-	- / -	-	-	- / - /	-	-	4786 / 320	148	858
Merge4 + Bitonic	-	2973 / 2	299	1196	- / - /	-	-	- / -	-	-

End-to-end sorting system: We integrated a generated design for insertion sort with CPU using RIFFA [72]. HLS sorting architectures use AXI stream in HLS. Then corresponding signals of AXI interface is connected to signals of RIFFA. The area and performance of the demo is shown in Figure 7.3. RIFFA uses 12 channels each is 128-bit long. Currently, we area using only 32-bit of one channel to transfer data between CPU and FPGA. If we use more channels, our demo system can be faster.

Table 7.3. Area and performance of End-to-End demo

	LUT	FF	BRAM	Frequency
Insertion sort + RIFFA Demo	11868	8793	19	125

7.6 Related work

There are a variety of published works exploring sorting architectures on FPGA platforms. Several works have implemented a single sorting algorithm on a FPGA [104, 84, 20, 139, 95], and some have explored high performance sorting of large size inputs [32, 77].

Table 7.4. Streaming insertion sort generated in this paper vs. Interleaved linear insertion sorter [104].

	16	64	128	256
Throughput (MSPS) [104]	18.6	4.6	2.33	1.16
Throughput of this work (MSPS)	22.1	5.3	2.54	1.29
Ratio	**1.18X**	**1.13X**	**1.08X**	**1.1X**
Slices [104]	278	1113	2227	4445
Slices of this work	187	792	1569	3080
Ratio	**0.67X**	**0.7X**	**0.7X**	**0.69X**

In this paper, we compare throughput and area results of generated HLS designs with the interleaved linear insertion sorter (ILS) by Ortiz et al. [104], sorting networks

by Zuluaga et al. [139] and merge sorter by Koch et al. [77]. We selected these works because insertion sorter is usually best suited for small size arrays, while sorting networks are used for both small and medium size arrays, and a merge sorter is best for larger size arrays. The Table 7.4 presents throughput and area results of interleaved liner insertion sorter (ILS) and our streaming insertion sorter for different sizes (16, 64, 128 and 256). We calculated slices of ILS by using slices per node $\times$ number of elements (size). Slices per node for $w = 1$ is obtained from [104]. The throughput is the number of mega samples per second for a given size (size=16, 64, 128 and 256). Our insertion sorter has average 1.1X better throughput while using 0.6X less slices.

We also compared our bitonic sort results to the bitonic sorting network presented in [139]. Our work achieves 7X better throughput for size 16 and achieves the same throughput for size 2048 using 200 more block rams. Using hybrid design (1024 linear insertion sort + streaming merge sort), we can achieve the same throughput almost using no block ram with our framework. We also compared our results with work by Koch et al [77] which sorts 64-bit integers of size 43K. We generated a sorting architecture (hybrid of radix and merge sort) for 64-bit data that sorts 43K size array. We achieved the same throughput (1 GB/s) using 3 times more BRAMS. We use more BRAMS because [77] implements fifo-based merge sort using a shared memory blocks for both input streams. Writing a fifo by two different processes while doing functional pipelining is not supported by current HLS tool.

7.7 Conclusion

In this chapter, we present a framework that generates customized sorting architectures. The framework is built using templates of basic sorting algorithms and design patterns. The framework provides a small embedded domain-specific language built in Python (mainly sort function). Generated designs have competitive results with manually

design RTL sorting architectures. In the future, we plan to fully automate the generation of end-to-end custom sorting architectures using RIFFA.

Acknowledgements

This chapter contains materials from a work that is currently in progress (to be submitted within 2015 to relevant conference). Matai, Janarbek; Richmond, Dustin;Lee, Dajung; Zac Blair; Kastner, Ryan. The dissertation author is the primary investigator and author of this work.

Chapter 8

Future Research Directions

High-level synthesis method generates high quality RTL when the input code is restructured to reflect underlying micro-architectural details. Restructured code writing process remains difficult and non-intuitive task. One needs 1) domain knowledge about the application, 2) hardware design expertise, and 3) the ability to translate the domain knowledge into an efficient hardware design in order to write restructured code for an application. In this thesis, instead of generating all possible hardware (using different combination of ”C”+directives) from HLS code, we studied the best hardware designs for common applications kernels, and we developed a number of highly parameterizable templates for HLS. We learned that having highly optimized HLS templates allow us to build domain-specific tools to design hardware (e.g., sorting) using different computational and structural patterns. We suggest following researches directions which immediately follow from this work.

8.1 End-to-End System Design

In addition to the programming model challenges presented above, verification and communication with HLS kernel on a real FPGA is essential. Current methods of verifying an HLS core on a real FPGA involve several tool flows and non-trivial IP cores such as, DDR controllers, PCI Express Interfaces, and DMA engines to access FPGA

memory from an external processor. Using the vendor specific tools with correct IP cores remains difficult even for hardware engineers. One way to solve this problem is providing easy to use high level communication framework between FPGA and host CPU (or FPGA to FPGA or FPGA to Network). Open source frameworks such as RIFFA provide a good abstraction for software developers to access communicate with an FPGA [72]. Integrating easy to use frameworks such as RIFFA with HLS tools will allow easy verification of application code for software developers.

8.2 Design Space Exploration

Design space exploration (DSE) with HLS tools is an essential feature for exploring the performance, area, and power trade-off of different architectures and underlying implementations. Currently, DSE with HLS is usually done manually. Automatic, but efficient DSE for given code is needed to allow HLS users to tailor hardware for their specific needs. Since DSE is a difficult problem because of large search space, blindly optimizing C/C++ code does not produce efficient hardware. In fact doing DSE on the provided code will result excessive run-time for a single run with current state-of-the-art HLS tools. Using restructured domain specific templates as an input to DSE will allow automatic and efficient DSE with HLS. This is due to the fact that restructured domain specific templates efficiently capture the hardware architecture. These hardware architectures are represented by small number of restructured HLS codes. As a result, domain specific templates reduce the size of the search space. As a future work, it is natural to build an automated tool based on domain-specific restructured HLS templates which will provide faster and efficient design space exploration of a given domain.

In addition to above research directions, building domain and application specific restructured code and identifying computational patterns are important to generate optimized accelerators. We identified following application domains which can benefit from

hardware accelerators.

8.3 Data Processing on an FPGA

FPGAs provide opportunities to process high-volume data in a fast and efficient way. Studying the ways to enable any "'data scientist"' to benefit from designing FPGAs. This can be done by studying common database kernels and finding ways to make them highly parametrizable and composable in a structured way (like sorting). Some of the immediate candidates of database operations are combining sorting to join operation. This acceleration will allow to join two database tables in hardware. This research allows us generate a scalable "accelerated custom"' processor for most data processing operations from user constraints. This will allow "'data processing"' to benefit both from performance and power efficiency of FPGAs.

8.4 Machine Learning Acceleration

In the past, I designed a number of efficient hardware architectures for various computer vision algorithms. Based on these experiences, I will explore domain-specific language based hardware acceleration designs for machine learning algorithms. The primary goal of this project is to generate highly optimized hardware architectures for low power mobile devices without writing low-level RTL and to enable scientists and software programmers to benefit from performance and power efficiency of hardware accelerators easily. Additionally, this research will benefit large companies such as Google, Facebook and Amazon. If successful, this research allows efficient computation of many applications that run on large clusters (data centers).

Appendix A

HLS Codes

A.1 Restructured Code

We provided following restructured HLS codes for different domains. Each folder contains necessary restructured code (*.cpp files), testbench files (*_test.cpp) and scripts to create a project. These restructured designs grouped into four different folders and uploaded into bitbucket repository (https://bitbucket.org/janarbek/templates). We also present set of restructured HLS designs for sorting domains. Our restructured HLS sorting covers all basic sorting algorithms. Restructured designs for sorting domain is located in different repository (https://bitbucket.org/janarbek/sortingframework).

1) Linear algebra restructured HLS designs:

```
/
└── linearalgebra
    ├── vecvec
    ├── matvec
    ├── matmat
    ├── spmv
    ├── matrixdecomposition
    │   ├── lu
    │   ├── qr
    │   └── cholesky
    └── matrixinversion
        ├── lu
        ├── qr
        ├── cholesky
        └── gaussianelimination
```

2) Signal processing restructured HLS designs:

```
/
└── signalprocessing
    ├── FFT
    ├── FIR
    ├── DFT
    ├── phaselocker
    └── channelequalizer
```

Following are computer vision codes.

3) Computer vision restructured HLS designs:

```
/
└── vision
    ├── tutorial
    ├── convolution
    ├── skincolordetection
    ├── lanedetection
    ├── harris
    ├── cannyedge
    ├── facerecognition
    └── facedetection
```

4) Other restructured HLS designs:

```
/
└── misc
    ├── histogram
    ├── prefixsum
    ├── Huffman
    ├── LM
    └── BFS
```

A.2 Streaming and Blocking Matrix Multiplication

```
1
2 typedef struct{
3   DTYPE a[BLOCK_SIZE];
4 } blockvec;
5
6 typedef struct{
7   DTYPE out[BLOCK_SIZE][BLOCK_SIZE];
8 } blockmat;
```

```
 9
10
11 void matmatmul(hls::stream<blockvec> &matrix1, hls::stream<blockvec> &matrix2,
       hls::stream<blockmat> &out, DTYPE it)
12 {
13 #pragma HLS DATAFLOW
14
15 blockvec m ;
16 DTYPE v = 0;
17 DTYPE c11 = 0;
18 DTYPE c12 = 0;
19 DTYPE c21 = 0;
20 DTYPE c22 = 0;
21
22 blockmat out_temp;
23 int counter =0 ;
24 counter = it%BLOCK_SIZE;
25
26 blockvec tempA;
27 blockvec tempB;
28 static DTYPE A[BLOCK_SIZE][SIZE];
29 //Need to fix A[i][j] and A[i+1][j]
30 if(counter==0) {
31   for(int i=0;i<BLOCK_SIZE;i=i+BLOCK_SIZE) {
32     for(int j=0;j<SIZE;j++) {
33       #pragma HLS PIPELINE II=1
34       tempA= matrix1.read();
35       A[i][j] =tempA.a[0];
36       A[i+1][j] =tempA.a[1];
37     }
38   }
39   //coutMatrix2D(BLOCK_SIZE, SIZE, A);
40 }
41
42 for(int i=0;i<BLOCK_SIZE;i=i+BLOCK_SIZE) {
43   c11 = 0;
44   c21 = 0;
45   c12 = 0;
46   c22 = 0;
47   for(int j=0;j<SIZE;j++) {
48   //#pragma HLS UNROLL factor=4
49   #pragma HLS PIPELINE II=1
50
51   tempB =matrix2.read();
52
53   //These are block sizes 0 and 1
```

```
54   c11 = c11 + A[i][j]*tempB.a[0];
55   c21 = c21 + A[i+1][j]*tempB.a[0];
56
57   c12 = c12 + A[i][j]*tempB.a[1];
58   c22 = c22 + A[i+1][j]*tempB.a[1];
59
60   }
61
62   out_temp.out[0][0] =c11;
63   out_temp.out[1][0] =c21;
64   out_temp.out[0][1] =c12;
65   out_temp.out[1][1] =c22;
66
67
68
69   out.write(out_temp);
70 }
71 }
```

Listing A.1. Streaming and blocking matrix multiplication

```
1
2 for(int it1=0;it1<SIZE;it1=it1+BLOCK_SIZE){
3 for(int it2=0;it2<SIZE;it2=it2+BLOCK_SIZE){
4
5   row = it1;//row + BLOCK_SIZE*factor_row;
6   col = it2;//col + BLOCK_SIZE*factor_col;
7
8   for(int i=0;i<BLOCK_SIZE;i=i+2){
9     for(int k=0;k<SIZE;k++){
10      //Send BLOCK_SIZE x  SIZE matrix portion of A whenever needed.
11      if(it%BLOCK_SIZE==0) {
12        strm_matrix1_element.a[0] = matrix1[row][k];
13        strm_matrix1_element.a[1] = matrix1[row+1][k];
14        strm_matrix1.write(strm_matrix1_element);
15      }
16
17      strm_matrix2_element.a[0] = matrix2[k][col];
18      strm_matrix2_element.a[1] = matrix2[k][col+1];
19      strm_matrix2.write(strm_matrix2_element);
20      }
21  }
22  matmatmul(strm_matrix1, strm_matrix2, strm_out, it);
23  strm_element_out = strm_out.read();
24  matrix_hwout[row][col] = strm_element_out.out[0][0];
25  matrix_hwout[row+1][col] = strm_element_out.out[1][0];
```

```
26    matrix_hwout[row][col+1] = strm_element_out.out[0][1];
27    matrix_hwout[row+1][col+1] = strm_element_out.out[1][1];
28
29    it = it +1;
30  }
31
32  }
```

Listing A.2. Streaming and blocking matrix multiplication testbench

Bibliography

[1] Altera sdk for opencl. http://www.altera.com/.

[2] Face detection verilog design. http://ercbench.ece.wisc.edu.

[3] Impulse c. http://www.impulseaccelerated.com/.

[4] Intel power gadget. http://software.intel.com/en-us/articles/intel-power-gadget-20.

[5] Mako template language. http://www.makotemplates.org.

[6] Mitron c. http://mitrionics.com/.

[7] Orl face database. http://www.cl.cam.ac.uk/research/dtg/attarchive.

[8] Shttp://cadlab.cs.ucla.edu/. CMOST.

[9] Synphony c. http://www.synopsys.com/.

[10] Vivado design suite user guide: High-level synthesis. http://www.xilinx.com/.

[11] Xpress compression algorithm. http://msdn.microsoft.com/en-us/library/hh554002.aspx.

[12] Zynq-7000 ap soc measuring zc702 power using standalone application tech tip. http://www.wiki.xilinx.com/.

[13] Zynq all programmable soc sobel filter implementation using hls. http://www.xilinx.com/.

[14] Christophe Alias, Alain Darte, and Alexandru Plesco. Optimizing remote accesses for offloaded kernels: application to high-level synthesis for fpga. In *Conference on Design, Automation and Test in Europe*, pages 575–580. EDA Consortium, 2013.

[15] Oriol Arcas-Abella, Geoffrey Ndu, Nehir Sonmez, Mohsen Ghasempour, Adria Armejach, Javier Navaridas, Wei Song, John Mawer, Adrián Cristal, and Mikel Luján. An empirical evaluation of high-level synthesis languages and tools for database acceleration. In *24th International Conference on Field Programmable Logic and Applications (FPL)*, pages 1–8. IEEE, 2014.

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

[17] Zulfakar Aspar, Zulkalnain Mohd Yusof, and Ishak Suleiman. Parallel huffman decoder with an optimized look up table option on fpga. In *TENCON 2000. Proceedings*, volume 1, pages 73–76. IEEE, 2000.

[18] Jonathan Bachrach, Huy Vo, Brian Richards, Yunsup Lee, Andrew Waterman, Rimas Avižienis, John Wawrzynek, and Krste Asanović. Chisel: constructing hardware in a scala embedded language. In *Proceedings of the 49th Annual Design Automation Conference*, pages 1216–1225. ACM, 2012.

[19] Brian Bailey. *TLM driven design and verification methodology*. Cadence Design Systems, 2010.

[20] Marcus Bednara, Oliver Beyer, Juergen Teich, and Rolf Wanka. Tradeoff analysis and architecture design of a hybrid hardware/software sorter. In *International Conference on Application-Specific Systems, Architectures, and Processors*, pages 299–308. IEEE, 2000.

[21] Guy E Blelloch. Prefix sums and their applications. 1990.

[22] Thomas Bollaert. Catapult synthesis: a practical introduction to interactive c synthesis. *High-Level Synthesis: From Algorithm to Digital Circuit*, 2008.

[23] K.C. Borries, G. Judd, D.D. Stancil, and P. Steenkiste. Fpga-based channel simulator for a wireless network emulator. In *Vehicular Technology Conference*, pages 1–5. IEEE, 2009.

[24] Gary Bradski and Adrian Kaehler. *Learning OpenCV: Computer vision with the OpenCV library*. " O'Reilly Media, Inc.", 2008.

[25] Vladimir Brajovic and Takeo Kanade. A sorting image sensor: An example of massively parallel intensity-to-time processing for low-latency computational sensors. In *International Conference on Robotics and Automation*, volume 2, pages 1638–1643. IEEE, 1996.

[26] Michael Burrows and David J Wheeler. A block-sorting lossless data compression algorithm. 1994.

[27] S. Buscemi and R. Sass. Design of a scalable digital wireless channel emulator for networking radios. In *MILITARY COMMUNICATIONS CONFERENCE*, pages 1858–1863. IEEE, 2011.

[28] B. Buyukkurt and WA Najj. Compiler generated systolic arrays for wavefront algorithm acceleration on fpgas. In *International Conference on Field Programmable Logic and Applications*, pages 655–658. IEEE, 2008.

[29] Raul Camposano. Design process model in the yorktown silicon compiler. In *Design Automation Conference, 1988. Proceedings., 25th ACM/IEEE*, pages 489–494. IEEE, 1988.

[30] Andrew Canis, Jongsok Choi, Mark Aldham, Victor Zhang, Ahmed Kammoona, Jason H Anderson, Stephen Brown, and Tomasz Czajkowski. Legup: high-level synthesis for fpga-based processor/accelerator systems. In *Proceedings of the 19th ACM/SIGDA international symposium on Field programmable gate arrays*, pages 33–36. ACM, 2011.

[31] L Carloni, Fernando De Bernardinis, Claudio Pinello, Alberto L Sangiovanni-Vincentelli, and Marco Sgroi. Platform-based design for embedded systems. *The Embedded Systems Handbook*, pages 1–26, 2005.

[32] Jared Casper and Kunle Olukotun. Hardware acceleration of database operations. In *Proceedings of the 2014 ACM/SIGDA international symposium on Field-programmable gate arrays*, pages 151–160. ACM, 2014.

[33] Leonardo Chang, Ivis Rodés, Heydi Méndez, and Ernesto del Toro. Best-shot selection for video face recognition using fpga. In *Progress in Pattern Recognition, Image Analysis and Applications*, pages 543–550. Springer, 2008.

[34] Chun Chen. Polyhedra scanning revisited. In *ACM SIGPLAN Notices*, volume 47, pages 499–508. ACM, 2012.

[35] Dake Chen and Jiu-Qiang Han. An fpga-based face recognition using combined 5/3 dwt with pca methods. *Journal of Communication and Computer*, 6(10):1–7+, 2009.

[36] Jianwen Chen, Jason Cong, Ming Yan, and Yi Zou. Fpga-accelerated 3d reconstruction using compressive sensing. In *Proceedings of the ACM/SIGDA international symposium on Field Programmable Gate Arrays*, pages 163–166. ACM, 2012.

[37] Junguk Cho, Bridget Benson, and Ryan Kastner. Hardware acceleration of multi-view face detection. In *7th Symposium on Application Specific Processors*, pages 66–69. IEEE, 2009.

[38] Junguk Cho, Shahnam Mirzaei, Jason Oberg, and Ryan Kastner. Fpga-based face detection system using haar classifiers. In *Proceedings of the ACM/SIGDA international symposium on Field programmable gate arrays*, pages 103–112. ACM, 2009.

[39] Jason Cong, Mohammad Ali Ghodrat, Michael Gill, Beayna Grigorian, and Glenn Reinman. Charm: a composable heterogeneous accelerator-rich microprocessor. In *ISLPED*, pages 379–384. ACM, 2012.

[40] Jason Cong, Mohammad Ali Ghodrat, Michael Gill, Beayna Grigorian, and Glenn Reinman. Architecture support for domain-specific accelerator-rich cmps. *ACM Transactions on Embedded Computing Systems (TECS)*, 13(4s):131, 2014.

[41] Jason Cong, Bin Liu, Stephen Neuendorffer, Juanjo Noguera, Kees Vissers, and Zhiru Zhang. High-level synthesis for fpgas: From prototyping to deployment. *Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 30(4):473–491, 2011.

[42] Jason Cong, Peng Zhang, and Yi Zou. Optimizing memory hierarchy allocation with loop transformations for high-level synthesis. In *Design Automation Conference*, pages 1233–1238. ACM.

[43] James W Cooley and John W Tukey. An algorithm for the machine calculation of complex fourier series. *Mathematics of computation*, 19(90):297–301, 1965.

[44] Giovanni De Micheli, David Ku, Frederic Mailhot, and Thomas Truong. The olympus synthesis system. *Design & Test of Computers, IEEE*, 7(5):37–53, 1990.

[45] Giovanni De Micheli and David C Ku. Hercules system for high-level synthesis. In *Proceedings of the 25th ACM/IEEE Design Automation Conference*, pages 483–488. IEEE Computer Society Press, 1988.

[46] Jeffrey Dean and Sanjay Ghemawat. Mapreduce: simplified data processing on large clusters. *Communications of the ACM*, 51(1):107–113, 2008.

[47] L Peter Deutsch. Deflate compressed data format specification version 1.3. 1996.

[48] Giuseppe Di Guglielmo, Christian Pilato, and Luca P Carloni. A design methodology for compositional high-level synthesis of communication-centric socs. In *Proceedings of the The 51st Annual Design Automation Conference on Design Automation Conference*, pages 1–6. ACM, 2014.

[49] DIME-C. User guide. http://www.nallatech.com/.

[50] Yong Dou, Stamatis Vassiliadis, Georgi K Kuzmanov, and Georgi N Gaydadjiev. 64-bit floating-point fpga matrix multiplication. In *Proceedings of the 13th international symposium on Field-programmable gate arrays*, pages 86–95. ACM, 2005.

[51] Frederic Doucet, Sandeep Shukla, Masato Otsuka, and Rajesh Gupta. Balboa: A component-based design environment for system models. *IEEE Transactions on*

Computer-Aided Design of Integrated Circuits and Systems, 22(12):1597–1612, 2003.

[52] V. Erceg, KVS Hari, MS Smith, and D.S. Baum. Channel models for fixed wireless applications, 2001.

[53] Brian P Flannery, William H Press, Saul A Teukolsky, and William Vetterling. Numerical recipes in c. *Press Syndicate of the University of Cambridge, New York*, 1992.

[54] D.D. Gajski, N.D. Dutt, A.C.H. Wu, and S.Y.L. Lin. *High-level synthesis: introduction to chip and system design*. Kluwer Academic UK, 1992.

[55] Christos Gentsos, C-L Sotiropoulou, Spiridon Nikolaidis, and Nikolaos Vassiliadis. Real-time canny edge detection parallel implementation for fpgas. In *ICECS*, pages 499–502. IEEE, 2010.

[56] Nithin George, HyoukJoong Lee, David Novo, Tiark Rompf, Kevin J Brown, Arvind K Sujeeth, Martin Odersky, Kunle Olukotun, and Paolo Ienne. Hardware system synthesis from domain-specific languages. In *24th International Conference on Field Programmable Logic and Applications (FPL)*, pages 1–8. IEEE, 2014.

[57] Nithin George, HyoukJoong Lee, David Novo, Tiark Rompf, Kevin J Brown, Arvind K Sujeeth, Martin Odersky, Kunle Olukotun, and Paolo Ienne. Hardware system synthsis from domain-specific languages. In *International Conference on Field Programmable Logic and Applications*, 2014.

[58] Werner Geurts, Francky Catthoor, and Hugo De Man. Cathedral-iii: Architecture-driven high-level synthesis for high throughput dsp applications. In *Proceedings of the 28th ACM/IEEE Design Automation Conference*, pages 597–602. ACM, 1991.

[59] Maya Gokhale, Jan Stone, Jeff Arnold, and Mirek Kalinowski. Stream-oriented fpga computing in the streams-c high level language. In *Field-Programmable Custom Computing Machines, 2000 IEEE Symposium on*, pages 49–56. IEEE, 2000.

[60] Maya B Gokhale and Janice M Stone. Napa c: Compiling for a hybrid risc/fpga architecture. In *FPGAs for Custom Computing Machines, 1998. Proceedings. IEEE Symposium on*, pages 126–135. IEEE, 1998.

[61] Rajkiran Gottumukkal, Hau T Ngo, and Vijayan K Asari. Multi-lane architecture for eigenface based real-time face recognition. *Microprocessors and Microsystems*, 30(4):216–224, 2006.

[62] Goetz Graefe. Implementing sorting in database systems. *ACM Computing Surveys (CSUR)*, 38(3):10, 2006.

[63] Sumit Gupta, Nikil Dutt, Rajesh Gupta, and Alex Nicolau. Spark: A high-level synthesis framework for applying parallelizing compiler transformations. In *VLSI Design, 2003. Proceedings. 16th International Conference on*, pages 461–466. IEEE, 2003.

[64] Robert J Halstead, Jason Villarreal, and Walid A Najjar. Compiling irregular applciations for reconfigurable systems. *International Jounal on High Performance Computing and Networking*, 2014.

[65] Jeff Hammes, Bob Rinker, Wim Bohm, Walid Najjar, Bruce Draper, and Ross Beveridge. Cameron: high level language compilation for reconfigurable systems. In *Parallel Architectures and Compilation Techniques, 1999. Proceedings. 1999 International Conference on*, pages 236–244. IEEE, 1999.

[66] Mark Harris, Shubhabrata Sengupta, and John D Owens. Parallel prefix sum (scan) with cuda. *GPU gems*, 3(39):851–876, 2007.

[67] Wenhao He and Kui Yuan. An improved canny edge detector and its realization on fpga. In *World Congress on Intelligent Control and Automation*, pages 6561–6564. IEEE, 2008.

[68] Thomas Hill. Acceldsp synthesis tool floating-point to fixed-point conversion of matlab algorithms targeting fpgas. *White papers, Xilinx*, page 18, 2006.

[69] David A Huffman. A method for the construction of minimum-redundancy codes. *Proceedings of the IRE*, 40(9):1098–1101, 1952.

[70] Ali Irturk, Janarbek Matai, Jason Oberg, Jeffrey Su, and Ryan Kastner. Simulate and eliminate: A top-to-bottom design methodology for automatic generation of application specific architectures. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 30(8):1173–1183, 2011.

[71] C.D. Iskander and H.T. Multisystems. A matlab-based object-oriented approach to multipath fading channel simulation. *MATLAB White Paper*, 2008.

[72] Matthew Jacobsen, Richmond Dustin, Hogains Matthew, and Ryan Kastner. Riffa 2.1: A reusable integration framework for fpga accelerators. *ACM Transactions on Reconfigurable Technology and Systems (TRETS)*, 2015.

[73] Ahmed A Jerraya, Inhag Park, and Kevin O'Brien. Amical: An interactive high level synthesis environment. In *Design Automation, 1993, with the European Event in ASIC Design. Proceedings.[4th] European Conference on*, pages 58–62. IEEE, 1993.

[74] Marjan Karkooti, Joseph R Cavallaro, and Chris Dick. Fpga implementation of matrix inversion using qrd-rls algorithm. In *Conference on Signals, Systems and Computers*, pages 1625–1629, 2005.

[75] Shmuel Tomi Klein and Yair Wiseman. Parallel huffman decoding with applications to jpeg files. *The Computer Journal*, 46(5):487–497, 2003.

[76] Donald E Knuth. *The art of computer programming, volume 3: sorting and searching*. Addison-Wesley Professional, 1998.

[77] Dirk Koch and Jim Torresen. Fpgasort: a high performance sorting architecture exploiting run-time reconfiguration on fpgas for large problem sorting. In *Proceedings of the 19th ACM/SIGDA international symposium on Field programmable gate arrays*, pages 45–54. ACM, 2011.

[78] David C Ku and Giovanni De Micheli. Constrained resource sharing and conflict resolution in hebe. *Integration, the VLSI Journal*, 12(2):131–165, 1991.

[79] Christian Lauterbach, Michael Garland, Shubhabrata Sengupta, David Luebke, and Dinesh Manocha. Fast bvh construction on gpus. In *Computer Graphics Forum*, volume 28, pages 375–384. Wiley Online Library, 2009.

[80] Dajung Lee, Janarbek Matai, Brad Weals, and Ryan Kastner. High throughput channel tracking for jtrs wireless channel emulation. In *24th International Conference on Field Programmable Logic and Applications*. IEEE, 2014.

[81] Yew-San Lee, Tsyr-Shiou Perng, Li-Chyun Hsu, Ming-Yang Jaw, and Chen-Yi Lee. A memory-based architecture for very-high-throughput variable length codec design. In *Circuits and Systems*, volume 3, pages 2096–2099. IEEE, 1997.

[82] Yun Liang, Kyle Rupnow, Yinan Li, Dongbo Min, Minh N Do, and Deming Chen. High-level synthesis: productivity, performance, and software constraints. *Journal of Electrical and Computer Engineering*, 2012:1, 2012.

[83] ITU-R Recommendation M.1225. Guidelines for evaluation of radio transmission technologies for imt-2000, 1997.

[84] Rui Marcelino, Horácio Neto, and Joao MP Cardoso. Sorting units for fpga-based embedded systems. In *Distributed Embedded Systems: Design, Middleware and Resources*, pages 11–22. Springer, 2008.

[85] Jason Mars and Lingjia Tang. Whare-map: heterogeneity in homogeneous warehouse-scale computers. In *Proceedings of the 40th Annual International Symposium on Computer Architecture*, pages 619–630. ACM, 2013.

[86] Andrew Martin, Damir Jamsek, and Kanak Agarwal. Fpga-based application acceleration: Case study with gzip compression/decompression streaming engine. In *Special Session 7C, International Conference on Computer-Aided Design*. IEEE, 2013.

[87] Janarbek Matai, Joo-Young Kim, and Ryan Kastner. Energy efficient canonical huffman encoding. In *25th IEEE International Conference on Application-specific Systems, Architectures and Processors*. IEEE, 2014.

[88] Janarbek Matai, Pingfan Meng, Lingjuan Wu, Brad T Weals, and Ryan Kastner. Designing a hardware in the loop wireless digital channel emulator for software defined radio. In *2012 International Conference on Field-Programmable Technology (FPT)*. IEEE, 2012.

[89] Janarbek Matai, Dustin Richmond, Dajung Lee, and Ryan Kastner. Enabling fpgas for the masses. In *First International Workshop on FPGAs for Software Programmers*, 2014.

[90] Deepak A Mathaikutty. *Metamodeling driven IP reuse for system-on-chip integration and microprocessor design*. PhD thesis, 2007.

[91] Michael McCool, James Reinders, and Arch Robison. *Structured parallel programming: patterns for efficient computation*. Elsevier, 2012.

[92] Michael Meredith. High-level systemc synthesis with forte's cynthesizer. *High-Level Synthesis: From Algorithm to Digital Circuit*, 2008.

[93] Giovanni De Micheli. *Synthesis and optimization of digital circuits*, volume 94. McGraw-Hill New York, 1994.

[94] Rene Mueller, Jens Teubner, and Gustavo Alonso. Data processing on fpgas. *Proceedings of the VLDB Endowment*, 2(1):910–921, 2009.

[95] Rene Mueller, Jens Teubner, and Gustavo Alonso. Data processing on fpgas. *Proceedings of the VLDB Endowment*, 2(1):910–921, 2009.

[96] Rene Mueller, Jens Teubner, and Gustavo Alonso. Sorting networks on fpgas. *The VLDB JournalThe International Journal on Very Large Data Bases*, 21(1):1–23, 2012.

[97] Amar Mukherjee, N Ranganathan, J Flieder, and Tinku Acharya. Marvle: A vlsi chip for data compression using tree-based codes. *IEEE Transactions on Very Large Scale Integration Systems*, 1(2):203–214, 1993.

[98] P. Murphy, F. Lou, A. Sabharwal, and J.P. Frantz. An fpga based rapid prototyping platform for mimo systems. In *Thirty-Seventh Asilomar Conference on Signals, Systems and Computers*, volume 1, pages 900–904. IEEE, 2003.

[99] M. Myllyla, M. Juntti, and J.R. Cavallaro. Architecture design and implementation of the increasing radius-list sphere detector algorithm. In *International Conference on Acoustics, Speech and Signal Processing*, pages 553–556. IEEE, 2009.

[100] Stephen Neuendorffer and Kees Vissers. Streaming systems in fpgas. In *Embedded Computer Systems: Architectures, Modeling, and Simulation*, pages 147–156. Springer, 2008.

[101] Hau T Ngo, Rajkiran Gottumukkal, and Vijayan K Asari. A flexible and efficient hardware architecture for real-time face recognition based on eigenface. In *IEEE Computer Society Annual Symposium on VLSI*, pages 280–281. IEEE, 2005.

[102] Rishiyur Nikhil. Bluespec system verilog: efficient, correct rtl from high level specifications. In *Formal Methods and Models for Co-Design, 2004. MEMOCODE'04. Proceedings. Second ACM and IEEE International Conference on*, pages 69–70. IEEE, 2004.

[103] J. Noguera, S. Neuendorffer, S. Van Haastregt, J. Barba, K. Vissers, and C. Dick. Implementation of sphere decoder for mimo-ofdm on fpgas using high-level synthesis tools. *Analog Integrated Circuits and Signal Processing*, pages 1–11, 2011.

[104] Jorge Ortiz and David Andrews. A streaming high-throughput linear sorter system with contention buffering. *International Journal of Reconfigurable Computing*, 2011.

[105] Alexandros Papakonstantinou, Karthik Gururaj, John A Stratton, Deming Chen, Jason Cong, and W-MW Hwu. Fcuda: Enabling efficient compilation of cuda kernels onto fpgas. In *7th Symposium on Application Specific Processors*, pages 35–42. IEEE, 2009.

[106] Heonchul Park and Viktor K Prasanna. Area efficient vlsi architectures for huffman coding. *IEEE Transactions on Circuits and Systems II: Analog and Digital Signal Processing*, 40(9):568–575, 1993.

[107] J.P Paulin, P.G.; Knight. Force-directed scheduling for the behavioral synthesis of asics. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 8:661–679, 1989.

[108] Pierre G Paulin, John P Knight, and EF Girczyc. Hal: a multi-paradigm approach to automatic data path synthesis. In *Proceedings of the 23rd ACM/IEEE Design Automation Conference*, pages 263–270. IEEE Press, 1986.

[109] A Pavan Kumar, V Kamakoti, and Sukhendu Das. System-on-programmable-chip implementation for on-line face recognition. *Pattern Recognition Letters*, 28(3):342–349, 2007.

[110] William B Pennebaker. *JPEG: Still image data compression standard*. Springer, 1992.

[111] Andrew Putnam, Adrian M Caulfield, Eric S Chung, Derek Chiou, Kypros Constantinides, John Demme, Hadi Esmaeilzadeh, Jeremy Fowers, Gopi Prashanth Gopal, Jan Gray, et al. A reconfigurable fabric for accelerating large-scale datacenter services. In *41st International Symposium on Computer Architecture*. ACM, 2014.

[112] Suzanne Rigler, William Bishop, and Andrew Kennings. Fpga-based lossless data compression using huffman and lz77 algorithms. In *CCECE*, pages 1235–1238. IEEE, 2007.

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

[114] I Sajid, MM Ahmed, I Taj, M Humayun, and F Hameed. Design of high performance fpga based face recognition system. In *Proceeding of Progress In Electromagnetic Research Symposium (PIERS)*, pages 504–510, 2008.

[115] Michael Ian Shamos and Dan Hoey. Closest-point problems. In *Annual Symposium on Foundations of Computer Science*, pages 151–162. IEEE, 1975.

[116] Bhaskar Sherigar and K RamanujanValmiki. Huffman decoder used for decoding both advanced audio coding (aac) and mp3 audio, June 29 2004. US Patent App. 10/880,695.

[117] Rudrapatna K Shyamasundar, Frederic Doucet, Rajesh K Gupta, and Ingolf H Krüger. Compositional reactive semantics of systemc and verification with rulebase. In *Next Generation Design and Verification Methodologies for Distributed Embedded Control Systems*, pages 227–243. Springer, 2007.

[118] HDL Simulink. Coder 1.6, 2009.

[119] Sung-Hsien Sun and Shie-Jue Lee. A jpeg chip for image compression and decompression. *Journal of VLSI signal processing systems for signal, image and video technology*, 35(1):43–60, 2003.

[120] DSP Synplify. Synplicity, inc., 2006.

[121] Hamed Tabkhi, Robert Bushey, and Gunar Schirner. Function-level processor (flp): Raising efficiency by operating at function granularity for market-oriented mpsoc. In *IEEE International Conference on Application-specific Systems, Architectures and Processors*. IEEE, 2014.

[122] A. Takach, B. Bowyer, and T. Bollaert. C based hardware design for wireless applications. In *Proceedings of the conference on Design, Automation and Test in Europe-Volume 3*, pages 124–129. IEEE Computer Society, 2005.

[123] Jean-Pierre Talpin, Paul Le Guernic, Sandeep Kumar Shukla, and Rajesh Gupta. A compositional behavioral modeling framework for embedded system design and conformance checking. *International Journal of Parallel Programming*, 33(6):613–643, 2005.

[124] Justin L Tripp, Maya B Gokhale, and Kristopher D Peterson. Trident: From high-level language to hardware circuitry. *Computer*, 40(3):28–37, 2007.

[125] Matthew Turk and Alex Pentland. Eigenfaces for recognition. *Journal of cognitive neuroscience*, 3(1):71–86, 1991.

[126] Nicolas Vasilache, Cédric Bastoul, and Albert Cohen. Polyhedral code generation in the real world. In *Compiler Construction*, pages 185–201. Springer, 2006.

[127] Ganesh Venkatesh, Jack Sampson, Nathan Goulding, Saturnino Garcia, Vladyslav Bryksin, Jose Lugo-Martinez, Steven Swanson, and Michael Bedford Taylor. Conservation cores: reducing the energy of mature computations. In *ASPLOS 2010: Architectural Support for Programming Languages and Operating Systems*, 2010.

[128] J. Villarreal and W.A. Najjar. Compiled hardware acceleration of molecular dynamics code. In *International Conference on Field Programmable Logic and Applications*, pages 667–670. IEEE, 2008.

[129] Jason Villarreal, Adrian Park, Walid Najjar, and Robert 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. IEEE, 2010.

[130] Paul Viola and Michael J Jones. Robust real-time face detection. *International journal of computer vision*, 57(2):137–154, 2004.

[131] Sathaporn Visakhasart and Orachat Chitsobhuk. Multi-pipeline architecture for face recognition on fpga. In *2009 International Conference on Digital Image Processing*, pages 152–156. IEEE, 2009.

[132] K. Vissers, S. Neuendorffer, and J. Noguera. Building real-time hdtv applications in fpgas using processors, axi interfaces and high level synthesis tools. In *Design, Automation & Test in Europe Conference & Exhibition (DATE)*, pages 1–3. IEEE, 2011.

[133] Kazutoshi Wakabayashi. Cyberworkbench: Integrated design environment based on c-based behavior synthesis and verification. In *VLSI Design, Automation and Test, 2005.(VLSI-TSA-DAT). 2005 IEEE VLSI-TSA International Symposium on*, pages 173–176. IEEE, 2005.

[134] Xilinx XC2064. Xc2018 logic cell array, 2009.

[135] J. Xu, N. Subramanian, A. Alessio, and S. Hauck. Impulse c vs. vhdl for accelerating tomographic reconstruction. In *Field-Programmable Custom Computing Machines (FCCM)*, pages 171–174. IEEE, 2010.

[136] Jiyu Zhang, Zhiru Zhang, Sheng Zhou, Mingxing Tan, Xianhua Liu, Xu Cheng, and Jason Cong. Bit-level optimization for high-level synthesis and fpga-based acceleration. In *Proceedings of the 18th annual ACM/SIGDA international symposium on Field programmable gate arrays*, pages 59–68. ACM, 2010.

[137] Zhiru Zhang and Deming Chen. Challenges and opportunities of esl design automation, 2012.

[138] Jacob Ziv and Abraham Lempel. Compression of individual sequences via variable-rate coding. *Information Theory, IEEE Transactions on*, 24(5):530–536, 1978.

[139] Marcela Zuluaga, Peter Milder, and Markus Püschel. Computer generation of streaming sorting networks. In *Proceedings of the 49th Annual Design Automation Conference*, pages 1245–1253. ACM, 2012.

[140] Wei Zuo, Yun Liang, Peng Li, Kyle Rupnow, Deming Chen, and Jason Cong. Improving high level synthesis optimization opportunity through polyhedral transformations. In *International Symposium on Field Programmable Gate Arrays*, pages 9–18. ACM, 2013.