On the Programmability and Performance of OpenCL Designs for FPGA

Anshuman Verma

Thesis submitted to the Faculty of the
Virginia Polytechnic Institute and State University
in partial fulfillment of the requirements for the degree of

Master of Science
in
Computer Engineering

Wu-chun Feng, Chair
Peter Athanas
Huiyang Zhou

May 9th, 2017
Blacksburg, Virginia

Keywords: FPGA, OpenCL, HDL, HLS, AOCL, Verilog,
Accelerator, GEM, Stencil
Copyright 2017, Anshuman Verma

On the Programmability and Performance of OpenCL Designs for FPGA

Anshuman Verma

(ABSTRACT)

Field programmable gate arrays (FPGAs) have been emerging as a promising bedrock to provide opportunities for several types of accelerators that spans across various domains such as finance, web-search, and data center networking, among others. Research interests facilitating the development of accelerators on FPGAs are increasing significantly, in particular, because of their effectiveness with a variety of applications, flexibility, and high performance per watt. However, several key challenges remain that hinder their large-scale deployment. Overcoming these challenges would enable them to match the pervasiveness of graphics processor units (GPUs), their principal competitors in this arena. One of the primary reasons responsible for the slow adaptation by programmers has been the programming model, which uses a low-level hardware description language (HDL). Using HDLs require a detailed understanding of logic design and significant effort to implement and verify the behavioral models, with the latter growing with its complexity. Recent advancements in high-level language synthesis (HLS) tools have addressed this challenge to a considerable extent by allowing the programmers to write their applications in a high-level language named OpenCL. These applications are then compiled and synthesized to create a bitstream that configures the FPGA. This thesis characterizes the efficacy of HLS compiler optimizations that can be employed to improve the performance of these applications.

The synthesized hardware from OpenCL kernels is fundamentally different from traditional hardware such as CPUs and GPUs, which exploit instruction level parallelism (ILP) thread level parallelism (TLP), or data level parallelism (DLP) for performance gains. FPGAs typically use deep pipelining (i.e., ILP) for performance. A stall in this pipeline may severely undermine the performance of applications. Thus, it is imperative to identify and remove any such bottlenecks. To this end, this thesis presents and discusses a software-centric framework to debug and profile the synthesized designs generated using HLS tools. This thesis proposes basic code patterns, including a timestamp and a scalable framework, which can be plugged easily into OpenCL kernels, to collect and process run-time information dynamically. This scalable framework has a small overhead for area utilization and frequency but provides fine-grained information about the bottlenecks and latencies in design.

Additionally, although HLS tools have improved programmability, this may come at the cost of performance or area utilization. This thesis addresses this design trade-off via a comparative study of a hand-coded design in HDL and an architecturally similar, tool-generated design using an OpenCL compiler in the application area of 3D-stencil (i.e., structured grid) computation. Experiments in this thesis show that the performance of an OpenCL approach can achieve 95% of the peak attainable performance of a microkernel for multiple problem sizes. In comparison to the OpenCL approach, an HDL approach results in approximately 50% less memory usage and only 2% better performance on average.

On the Programmability and Performance of OpenCL Designs for FPGA

Anshuman Verma

(GENERAL AUDIENCE ABSTRACT)

A hardware chip consists of switches or transistors, and a modern chip can have a few billions of them. Specifying the interconnection among these transistors and their placement on a chip is a complex problem. To simplify this, the chip-design flow uses automated tools and abstraction at the different levels of the flow, such as architecture, design, synthesis, placement, among others. During design, an engineer specifies the behavioral model in a hardware description language (HDL), which is later used by the automated tools for further processing. Using the HDL requires a detailed understanding of logic design and significant effort to implement and verify the behavioral models, with the latter growing with its complexity. Recent advancements in high-level language synthesis tools have addressed this challenge to a considerable extent by allowing the programmers to write their applications in a high-level language. This thesis characterizes the efficacy of such a tool and available optimizations that can be employed to improve the performance of these applications.

Additionally, this thesis presents and discusses a framework to debug and profile the designs generated using high-level synthesis tools, which can be plugged easily into an application, to collect and process run-time information dynamically. This scalable framework has a small overhead but provides fine-grained information about the bottlenecks in the design. Furthermore, the experiments in this work show that a design generated from a high-level synthesis tool has similar performance when compared to a manual design in HDL, at the expense of area utilization.

Acknowledgments

This thesis would not have been possible without the support of numerous individuals who have been an integral part of the journey that has led me to this point. It is my humble effort to acknowledge their importance in my life and this work.

First and foremost, I would like to thank my advisor Wu Feng, who has nurtured and guided me to navigate the research, provided latitude to pursue my interests and ideas, and constructive comments that have helped me to improve myself as a person and researcher. His work ethic, dedication to research amidst his grueling work hours, and drive to impart education have been a constant source of my inspiration. Seeing him in the lab to attend a morning meeting, after a night of paper reviews, has always inspired and motivated me to work harder.

My journey as a graduate student would not have started had it not been for Poulami Das. Her constant encouragement and support have made my life productive, rejuvenated my once fading dreams and invigorated my resolve to pursue education. Thank you for being the pillar of my strength.

I want to extend my sincere thanks to my mentors and advisors, especially,

- Marius K. Orlowski: for providing me the first experience with research, helping and supporting me to pursue research, and also for financial assistance during my graduate studies.
- Huiyang Zhou: for providing guidance, helping with ideas, and publishing.
- Skip Booth: for providing me the opportunity to intern at Cisco Research and mentoring me during my stay there.
- Mark Gardner: for helping me with infrastructure and resources required to conduct my research.
- Peter Athanas: for providing insightful comments and feedback for the thesis.

I would like to thank my friends and colleagues at Synergy Lab for thoughtful discussion and support that made my stay in lab enjoyable. I appreciate the efforts of the members of

infrastructure team (IT) of Synergy Lab who worked extra hours, amidst their research, to keep the systems up. I acknowledge and thank Konstantinos Krommydas and Umar Kalim for reviewing, correcting, and providing feedback on the manuscripts. Thanks to Hao Wang for providing me the information about the latest and greatest in research and keeping me motivated to dig further. Special thanks to Sarunya ("Kwang") Pumma and Sajal Dash for bearing with my constant complaints and providing me great company.

I thank my friends, Abhinav Gunjal, Vikas BM, Ameya Khandekar, and Akash Agrawal for their camaraderie and companionship that made my stay in Blacksburg fun.

A very special thanks to Biraja Dash, who motivated me to pursue education and supported me as a graduate student and also for being a great friend through the years. Thanks to Amar Singh for helping me to come to Blacksburg and for being there to discuss anything from relevance of religion to physics since childhood.

Finally, I can not thank enough my loving parents Banarasi and Sadhana for believing in me and providing unconditional love and support. Thanks to my younger brothers, Ankur (Manu) and Anuj (Sonu) for being supportive during my days at school.

Anshuman Verma
May 2017, Blacksburg, VA

Contents

1 Introduction **1**

1.1 Introduction 1

1.2 Related Work 4

1.3 Contributions 10

1.4 Thesis Organization 11

2 Background **13**

2.1 Architecture 13

2.1.1 FPGA 13

2.1.2 Board 15

2.2 Programming Environments 16

2.2.1 OpenCL 17

2.2.2 Altera Offline OpenCL Compiler 20

2.3 Applications 24

2.3.1 GEM 25

2.3.2 3D-stencil 25

3 OpenCL Compiler Optimizations **27**

3.1 Performance Characterization 29

3.1.1 Stencil 29

3.1.2 GEM 31

3.2 FPGA Optimizations and Insights 31

3.2.1 3D-stencil Benchmark 32
3.2.2 GEM Benchmark 39
3.3 Fixed-plane-size Accelerator for 3D-stencil 41
3.3.1 Re-factored Algorithm 42
3.3.2 Cyclic Buffer Implementation 43
3.3.3 Loop Unrolling 44
3.3.4 Evaluation 45
3.4 Aligned Data Transfers: Host-side Optimization 46

4 A Scalable Framework for Profiling **47**
4.1 Primitive Patterns 48
4.1.1 Timestamp 48
4.1.2 Sequence Number 51
4.2 Framework 54
4.3 Use Cases 56
4.3.1 Pipeline Stall Monitors 57
4.3.2 Smart Watchpoints 59
4.4 Evaluation 61

5 An Accelerator for 3D-stencil **63**
5.1 Peak Performance for a Memory-Bound Kernel 64
5.2 OpenCL Accelerator 65
5.2.1 Problem Fragmentation 66
5.2.2 Column Transformation 67
5.2.3 Accelerator Design 69
5.2.4 On-chip Memory Management 72
5.3 HDL Accelerator 73
5.4 Evaluation 76

6 Conclusions and Future Directions **82**

6.1 Conclusion 82
6.2 Future Directions 83

Bibliography **85**

List of Figures

1.1 Performance per watt for different hardware design choices. (From Bob Brodersen, Berkeley Wireless Research Center, UCB) 3

1.2 Thesis chapter dependencies. 12

2.1 Stratix V ALM (from device handbook [1]) 15

2.2 Bittware s5phq_d8 board that uses Altera Stratix V FPGA (images from Bittware manual) 16

2.3 OpenCL memory model (from OpenCL specification 2.0 [2]) 19

2.4 Compilation flow for OpenCL kernels (from Altera programming guide [3]) . 21

2.5 A cell of 3D-stencil grid that uses current value of its six-neighbors to compute the value of cell for next iteration. 26

3.1 Architecture-agnostic kernel performance for Stencil and GEM normalized to Tesla-K20Xm (lower is better) 30

3.2 Optimized 3D-stencil kernel implementations (higher is better) 33

3.3 Single work-item kernel, manual and compiler vectorization, multiple compute units: profiling (bandwidth, ALM usage, memory stalls, and clock frequency) 35

3.4 Data sharing in X direction, and algorithmic refactoring: profiling 37

3.5 Optimized GEM kernel implementations 40

3.6 Speedup and profiling information for unroll 45

3.7 Data transfer time 46

4.1 An overview of the iBuffer structure. 48

4.2 The execution/scheduling order of the loop iterations or work-items, (a) for Listing 4.6 and (b) for Listing 4.7. 54

4.3 The state machine for an iBuffer. States can be changed by writing commands into the command channel, or by completion of tasks in the iBuffer. 55

4.4 Monitoring pipeline-stalls using timestamps and the iBuffer. 58

4.5 Smart watchpoints using timestamp and the iBuffer. 61

5.1 Performance of micro-benchmark kernel with different unroll factors (one, two, four, eight, and twelve) for problem sizes 256x256x256 (P), 256x256x512 (2P), 256x256x1024(4P). It performs best with unroll factor of eight, after which unrolling inner loop has diminishing returns. 65

5.2 Speed-up with respect to a naive single work-item 3D-stencil kernel and corresponding area, bandwidth utilization, and clock frequencies using different compiler optimizations (discussed in Chapter 3). Horizontal line at speedup 8.3x represents the performance of micro-benchmark kernel for the same problem size. 67

5.3 Dividing 3D-stencil problem bigger than accelerator plane size into subproblems. Dashed lines represent boundary elements for each subproblem that are transformed by the host. 68

5.4 Stencil accelerator comprising of single work-item kernels that communicate with each other using the Altera-channels. 69

5.5 On-chip memory data layout for two consecutive fetch cycles. (a) Memory P1 provides the down data, memory P2 provides bottom data, top data is fetched from DRAM. Memories CB0 and CB1 provide center and north data. (b) Top data and south data fetch in the previous cycle gets stores in memory P0, and CB1 respectively. All other elements required for computation are fetched from the memory location next to previously accessed location. . . . 73

5.6 Block diagram for the HDL implementation of stencil kernel that connects to memory arbiter using Avalon memory ports 75

5.7 (a) State machine for address generation (b) Floating point unit for 3d-stencil computation 76

5.8 Speed-up for the OpenCL accelerators for different plane size with respect to micro-benchmark kernel. Results are reported for problem sizes of 64x64x4096 (P_64_cubed), 128x128x128 (P_128_cubed), 256x256x256 (P_256_cubed), and 512x512x512 (P_512_cubed). (Note: Bars corresponding to plane sizes smaller than accelerator plane size are not reported) 77

5.9 Memory, ALM, RAM usage, and clock frequencies for the OpenCL-accelerators with different plane sizes 78

5.10 Speed-up for OpenCL (OCL-256x256 and naive task implementation SWI) and RTL (HDL-256x256) accelerators with respect to micro-benchmark kernel performance for problem sizes of 512x512x512 (P_512_cubed), 768x768x768 (P_768_cubed), 1024x1024x256 (P_1024_256), and 1280x1280x256 (P_1280_256). 79

List of Tables

2.1 Stratix V FPGA details 16
2.2 OpenDwarfs benchmarks used 25

3.1 Specification of test architectures 29
3.2 GEM kernel implementation's features 42

4.1 The logic, memory usage, and frequency results. 62

5.1 The logic, memory usage, and frequency comparison for OpenCL and RTL accelerators for plane size 256x256. 79
5.2 The memory bandwidth utilization for different accelerators. 80

Listings

2.1 Serial pseudo-code for three-dimensional seven-point stencil 26
3.1 Single work-item (SWI-1) 34
3.2 Work-group size one (RWG-1) 34
3.3 Shift register implementation in OpenCL 43
3.4 Pseudo-code for 3D-stencil implementation 44
4.1 The timestamp pattern using a persistent autorun kernel with a free-running counter. 49
4.2 Read site(s) of the timestamp. 49
4.3 The timestamp pattern using a verilog module containing a free-running counter and its OpenCL interface. 50
4.4 The read sites(s) of the timestamp using HDL counters. 51
4.5 Sequencing number: The persistent kernel containing a sequence counter. . . 52
4.6 Sequencing number: The read site of the sequence number in a single-task kernel. 53
4.7 Sequencing number: The read site of the sequence counter in an NDRange kernel. 53
4.8 Autorun iBuffer persistent kernel. 57
4.9 Measuring load latency using stall monitor. 58
4.10 Host interface kernel to forward control commands from host to iBuffer and to read data from iBuffer. 59
4.11 Adding a watchpoint for a specified address and monitoring memory operations. 60
5.1 Micro-benchmark kernel for assessing peak performance 64
5.2 Pseudo-code for compute 3D-stencil kernel 71

5.3 Stencil kernel with RTL library 74

Chapter 1

Introduction

1.1 Introduction

Field programmable gate arrays (FPGAs) have gained significant attention in the research community since their foray into the computing landscape through Catapult [4], a project that enabled acceleration of the Microsoft Bing search algorithm using a reconfigurable fabric. The recent trend of integrating FPGAs into server processors also makes them a desirable reconfigurable accelerator for general-purpose computations.

Research has been conducted in both academia and industry to use FPGAs as a solution to combat bottlenecks and accelerate many diverse and complex applications. One such work involves the acceleration of the memcached [5] operation to target server applications. Another example is Cryptorapter [6], a high performance, low power, and reconfigurable cryptographic processor with both ASIC and FPGA implementations, highlighting the growing interests in reconfigurable computing. In addition, FPGAs are finding a place in accelerating [7] deep convolution neural networks,and more generally, machine learning, e.g., Tabla [8],

a template-based framework for accelerating statistical machine learning on FPGAs. Virtualized FPGA accelerators [9] have proven themselves to be suitable for cloud computing even with the presence of virtualization overhead over software. For decades, the streaming architecture of FPGAs has delivered accelerated performance across many application domains, such as option pricing solvers in finance [10], computational fluid dynamics in oil and gas [11], and packet processing [12] in network routers and firewalls.

All these are indications that FPGAs will continue to play a crucial role in the space of customizable accelerators over the coming years and will be a fundamental component of heterogeneous computing, thus using the on-chip resources of a FPGA efficiently is essential. FPGAs have been primarily used as an alternative to application-specific integrated circuits (ASICs). ASICs provide the best performance per watt (Figure 1.1) but offer little or no flexibility. On the other hand, FPGAs offer more flexibility but lesser performance per watt. Additionally, with the fast pace of growth in the areas of emerging workloads such as machine learning (ML) and artificial intelligence (AI), an ASIC solution may already be obsolete by the time it is in production due to its long design and manufacturing time.

Traditionally, an FPGA is configured using a bitstream this is created by synthesizing the applications written in hardware description languages (HDLs). Designing applications in HDLs to configure FPGAs involve significant logic design and verification knowledge, engineering effort, and time. It gets worse for complex applications. As a result, programmability for FPGAs remains a concern, and Bacon et al. [13] note that it must improve for their mass deployment.

To enhance the programmability, multiple high-level synthesis tools [14] have been introduced by companies and academia. For example, C-based Chisel [15] provides a Scala-based platform for HLS. Catapult-C [16], Cadence C-to-Silicon Compiler, and Synopsys Symphony C Compiler are C-based compilers to generate HDLs. Now, there has been the effort to use

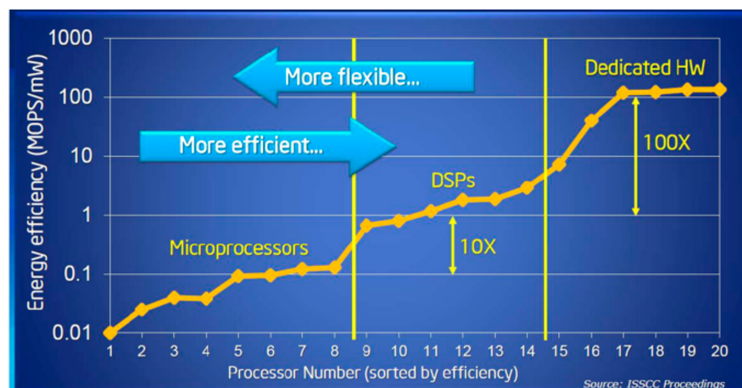


Figure 1.1: Performance per watt for different hardware design choices. (From Bob Brodersen, Berkeley Wireless Research Center, UCB)

languages that have integrated parallelism models, such as OpenCL [2].

Recent advances in high-level language compilers, such as Altera-OpenCL [3] and Xilinx SDAccel [17], allow users to generate hardware designs from applications written in OpenCL, facilitating the use of a higher level of abstraction without getting into the low-level details of digital logic design. A programmer specifies OpenCL kernels, from which the OpenCL-for-FPGA compiler synthesizes it into a bitstream. It either leverages explicit thread-level parallelism (TLP) or extracts implicit loop-level parallelism (LLP) from kernel functions to achieve high throughput computation. The small overhead in fetching instructions and deeper pipelines make FPGA faster [18] than CPUs and even GPUs.

A key challenge emanates from how differently the synthesized hardware operates from an apparently sequential processor. The synthesized hardware is fundamentally parallel and either TLP or LLP is converted to pipeline-level parallelism. Although highly promising, significant challenges remain for such OpenCL-for-FPGA design paradigms to be widely adopted, primarily by software/system designers. It is essential to provide software developers with facilities to see how operations are executed and reason about the execution results as well as the achieved performance. To this end, we present our work on developing run-time

dynamic profiling and debugging support in OpenCL for FPGAs.

To improve the performance of OpenCL kernels on FPGAs, we identify general techniques to optimize OpenCL kernels for FPGAs under device-specific hardware constraints. We then apply these optimizations techniques to the selected applications from the OpenDwarfs benchmark suite, which has various parallelism profiles and memory-access patterns to evaluate the effectiveness of the optimizations with respect to performance and resource utilization. We present the performance of the structured grid and N-body dwarf-based benchmarks in the context of various optimizations.

Although a synergistic combination of compiler, board support package, PCIe drivers, hardware design toolchain, and a high-level language to write applications makes FPGAs more accessible to software programmers, this programmability may come at the expense of area usage or performance. That is, we trade the FPGA design space for enhanced programmability using high-level language synthesis. We address this design trade-off for a regular, memory bound, stencil application.

1.2 Related Work

Compiler Optimizations

Over the years, multiple approaches have been proposed to address the challenges of programming FPGAs by abstracting the hardware implementation details using high-level programming models. Traditional high-level synthesis (HLS) tools, such as Catapult C, AutoPilot, Handel-C, and C-to-Silicon, use a variation of C/C++ with non-standard and unique language extensions [19]. Moreover, these C-like programming models are inherently se-

quential and lack the specifications for concurrency and data transfer, which usually result in poor hardware design [20]. Unlike traditional HLS approaches, OpenCL is a standard and portable programming model that is explicitly parallel. This programming model allows software developers to control the parallelism at different levels of granularity and manage data movement through the memory hierarchy. Hence, OpenCL has the potential to generate efficient hardware designs that match the characteristics of the applications [21].

Several research studies investigated the use of this HLS framework after the release of the Altera OpenCL compiler in 2013; the first OpenCL-to-FPGA compiler to pass Khronos conformance. Some of them are in diverse application domains including option pricing [22], information filtering [23], embedded systems [24], DSP and image processing [25, 26, 24].

Inggs et al. [22] used an options pricing benchmark (with Black-Scholes and Heston models) to evaluate the performance of several HLS tools including Altera OpenCL. The authors analyzed several optimization techniques such as pipelining, task-level parallelism, and data blocking. The authors showed that FPGAs (Stratix V GXA7) achieve two orders of magnitude speedup over a sequential CPU (Intel Core-i7 2600) when programmed with an optimized OpenCL implementation. In [23], Chen et al. demonstrated an efficient FPGA design of a document filtering algorithm, implemented using optimized OpenCL kernels. The authors explored the OpenCL optimization space on FPGAs (Altera Stratix IV 530) and showed that their performance-per-watt outperforms CPU (Intel Xeon W3690) and GPU (NVIDIA Tesla C2075) by a factor of 5x.

In [25], the authors studied the performance and programmability of OpenCL in comparison with VHDL using three image processing kernels. The results showed that OpenCL improves the productivity by six-folds while consuming up to 70% more resources than VHDL designs. Chen et al. [26] evaluated CPUs (Intel Xeon W3690), GPUs(NVIDIA Fermi C2075), and FPGAs (Altera Stratix IV 530 and Altera Stratix V 5SGXA7) using fractal video compres-

sion application implemented in OpenCL. The authors showed that with application-specific optimizations, FPGAs achieved 3x and 114x performance speedup over the GPU and CPU, respectively.

CHO [24] is an OpenCL benchmark suite for FPGAs that contains workloads implemented using the single work-item programming model. Although CHO kernels belong to real-world applications, they are limited to the DSP and embedded computing domain, which is the traditional domain for reconfigurable computing architectures. Since the benchmark suite is intended to evaluate the effectiveness of FPGA HLS tools, the authors do not provide optimized kernels.

Unlike the previous approaches, we address the challenges of programming FPGAs using OpenCL in high-performance computing (HPC) and scientific computing workloads, which are the traditional domain for many-core accelerators such as GPUs and Intel MIC.

Profiling and Debugging

Debugging circuits and interfaces are well-studied topics, and a plethora of research work is continuing in this area. Bart et al. [27] use trace buffers for creating an on-chip debugging infrastructure. Goeders and Wilson [28, 29] propose efficient trace-buffer structures and ways to improve the information storing in them. Hung [30] develop mechanisms to predict what signals will be useful during debugging. Similar to work by Calagar et al. [31], these mainly utilize logic analyzers such as SignalTap to collect debug information.

To verify the design generated by high-level synthesis tools, Yang et al. [32] develop a framework that uses just-in-time traces and inserts debugging logic into tool-generated RTL code. To address the integration of custom logic with HLS tool-generated designs, Monson et al. [33]

develop tools to map the user instructions and memory addresses to signals in creating a "software-like" debugging interface.

None of these works address the "OpenCL for FPGA" design paradigm. Altera [3] provides profiling support for OpenCL for FPGA designs, which is inserted into the generated logic during synthesis and manifests information on the accumulated bandwidth and channel stalls. In comparison, our proposed framework provides detailed insight into synthesized designs and supports smart debugging functions.

Stencil Accelerator

Hasitha et al. [34] design a stencil accelerator for OpenCL and develop a methodology to generate optimal stencil accelerators. This work calculates the next iteration value for multiple cells per cycle termed as "cell-parallelism" in an iteration of computation. Independent grid cells across an iteration are computed in parallel. In contrast, we compute multiple cells in an iteration to achieve optimal bandwidth. Their approach requires design compilation and generating bit-streams as the plane-size of problems change. Our work addresses this bottleneck by fragmenting the problem and thus reducing the need to re-compile the OpenCL designs whenever the plane-size changes. However, if the problem plane-size does not match the accelerator size, stride accesses to DRAM memory suffer from inefficient bandwidth utilization. Our result indicates that an accelerator of plane size 256x256 does not suffer from this limitation. In our experiments, we were unable to fit an accelerator of plane-size 512x512 on the Stratix V due to on-chip memory constraints. The limited real-estate availability makes problem fragmentation imperative to compute stencils larger than 256x256 plane size. In [34], the authors provide details primarily for single-precision computation, so we can not compare their results directly with ours. They also report results

for double-precision 3D-seven points, but the number of "Pcells" used for their computation is unavailable.

Sano et al. [35] did a comprehensive study for stencil accelerators targeted for FPGAs to develop a multi-FPGA accelerator using HDL. This work aims at accelerating stencil computation using multiple FPGA boards connected using an IO interface. The accelerator design we present is similar in design to a single scalable streaming array (SSA), but we address the stencil computation in the context of the performance-programmability gap using OpenCL and HDL.

Jia and Zhou [36] explore compiler optimizations to tune the code for 2D and 3D stencil computation. This solution uses a shift register to create a single streaming array for optimization. The optimization is similar to SSA [35] and cell-parallelism [34] in the creation of a library. The presented approach in this thesis uses the manual tiling of on-chip memory for a 3D-stencil code to solve large-dimension stencil problem. Using the sliding window or shift register approach would require a significant amount (i.e., two times the length x width of stencil problem) of on-chip memory resource. These resources may not be available on-board. However, the shift register approach is well suited to solve 2D-stencil problems as the maximum number of elements to be stored in on-chip memory are limited by its dimension in the x-direction.

Zohouri et al. [37] characterize six applications from the Rodinia [38] benchmark. They primarily detail the possible compiler optimizations and their efficacy in the context of these applications. In contrast, our work details a re-factored 3D-stencil algorithm in OpenCL that achieves close to peak performance, and its comparison to an equivalent RTL approach.

A later published study by Wang et al. [39], where they propose the use of task-based kernels connected by channels and development of a data partitioning scheme for database

applications, corroborates our work with stencil kernels and the reasoning to partition the kernels as concurrently running kernels connected via channels for the optimal performance.

In [40], the authors studied the performance and programmability of OpenCL in comparison with VHDL using three image processing kernels. The results showed that OpenCL improves the productivity (time to develop and verify the design) by six-folds while consuming up to 70% more resources than VHDL designs. We address the same problem in the context of 3D-stencil kernel, and our studies show that OpenCL design consumed approximately 50% additional resources while suffering a minor 2% performance penalty.

Synthesis and Routing Improvements

Although HLS tools improve programmers productivity, long design compilation times are a hindrance for FPGAs democratization [13]. Apart from their cumbersome programmability, some of the well-known bottlenecks for widespread adaptations of FPGAs are following:

1. Significant compile time (in the order hours) for FPGAs, when compared to a few seconds for typical CPU or GPU programs.
2. Time to configure (order of milliseconds) the FPGAs that limits the context switching.

Although addressing the reduction in compilation time for FPGAs is outside the scope the of this thesis, we present some of the work that is trying to address these problems. One of the ways to reduce this compilation time is to use placed and routed macroblocks during the synthesis process [41, 42, 43]. Tessier [44] presents an approach by breaking the problem into a clustering of steps that starts with the placement of macros and gets refined in the timing constraints. These techniques have shown to reduce the synthesis and place and route time by 3x-5x while incurring a performance penalty with respect to area and clock frequency.

Partial reconfiguration [45] can reduce the time to configure the FPGA by uploading a partial bitstream, which is supported by most of the recent FPGAs. For example, Feilen et al. [46] present a data partitioning scheme for a reconfigurable broadcasting digital receiver and demonstrate its benefits. Wisniewski et al. [47] detail the partial reconfiguration techniques for controllers. Advancement in these approaches has the potential to lower the barrier of entry for the adoption of FPGAs for the masses.

1.3 Contributions

This thesis makes the following significant contributions:

- **Extensive characterization of FPGA-specific compiler optimizations along the dimensions of performance and resource utilization.** Our results indicate that only modest improvements are possible with compiler optimizations with over 80% of the performance remaining untapped [48, 49].
- An open-source software-centric framework to profile synthesized OpenCL kernels with minimal overhead [50].
- **A custom accelerator design for 3D-stencil computations and its implementation in OpenCL, which achieves about 95% of the theoretical peak performance**.
- **An implementation of the accelerator mentioned above in HDL and a comparison against an OpenCL compiler-generated design**. Our experiments demonstrate that a carefully designed kernel in OpenCL achieves similar performance as the HDL design but at the expense of area utilization.

Fundamental Contributions: Existing work on source-level debugging for FPGAs [31] are mainly build upon logic analyzers such as SignalTap [51] by Altera and ChipScope [52] by Xilinx. The logic analyzers observe selected signal values, which are then stored in a trace buffer to record a window of execution. Compared to these works, our proposed scheme is built in OpenCL and does not depend upon any logic analyzer. More importantly, our software-centric approach enables intelligent data processing rather than merely recording the selected signals. To the best of our knowledge, our proposed scheme is the first open-source design library that is entirely coded in OpenCL for profiling and debugging HLS designs.

1.4 Thesis Organization

Rest of the document is organized as follows: Chapter 2 articulates the necessary background for the FPGAs, OpenCL programming model, and Altera-specific compiler features. Compiler features, their evaluation, and efficacy are presented in Chapter 3. A scalable debugging and profiling interface for synthesized kernel designs and its performance and overhead assessment is detailed in Chapter 4. This open-source debugging framework is a library that can be easily plugged in with kernels to provide fine-grained information about potential bottlenecks. Chapter 5 presents an accelerator for 3D-stencil computation implemented in both OpenCL and HDL. These two different design processes capture the trade-off between programmability and performance. The findings of this thesis are summarized in Chapter 6 and a discussion for future work that can be built upon the work presented in this document.

Figure 1.2 represents the dependencies among chapters. After this introductory chapter, Chapter 2 presents background material that is required to understand the rest of the thesis. Chapter 3 explores OpenCL compiler options for optimizing an OpenCL program for FPGA.

Next, Chapter 4 presents our profiling framework, which when used in conjunction with OpenCL can help to create a highly-efficient and stall-free accelerator that is similar to the custom-designed 3D-stencil in Chapter 5. We expect and hope that the findings in this thesis can be generalized to create efficient designs for FPGAs using OpenCL compilation tools.

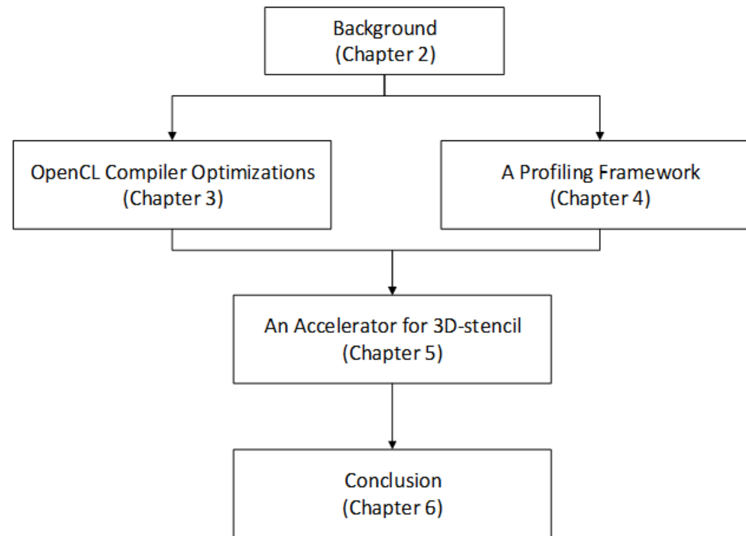


Figure 1.2: Thesis chapter dependencies.

Chapter 2

Background

In this chapter, we provide the architectural details of the FPGA, board configuration, and a brief introduction to OpenCL programming environment and Altera compiler. Furthermore, this chapter provides the details about the applications explored in this thesis, in the context of a high-level synthesis tool and its efficacy.

2.1 Architecture

2.1.1 FPGA

Field programmable gate arrays (FPGAs) are reprogrammable circuits which use small building blocks called adaptive logic modules (ALMs) (Figure 2.1 provides a high-level block diagram for Altera Stratix V). An ALM consists of the combinational logic block, register(s) the look-up table (LUT), and may additionally have few other logic elements such as adders, fast arithmetic units and carry chains. Each ALM has multiple outputs which can be programmed to be a function of an ALM's inputs by configuring the values in the LUT. The

number of input and output ports for an ALM varies among vendors and device families. These ALMs are used to create a complex logic block (CLB), which connects to a complex switching matrix. This programmable switching matrix determines the signal route to interconnect the FPGA components. Routing the clock signals to registers is susceptible to issues related to clock skew, that may result in timing violations. Dedicated routes for clocks in a device avoid this problem. Additionally, FPGAs also have on-chip memory blocks and may have few other complex function-hardened macros such as floating-point operators. Having these hardened logic blocks helps to reduce the ALM consumption and create efficient designs.

The programmable logic and interconnect on FPGA is configured via a *bitstream* to create a hardware system by writing bits into LUTs and switching network control bits. Hence it needs reconfiguration when the device goes through a power cycle. Recent FPGAs allow partial reconfiguration to modify just a few bits and update the realized hardware.

Finding the required bitstream to configure each ALM and routing network manually is possible, but extremely hard because a modern FPGA has hundreds of thousands to a few million ALMs (Table 2.1). Fortunately, this process is automated and done by tools. The user designs the circuit in HDL and verifies it. Once satisfied with functionality, the design can go through the synthesis, timing, and placement and route tools. The synthesis tool transforms the HDL behavioral models to a gate-level design by utilizing the user provided timing constraints that specify the clock frequency, input and output delays, and timing-related parameters. The timing tool checks if all the timing paths from a launch point to a capture point are satisfied for various operating and manufacturing variations and reports if there are any violations. The user can fix these timing paths by employing various techniques such as redesigning the critical paths, replicating the logic block to reduce the fan-out, or relaxing the operating frequency. Afterwards, the design goes through the placement and

routing. During this step, the tool tries to optimize the placement of logic elements and their mapping to ALMs that would result in the best performance. Once all these steps are complete, the tool can generate a bitstream. This bitstream configures the FPGA.

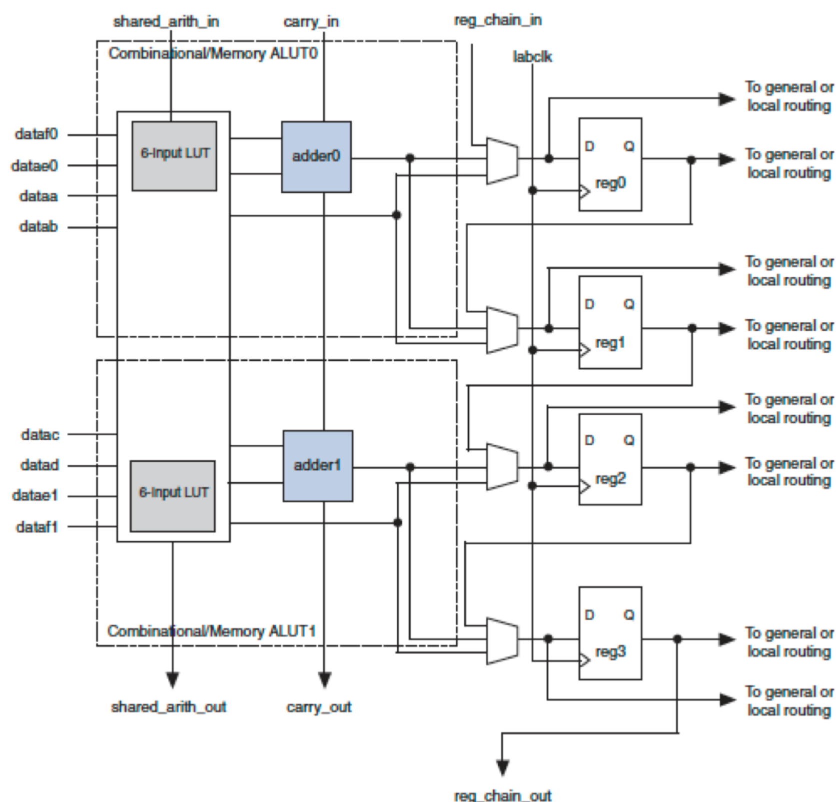


Figure 2.1: Stratix V ALM (from device handbook [1])

2.1.2 Board

The experiments in this thesis are conducted on a half-length PCIe board from Bittware (s5phq_d8), comprising a Stratix V FPGA (details listed in Table 2.1) from Altera. Figure 2.2 shows an image of the board and its block diagram. The host communicates with FGPA and configures it using PCIe interface. The board has a PCIe Gen-2 interface that supports a high-speed data transfer of 500MB/s per lane. To store the data on-board, it has an 8GB DDR3 DRAM. Additionally, it has a board management controller (BMC) to monitor

and configure the FPGA image, along with an I2C and JTAG port.Bittware also provides OpenCL board support package (BSP) that is compatible with the Altera compiler, which helps the compiler to integrate the compiled kernels with the board hardware seamlessly.

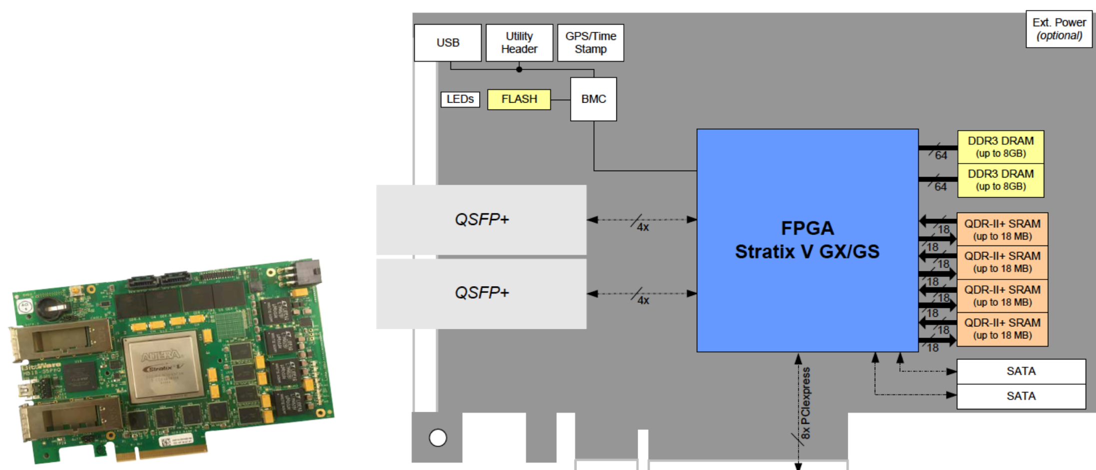


Figure 2.2: Bittware s5phq_d8 board that uses Altera Stratix V FPGA (images from Bittware manual)

Table 2.1: Stratix V FPGA details

Device	**ALM**	**Block Memory(bits)**	**DSP Blocks**
5SGSMD8K2F40C2	262,400	52,572,160	1963

2.2 Programming Environments

In 2005, when Intel announced the cancellation of their efforts to develop a 4-Ghz, single-core processor and moved to a multi-core strategy [53], it marked the arrival of multi-core processors. The reasons for this paradigm shift included the power wall and the diminishing returns from instruction-level parallelism (ILP) [54]. It was no longer possible to run a processor at ever-increasing clock frequency and reap the benefits of Dennard scaling [55]. This paradigm shift caused the architects to look beyond serial processors as the mainstream

substrate. As a result, various devices such as GPUs, vector processors, accelerators, fused architectures (CPU+GPU, CPU+FPGA), and various other heterogeneous platforms are becoming more prevalent.

NVIDIA GPUs can be programmed in at least four programming environments: OpenGL [56], CUDA: Compute Unified Device Architecture [57], OpenACC [58], and OpenCL [59]. OpenGL, short for the Open Graphics Library, defines a way for GPUs to accelerate rendering. Its programmability for general-purpose GPU computing is very low as one must program general computation via graphical constructs. CUDA is a low-level programming interface that democratized the notion of general-purpose computation on the GPU. To make programming such GPUs easier, NVIDIA introduced a higher-level, directive-based programming model called OpenACC [58]. For FPGA devices, the main drawback of all the above programming models is that they only work on GPUs, and in the latter two cases, only NVIDIA GPUs.

With the proliferation of different types of parallel-computing devices, however, there existed a need for a common parallel-programming model that worked across these different devices. As such, in 2008, OpenCL [59] was introduced. OpenCL supports the programming of CPUs, GPUs, and with the recent advances with HLS, now even FPGAs [60]. Thus, in the following sections, we will present and discuss this programming language and the Altera OpenCL compiler, followed by the case-study applications used in this thesis.

2.2.1 OpenCL

In the rest of this thesis, we use OpenCL terminology frequently, and because this language is still not pervasive for FPGA programming, we provide some background here. For deeper insights and understanding, we refer the reader to OpenCL manual [59].

The OpenCL device model consists of a host and the compute devices. Each of the computing devices has one or more compute units (CUs), and each CU comprise one or more processing elements (PEs). The host controls the computation off-loading onto the devices and data movement between host and devices. The host can build the program to run on a device at run time or use a pre-compiled program. The programming model provides application program interfaces (APIs) to compile and build the applications, data transfers, and computations done by kernels.

A *kernel* is a function instance in OpenCL. A host program executes a *kernel* in an N-dimensional space, i.e., *NDRange*; which can be either one, two, or three dimensional. A single instance of a *kernel* at a point in the index space is called a *work-item*. Multiple *work-items* are grouped in a *work-group* that are conceptually scheduled together. A kernel specified to run as a one-dimensional kernel for a work-group size of one and launched for a single workgroup is called a *single work-item* kernel or *task*. A `barrier` in an NDRange kernel is a synchronization point for all the work-items in a work-group. The execution of statements after the barrier continues when all work-items in a work-group reach the barrier.

Kernels are launched on a device by using a *queue*, which is created for a device within a *context*. Kernel execution for a queue is serialized, i.e., kernels are executed in a first-in-first-out manner, and the next kernel is started only when the previously launched one finishes execution. Since a FPGA can run multiple kernels concurrently, one can create multiple command queues for the device and start each concurrent kernel in a separate command queue. FPGAs also support built-in kernels on the device. These can start automatically, or the host can launch them.

A kernel on a device accesses the device memory to process the data. The OpenCL memory model specifies the memory hierarchy and its consistency model. The OpenCL memory model [61] divides memory in two parts (a) host memory (b) device memory. Host memory is

a directly-available memory in the host, and device memory is available to kernels executing on the compute device. Device memory has a four-level memory hierarchy (Figure 2.3) and supports weak consistency model [62]. *Global memory*: allows read-write access to all the work-items executed in work-groups on a device. *Constant memory*: has read-only access for all work-items and remains constant during the execution. *Local memory*: allows access to work-items that are within a work-group, and *Private memory*: provides access to a work-item; other work-items in the work-group do not have access to this private memory. For FPGAs, global memory can be implemented by DRAM, while local memory and constant memory are implemented using on-chip memory resources. Private memory can be implemented using either on-chip memory or registers in ALMs.

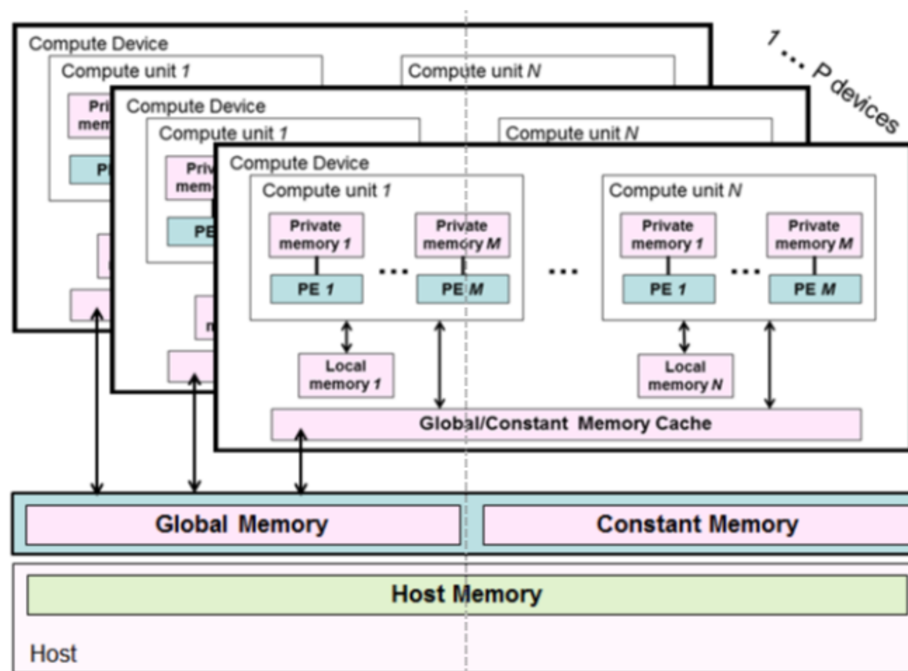


Figure 2.3: OpenCL memory model (from OpenCL specification 2.0 [2])

2.2.2 Altera Offline OpenCL Compiler

The Altera offline OpenCL compiler (aocl) takes the OpenCL kernels as inputs and compiles them to generate a synthesized hardware for an FPGA. Altera recommends creating a consolidated file for all the kernels in an application (Figure 2.4) and use "aocl" tool to compile this file into a binary file (with .aocx suffix). The host program uses this binary file to configure the FPGAs during run time with the help of memory mapped device (MMD) drivers [63]. The MMD drivers interact with the FPGAs using the PCIe interface. It can configure the FPGAs, transfer the data between host and device, and launch the kernels on the FPGA. These drivers are seamlessly integrated with inbuilt OpenCL functions for data-transfer, kernel launch, and build.

The kernels are compiled offline before running the application on the host. This process takes a significant amount of time (in order of hours), depending on the complexity of the kernel. Owing to high compilation time, the penalty for finding a bug post-compilation is significant. To mitigate this problem, Altera provides an emulation tool that can be employed while writing the application to find functional bugs. Once the user is satisfied with the functionality in the emulator, he or she can proceed with the full-compilation.

During the full-compilation phase, the tool generates an HDL design for the input kernels and integrates it into a board support package (BSP) provided by the board vendor. The BSP has PCIe, DRAM interfaces, and different input and output interfaces, which help the tool to integrate the design generated from kernels and a specific board. This separation between the kernels and the BSP allows the same compiler to generate the designs compatible with multiple boards.

This integrated design comprising of the compiled kernels and BSP goes through synthesis, placement and routing, and timing analysis during the full-compilation phase. The compiler

tool decides the maximum operating frequency F_{max} after timing analysis is complete and configures the clock generation module to run the kernel at F_{max}.

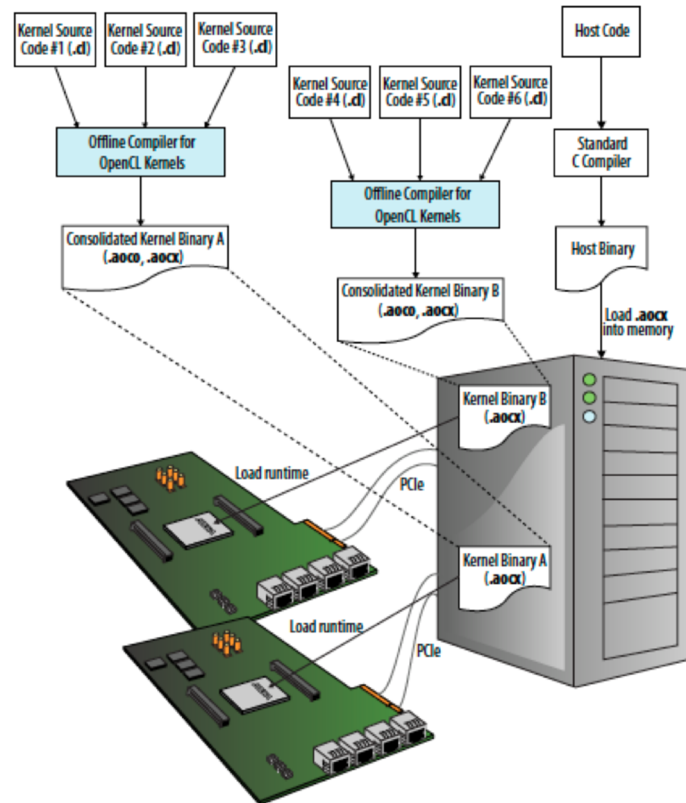


Figure 2.4: Compilation flow for OpenCL kernels (from Altera programming guide [3])

This synergistic combination of driver libraries, BSPs, emulator, and toolchain makes the design and verification cycle shorter for FPGAs. Developing an application in HDL is a time-consuming task, and integrating this design with different interfaces (PCIe, DRAM, ethernet etc.) incurs additional engineering overhead. The complete process of creating and verifying an application in HDL can take up to months of hard-skilled labor, depending upon the complexity of the application. The OpenCL compilers reduce this duration significantly. Using different BSPs allows the user to configure the same application on different boards, provided the boards have the required support packages. Porting an application to a different board may require further tuning and optimizations depending upon the hardware

constraints. However, the migration is fast and relatively easy. The Altera compiler also supports integrating designs written in HDL with OpenCL kernel. A kernel can use the HDL module as a function provided the HDL module follows the recommended guidelines, has appropriate interfaces, and argument bindings to be compiled as a functional library. We use this feature to report the HDL performance comparisons and it helped to reduce the integration time for HDL with PCIe and DRAM significantly.

Additionally, FPGAs have a different architecture from GPUs and are fundamentally different from them in parallelism extraction from an application. GPUs extract performance by exploiting data-level parallelism (DLP) and hiding memory latency in a highly threaded streaming multiprocessor. GPUs also have a high bandwidth DRAM (GDDR5). On the other hand, FPGAs have significantly lower DRAM bandwidth and rely on deep pipelining (i.e., ILP) to extract performance. Because of these fundamental differences between reconfigurable architectures and fixed architectures, Altera OpenCL compiler introduces a few constructs that are not available in standard OpenCL. Some of these constructs are of interest in this thesis and noted below:

- **Channel**: This acts as an interface between two concurrently running kernels. A channel is implemented as a queue or a first-in-first-out (FIFO) buffer in hardware. The user can guide the depth of channel by setting an attribute`depth` during declaration. However, the actual depth of the channel is decided by the compiler after static analysis of the kernel code. A *channel* is similar to the *pipe* construct in OpenCL [2]. Channels support non-blocking and blocking reads or writes. A blocking operation stalls the pipeline till it is successful; a non-blocking (NB) operation does not stall the pipeline and optionally provides the status of a NB operation indicating if it was successful.

- **Mem_fence**: The memory fence operation is a substitute of a *barrier* operation in

OpenCL. A barrier operation is meaningless for a task-based or single work-item kernel since it has only one work-item in the work-group of size one. However, it is required to ensure the ordering of data in a single work-item kernel, especially so if the data is provided to another kernel via memory or channels. For example, they are of vital importance in the implementation of the memory buffer residing in global memory, used by two or more task kernels. The user can ensure that all the global writes to memory are complete by using a mem_fence, before passing the control to another kernel that reads this memory buffer.

- **Autorun**: An autorun kernel starts on the device without any intervention from the host. A kernel is declared to be an autorun-kernel by setting an attribute. This kernel does not incur additional overhead required for launching it from the host. An autorun kernel can communicate with other kernels using channels.

To optimize the performance of a kernel, the user can utilize any remaining resources on the FPGA. It can be done by vectorization or data-parallelism, or by having multiple CUs replicated on the board. Altera compiler provided attributes for optimization can be used stand-alone or in conjugation with other options to improve the performance of a kernel. Some of the compiler optimizations investigated in this thesis are listed below:

- Multiple compute units can be created by using the attribute `num_compute_units`. It creates multiple CUs for an NDRange kernel.
- Kernel vectorization can help the performance by extracting the benefits of data-parallelism. By setting the attribute `num_simd_work_items` in an NDRange kernel. The user must ensure that the work-group size for this kernel is a multiple of the vectorization unit.

- Loop unrolling can extract the parallelism in a loop if there are no loop-carried dependencies. The user unrolls a loop by using a `pragma unroll` complemented with an unroll factor; in the absence of this factor, the loop is fully unrolled. Loop unrolling creates multiple copies of the loop pipeline. This optimization is used when the loop iterations can be determined statically.

- Kernel replication can be done for a single work-item kernel by setting the attribute `num_compute_units(x,y,z)`, where x, y, and z are dimensions along x-axis, y-axis, and z-axis respectively. Furthermore, the user can use this kernel grid to define communication among them by using the function `get_compute_id(<dim>)` in the kernel. However, this option can be used if an only if a kernel is an autorun kernel and it is declared by the user as a single work-item kernel using `max_work_dim(0)`. We use this feature to provide a scalable framework for profiling in Chapter 4.

- Static coalescing of memory access to DRAM can benefit performance by exploiting spatial locality. The user can do this by using the OpenCL vector data types or by diligent implementation in the kernel so that the compiler can extract the information statically.

2.3 Applications

We use applications from the OpenDwarf [64] suite, an architecture-agnostic OpenCL benchmark that captures basic computation and communication patterns across a broad spectrum of scientific and engineering applications, to study the performance of the OpenCL programming model on FPGAs. In OpenDwarfs, none of the benchmarks contains optimizations that favor a particular architecture over another. We discuss the details of the algorithms (Table 2.2) used in the study in following the sub-sections.

Table 2.2: OpenDwarfs benchmarks used

Dwarf	**Benchmark**	**Input data**
N-body Methods	GEM	nucleosome data-set
Structured Grid	3D Stencil	256^3 double floating-point grid

2.3.1 GEM

N-body algorithms are characterized by all-to-all computations within a set of particles. In GEM, an N-Body application, the electrostatic surface potential of a bio-molecule is calculated as the sum of charges contributed by all atoms in the bio-molecule due to their interaction with a specific surface vertex (two sets of bodies). The algorithm complexity is $O(N \times M)$, where N is the number of points along the surface, and M is the number of atoms. N-body is a compute-bound algorithm.

2.3.2 3D-stencil

3D-stencil from the OpenDwarf [64] suite solves partial differential equations in a three-dimensional (3D) grid by applying the 7-point finite-difference algorithm. In this method, each cell value is computed as an average of its six neighbors and the point itself in the X, Y, and Z directions, namely in north, south, east, west, top, and bottom as depicted in Figure 2.5. Similar to other "structured grid" algorithms, 3D-stencil is also memory-bound, low-computational intensity application computed on a three-dimensional grid, where each point on the grid can be updated independently. However, synchronization is required before proceeding to the next grid update step. Therefore, it requires a significant number of PEs and high memory bandwidth, thus making this problem suitable for GPU architectures.

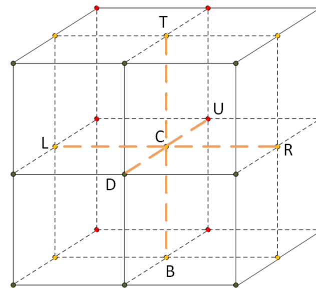


Figure 2.5: A cell of 3D-stencil grid that uses current value of its six-neighbors to compute the value of cell for next iteration.

```
stencil_basic(double *in, double *out,...)
plane = row*col
for k in 1 to height{
  for j in 1 to col{
    for i in 1 to row{
      ptr = pointer_to_curr_elem(k,j,i)
      left       = in[ptr-1];       right = in[ptr+1];
      center     = in[ptr];         down  = in[ptr+row];
      up         = in[ptr-row];     plane_up = in[ptr+plane];
      plane_down = in[ptr-plane];
      compute   = (left + right + up +  down + center + plane_up
                   + plane_down)/7.0f;
      if(boundary_element(k,j,i)) out[ptr] = in[ptr];
      else out[ptr] = compute;
}}}
```

Listing 2.1: Serial pseudo-code for three-dimensional seven-point stencil

Chapter 3

OpenCL Compiler Optimizations

For decades, the reconfigurability of FPGAs has allowed the user to create the deeply pipelined and highly parallelized datapath accelerators. This has delivered performance benefits across various application domains, such as option pricing solvers in finance, computational fluid dynamics in oil and gas, and packet processing in network routers and firewalls. However, this performance comes at the expense of programmability, i.e., the performance-programmability gap. FPGA programmers use hardware design language (HDL) to implement the application data path and to create hardware modules for computational pipelines, memory management, synchronization, and communication interfaces at the register transfer level (RTL). It means that the programmers must specify the cycle-accurate behavior for the data path in every module and register in the design [21]. This process is similar to traditional programming for CPUs in assembly language with the additional complexity of scheduling the instructions and data on a cycle-by-cycle basis, which requires extensive low-level knowledge of the target architecture and consumes significant development time and effort. In contrast, GPUs took the parallel computing community by storm in the late 2000s by significantly enhancing the programmability of GPUs via higher-level programming

abstractions for general-purpose computing, namely CUDA and OpenCL.

To address this lack of programmability of FPGAs, OpenCL provides an easy-to-use and portable programming model for CPUs, GPUs, APUs, and now, FPGAs [60]. However, this significantly improved programmability and portability can come at the expense of performance. Although the HLS compilers can generate functionally-correct hardware designs from the architecture-agnostic OpenCL kernels, it is unlikely that this synthesized hardware will utilize the FPGA resources efficiently to meet the required performance; that is, there remains a performance-programmability gap. We address the following aspects of the programmability-performance gap in this chapter:

1. We assess the performance gap between fixed and reconfigurable architectures by characterizing the performance of a small subset of benchmarks in the OpenDwarfs benchmark suite on multi-core CPUs, GPUs, Intel MIC, and FPGAs. The results indicate that the architecture-agnostic OpenCL kernels result in inefficient hardware designs on FPGAs *(Section 3.1)*.

2. To improve the performance of OpenCL kernels on FPGAs, and thus, bridge the performance-programmability gap, we identify general techniques to optimize OpenCL kernels for FPGAs under device-specific hardware constraints. We then apply these optimization techniques to two example cases from the OpenDwarfs benchmark suite to evaluate their effectiveness regarding performance and resource utilization, and present performance of the optimized implementations *(Section 3.2)*.

3. We present a realization of an efficient algorithm for a three-dimensional stencil using OpenCL for FPGA, and an evaluation of its performance *(Section 3.3)*.

3.1 Performance Characterization

In this chapter, our primary goal is to study the performance of the OpenCL programming model for FPGAs using the OpenDwarfs benchmark suite. First, We assess this performance gap between fixed and reconfigurable architectures by characterizing the performance of two applications from the OpenDwarfs benchmark suite on multi-core CPUs, GPUs, Intel MIC, and FPGA. Second, we provide the detailed analysis of the compiler and manual optimizations that can be applied on the OpenCL kernels to bridge this performance gap.

Table 2.2 (in Chapter 2) presents the OpenDwarfs subset considered in this study, i.e., structured grid (3D-stencil) and N-body (GEM), and their input data-sets and/or parameters. Table 3.1 lists the target fixed architectures for evaluation of these applications and Section 2.1.2 provides the detail about the FPGA.

Table 3.1: Specification of test architectures

Model	**Intel i5-2400**	**Intel Xeon E5-2700**	**Intel MIC 7100**	**Tesla C2070**	**Tesla K20X**
Type	CPU	CPU	Co-proc.	GPU	GPU
Freq. (GHz)	3.1	2.7	1.238	1.15	0.732
Cores	4	12	61	14	14
SIMD (SP)	8	8	16	32	192
GFLOPS (SP)	198.4	518.4	2415.6	1030	3950
On-Chip mem.	7.125	33.375	32.406	3.375	3.032
B/W (GB/s)	21	59.7	352	148.42	250
Process (nm)	32	22	22	40	28
TDP (W)	95	130	270	238	235

3.1.1 Stencil

An efficient GPU version of the 3D-stencil algorithm divides the problem into smaller three-dimensional blocks, where each element of the smaller blocks is a computed by a separate

thread. Each thread in a thread-block fetches an element to operate upon and waits for other threads to reach a barrier. All the elements required to compute the stencil for a smaller block are available in the shared memory when each of the thread from a thread-block has reached this synchronization point. Then, these threads proceed to calculate the function and wait at a barrier. Once each element on the plane is computed, threads proceed to compute the next level.

Figure 3.1 shows the performance of the 3D-Stencil across fixed and reconfigurable architectures. Tesla K20XM and Tesla C2070 GPUs outperform other devices for this implementation owing to their efficient intra-block synchronization and utilization of (fast) shared memory, while former doing better than the latter due to higher memory bandwidth. Intel MIC 7100 performs better than CPUs(Intel Xeon E5-2700 and Intel i5- 2400) because of the higher number of cores and memory bandwidth. FPGAs perform poorest due to the substantial synchronization overhead.

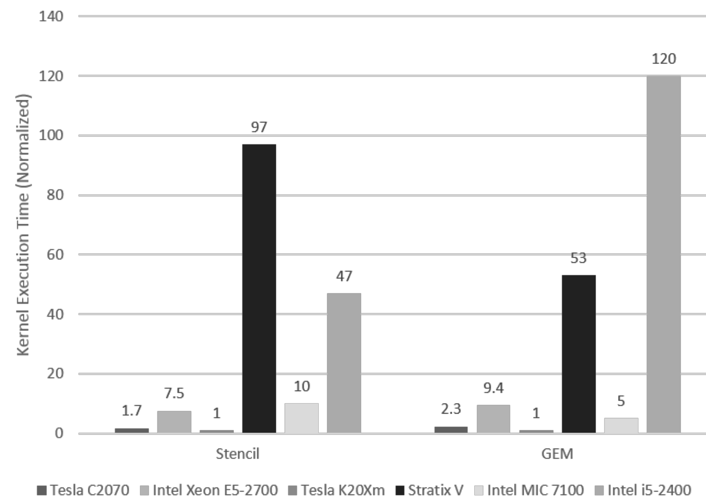


Figure 3.1: Architecture-agnostic kernel performance for Stencil and GEM normalized to Tesla-K20Xm (lower is better)

3.1.2 GEM

GEM is a regular compute-bound N-body algorithm that calculates the electrostatic surface potential of biomolecules in a set of particles. The computation of surface potential for a biomolecule requires atoms' data from all of its neighbors and this data can be reused during the computation of any other biomolecule in this set. To take the benefit of data reuse, the computation for a biomolecule is launched as a thread in a thread-block, where each thread independently accumulates the potential at a single point due to its neighbors. This computation requires a significant number of processing elements (PEs) and it is sensitive to data locality. Hence, GPUs with their massive number of PEs achieve the best performance (Figure 3.1).

Architecture-agnostic implementation of the OpenCL kernels results in inefficient hardware designs on FPGAs. Thus, we detail optimization strategies to improve their performance on the FPGA in Section 3.2.

3.2 FPGA Optimizations and Insights

To improve the performance of OpenCL kernels on FPGAs, we exploit different levels of parallelism — task, data (i.e., SIMD vectorization), and instruction-level parallelism (ILP) — and minimize memory access latency by controlling data movement across the memory hierarchy levels and coalescing memory accesses. Because FPGAs have limited hardware resources and memory bandwidth, it is imperative that we analyze different combinations of these optimization techniques to identify the best set and generate the most efficient (regarding performance and resource utilization) hardware design for the application under consideration. We explore the FPGA-oriented optimization space and present insights that

may be leveraged in the future OpenCL compilers targeting FPGAs.

3.2.1 3D-stencil Benchmark

To improve the poor performance of the NDRange kernel on FPGA, we employ different optimizations techniques. An abridged version of these optimization results is following: rewriting the kernel as a single work-item improved performance. Static memory coalescing and algorithmic refactoring further increased the performance of the single work-item kernel. Conversely, designs with multiple compute units, compiler vectorization, or data sharing among work-items using intra-kernel channels resulted in little or no performance improvement. The following subsections discuss the insights and details of these optimizations.

Single Work-Item Kernel

The pipeline parallelism for FPGAs works best when the pipeline is full and processes one work-item per clock cycle. In theory, it can produce one result per clock cycle in the absence of pipeline bubbles or stalls. Programming a kernel as a single work-item enables the compiler to attempt creating an efficient and stall-free pipeline. Any stalls in the pipeline reduce the throughput and it can occur for various reasons. For example, if the number of memory read accesses per clock cycle exceed the number of available read ports in the local memory, then the compiler inserts an arbiter to provide more ports. This extra hardware, in turn, increases access latency and results in pipeline stalls and, overall, an inefficient implementation. Barriers in a kernel can also lead to pipeline stalls. The stencil NDRange kernel (as originally designed for the GPU) results in sub-par performance when compiled and synthesized for the FPGA (Stratix V in Figure 3.1, GPU-O (GPU-optimized kernel) in Figure 3.2).

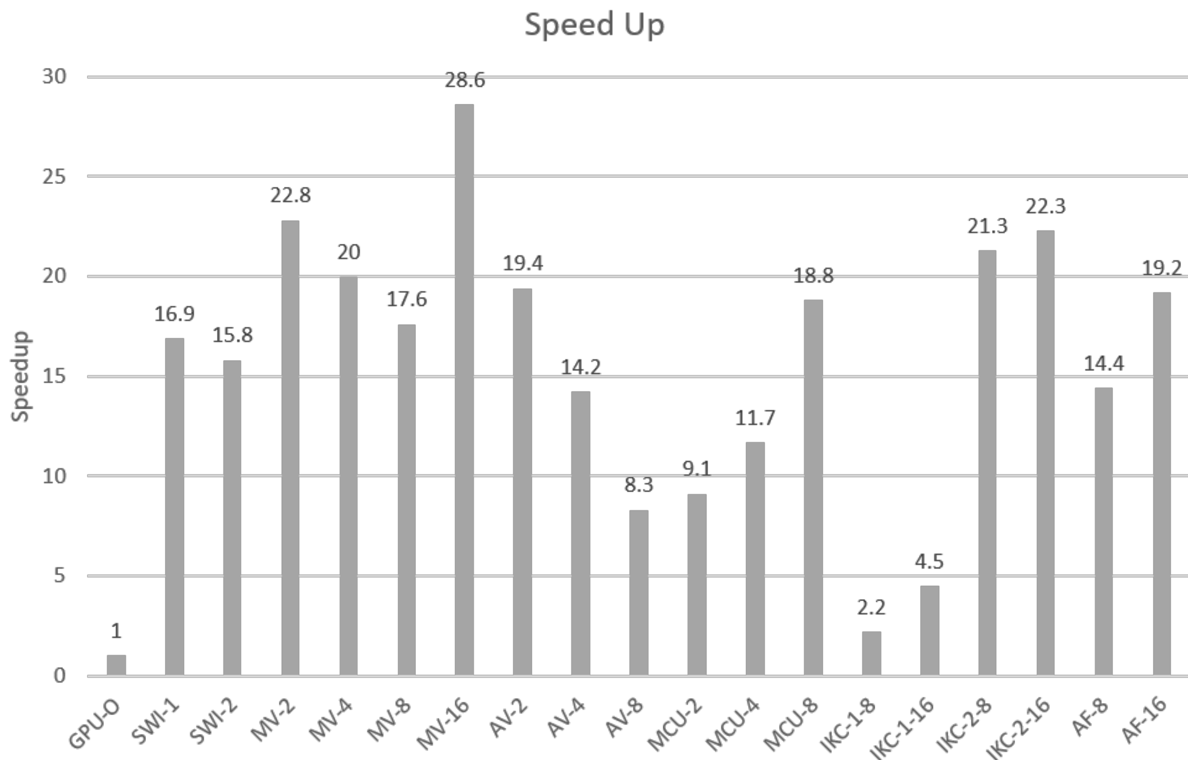


Figure 3.2: Optimized 3D-stencil kernel implementations (higher is better)

To benefit from pipeline parallelism on the FPGA, we implement the kernel as a single work-item kernel (SWI-1, Listing 3.1). SWI-1 results in an improvement of approximately 16x (SWI-1 in Figure 3.2) over the unoptimized kernel originally designed for the GPU for a grid size of $(256)^3$ double-precision floating point elements. This kernel is launched on the FPGA with a work-group size of one and the dimensions of the grid are provided using the kernel arguments.

A similar version of the same kernel is RWG-1 (Listing 3.2), that has a work-group size of one but uses a dimension from the work-group size. RWG-1 results in a larger footprint on the FPGA (has additional 5% ALM usage) and performs slightly better than SWI-1. Furthermore, it results in better memory bandwidth utilization because of the larger burst size and efficient coalescing, as indicated by bar RWG-1 in Figure 3.3. Both kernels

```
__attribute__((reqd_work_group_size(1,1,1)))
kernel_stencil_db(dimension, in,out) {
    for(i=0; i < dimension; i++) {
        fetch_neighbors();
        do_stencil();
    }
}
```

Listing 3.1: Single work-item (SWI-1)

```
__attribute__((reqd_work_group_size(1,1,1)))
kernel_stencil_db(in,out) {
    int id = get_group_id(0);
    fetch_neighbors(id);
    do_stencil();
}
```

Listing 3.2: Work-group size one (RWG-1)

perform the same operations, so this difference in the resource utilization could potentially be attributed to compiler optimizations done for memory access coalescing. We also noted during our experiments that SWI-1 is a better style of writing kernel for FPGA because of the benefits of using the shared memory and registers across work-items without introducing a barrier or stalls in the pipeline.

Insight: *Single work-group size kernels are suitable for FPGA.*

Manual Vectorization or Static Coalescing

FPGAs have limited memory bandwidth but high computational capability. Efficient memory accesses can potentially accelerate the application and this applies to 3D-stencil as well. To improve the performance of RWG-1, we use manual vectorization with SIMD width of two, four, eight, and sixteen. Figure 3.2 shows the corresponding performance in bars MV-2, MV-4, MV-8, and MV-16, respectively. MV-16 results in the highest memory bandwidth

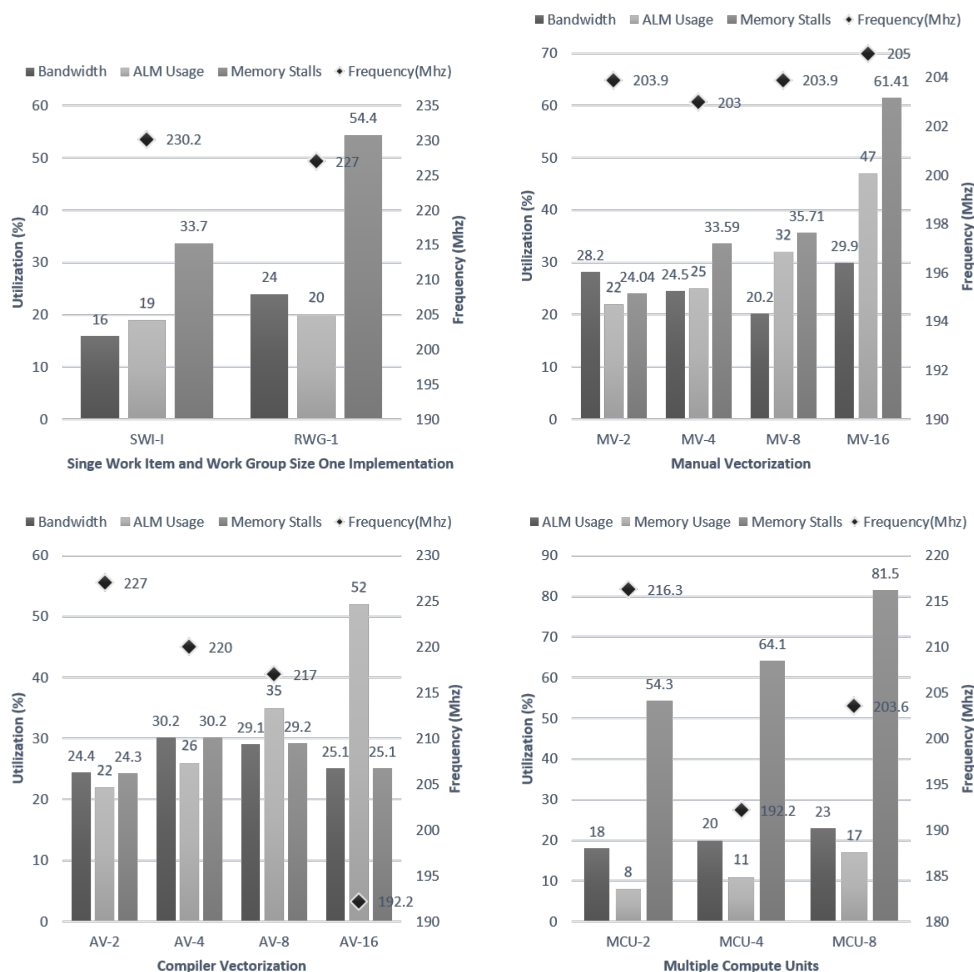


Figure 3.3: Single work-item kernel, manual and compiler vectorization, multiple compute units: profiling (bandwidth, ALM usage, memory stalls, and clock frequency)

and, accordingly, in the best performance with a speed up of 29x over the unoptimized kernel. This improved performance comes at the cost of area usage (Figure 3.3). We did not observe an increase in memory usage by applying manual vectorization owing to the simplicity of the stencil benchmark's kernel. Increasing the SIMD length results in higher number of pipeline stalls, but increased memory bandwidth utilization improves the performance.

Insight: *Manual vectorization may result in efficient memory coalescing and can potentially augment the performance.*

Compiler Vectorization

An OpenCL directive hints the Altera OpenCL compiler to attempt automatic code vectorization. This attribute requires a work-group size that is a multiple of vectorization length. We choose the RWG-1 implementation to evaluate compiler-generated vectorization. Bars AV-2, AV-4, AV-8, and AV-16 in Figure 3.3 correspond to the performance for vectorization width of two, four, eight, and sixteen, respectively. As 3D-stencil is a memory-bound application, it does not benefit from compiler vectorization. Specifically, although burst size increases because of memory access coalescing, a high percentage of stalls results in low bandwidth utilization. Performance deteriorates for RWG-1 with an increase in the vectorization width and results in lower clock frequency and higher resource utilization (ALM usage) (Figure 3.3).

Insight: *Compiler vectorization may not lead to an improvement of performance for memory bound algorithms.*

Multiple Compute Units

Replicating the pipeline of single work-item kernels may result in improving the performance. However, since stencil implementation RWG-1 is memory-bound, we do not expect, nor obtain, any performance gain on RWG-1 by this optimization, as shown by bars MCU-2, MCU-4, and MCU-8 in Figure 3.2, which correspond to two, four, and eight compute units, respectively. However, it is worth observing that the area usage, clock frequency, and memory stalls percentage (Figure 3.3) grow linearly as the number of compute units increase. The clock frequency remains roughly the same for all cases of the CUs.

Insight: *Multiple compute units optimization may not enhance performance for memory bound algorithms.*

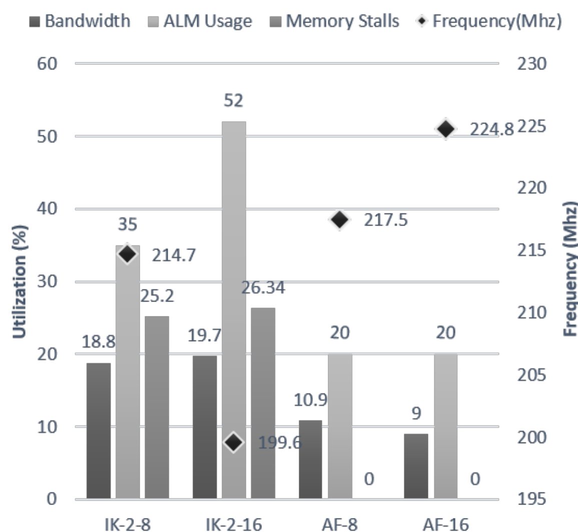


Figure 3.4: Data sharing in X direction, and algorithmic refactoring: profiling

Intra-Kernel Channels

Altera introduces *channels*, a custom solution that functions as the equivalent to OpenCL-2.0 [2] pipes. Channels are communication links that can be used to transfer data among kernels. It eliminates the need for memory transfers between the host and device and the corresponding data transfer overhead. To enhance the data locality for RWG-1, we perform computation along the X direction in blocks of 8 and 16 double words and reuse the data in the X direction. We use the channel to transfer data among the work-items in the same kernels. The above optimization results in a poor design and instead of improvement, we notice a slowdown compared to RWG-1. This performance loss can be attributed to the channel's latency (IKC-1-8, IKC-1-16 in Figure 3.2), which is stalled 98.7% of the time for both of the implementations. The above optimization applied for SWI-1 implementation using private registers results in performance improvement (IKC-2-8 and IKC-2-16 in Figure 3.2) over SWI-1. This difference in performance can be likely attributed to channel-related latency considerations in the pipeline implementation.

Insight: *Intra-kernel channels may degrade the performance without proper latency considerations. However, increasing data locality for single work-items improves performance.*

Algorithmic Refactoring

To improve data locality in Stencil, we implement an algorithm similar to the GPU implementation. We divide the problem into smaller three-dimensional blocks, where each block has the same height, but smaller size in the X and Y dimensions. The elements are fetched into an on-board cyclic buffer of size $(2*(x+2)*(y+2)+1)$ for computation, where x and y are the size of the smaller blocks in X and Y dimensions. An additional 2 elements (added to x) is needed to account for the boundary elements in a smaller block. This implementation is discussed in more detail in Section 3.3, where we focus on designing an accelerator for a fixed plane size 3D-stencil problem. Implementation for an 8x8 plane size did not result in better performance (AF-8 in Figure 3.2) because of poor memory coalescing. However, a 16x16 plane size (AF-16 in Figure 3.2) enables performance gains due to higher data-reuse. Moreover, AF-8 and AF-16 result in a smaller footprint on the FPGA (Figure 3.4) compared to SWI-1 and no memory stalls.

Insight: *A kernel that takes FPGA memory hierarchy and pipeline parallelism in design choices is most likely to exploit the benefits of reconfigurable computing.*

3.2.2 GEM Benchmark

Use of Restrict/Const Keywords and Kernel Vectorization

An optimization[1] strongly suggested in the Altera manual is using the *restrict* keyword for kernel arguments that do not use pointer-aliasing (i.e., point to the same memory location). Using *restrict* attribute in kernel allows the compiler to create more efficient designs for FPGA that result in improved performance. It does so by eliminating unnecessarily assumed memory dependencies. Although a side effect of such an optimization could be lower resource utilization, we find that this is not the case in our application. Cases IMP2 and IMP4 (Figure 3.5) highlight the difference (1.31 times higher utilization with *restrict*) across two otherwise identical implementations. Performance-wise, IMP4 is 3.94 times faster and this stems from the vast majority of memory accesses resulting in cache hits. Conversely, IMP2 has sub-optimal memory accesses that result in cache misses and pipeline stalls (about 80% of the time).

As far as `const` keyword is concerned, we do not observe any difference in resource utilization, or in execution time. Automatic kernel vectorization (SIMD), which is enabled with the appropriate OpenCL attribute, can yield performance gains of 5.85x (IMP4, IMP8) at the cost of increased (double) resource utilization.

Insight: *Using* ***`restrict`*** *keyword may result in higher performance but not necessarily lower footprint on FPGA.*

[1] Experiments for optimization on GEM were conducted by Konstantinos Krommydas and added here for completeness of the thesis.

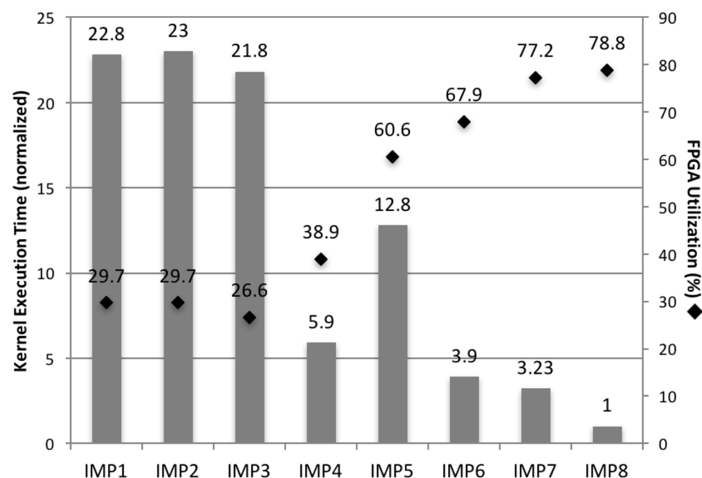


Figure 3.5: Optimized GEM kernel implementations

Compiler Resource-Driven Optimizations

When a kernel is compiled with the resource-driven optimizations, the compiler applies a set of heuristics and estimates resource utilization and throughput for multiple kernel attributes, like loop unroll factor, kernel vectorization, number of the CUs. This process should not always be expected to provide the best implementation. In our example application, we identify at least one case where a manual choice of kernel vectorization width surpasses (by 3.33x) the compiler selected attributes (*pragma unroll 4*) (IMP6, IMP5 in Figure 3.5). Profiling the kernel, we find that IMP6 benefits from coalesced memory accesses, while memory accesses in IMP5 result in costly pipeline stalls. Also, bandwidth efficiency is higher (more than double) in IMP6 (i.e., more of the data acquired from the global memory system is used by the kernel). Altera discusses the inherent limitations of static resource-driven optimizations in their optimization guide [3]. Developers should consider the limitations as mentioned above when compiling using the resource-driven optimization option.

Insight: *Manual search of optimization space can outperform automatic compiler resource-driven optimizations.*

Algorithmic Refactoring

A given algorithm implementation may solve an actual problem, but this does not mean that a set implementation is appropriate for every platform (e.g., CPU, GPU, FPGA). A different implementation for solving the same problem, i.e., produce the same output given the same input, may be necessary. While this may not be intuitive, or even applicable for all cases, certain algorithmic restructuring can prove very beneficial. To illustrate the above, we apply basic algorithmic refactoring in our example application. Specifically, we remove the complex conditional statements for different cases encapsulated in a single kernel and tailor the kernel to the problem at hand. It provides a two-fold benefit: (a) better resource utilization (in our examples the re-factored algorithm requires about 10% fewer FPGA resources, and (b) better performance (5% faster, IMP2, IMP3). What is more important, though, is that better resource utilization may allow wider SIMD or more compute units to fit on a given board. In our example (IMP6, IMP7 in Figure 3.5), the reduced resource utilization of the re-factored algorithm allows SIMD length of 16, whereas the original one accommodated up to 8 (logical elements being the limiting factor). It translates to a 1.22x faster execution of the former compared to the latter.

Insight: *Our experiments with GEM refactoring reiterates our experiments with Stencil, kernel designed with FPGA architecture in consideration has the potential to exploit reconfigurable architecture*

3.3 Fixed-plane-size Accelerator for 3D-stencil

This section discusses the design of an FPGA accelerator for 3D-stencil computation using the OpenCL language for a 32x32 double floating-point plane size and variable height. This

Implem.	**Refact.**	**Restrict**	**Constant**	**SIMD**	**CU**	**Unroll**
IMP1				1	1	1
IMP2			✓	1	1	1
IMP3	✓		✓	1	1	1
IMP4		✓	✓	1	1	1
IMP5			✓	1	1	4
IMP6			✓	8	1	1
IMP7	✓		✓	16	1	1
IMP8		✓	✓	8	1	1

Table 3.2: GEM kernel implementation's features

accelerator is similar in architecture to the streaming element for stencil as discussed by Sano et al. [35], but it is designed in the OpenCL and restricted to a single board. The data-load machine in the accelerator fetches cells from DRAM and writes it into a cyclic buffer that has a size of (32x32x2+1) double words. Once the cells from first two planes are present in this cyclic buffer, computation can proceed concurrently as the data-load machine continues to fetch cells. This results in the full utilization of available memory bandwidth because of enhanced data reuse and improved spatial locality. This implementation generates an efficient pipeline and in theory can perform one computation per data fetch, once the pipeline is full. In following subsections, we discuss re-factored algorithm, followed by optimizations used to design it in the OpenCL, and an evaluation of hardware generated by Altera compiler.

3.3.1 Re-factored Algorithm

The accelerator for 3D-stencil is designed as a single work-item kernel and its pseudo-code is provided in Listing 3.4. The outermost loop (variable i) iterates over the planes, the inner loop (variable j) iterates over rows, and the innermost loop (variable k) processes elements in the column. Sustaining highest throughput, i.e., computation of a grid-point in single cycle requires six read requests to the local memory, but the available memory on-board has only

```
int sh_register[size];
#pragma unroll
for(int i = 0; i< size; i++)
    sh_register[i] = 0;

for(int k = 0; k < loop_count; k++) {
    #pragma unroll
    for(int i=0; i < (size - 1); i++)
        sh_register[i] = sh_register[i+1];

    sh_register[size-1] = input_data ;
}
```

Listing 3.3: Shift register implementation in OpenCL

two ports. To handle this problem, the cyclic buffer is divided into multiple banks acting together as a large buffer. This large buffer consists of 65 smaller buffers, each of size 256 bytes. This memory partition avoids the need of an arbiter on the large buffer, which would result in a higher memory access latency. The benefit of this partition is evident in minimal memory stalls when the innermost loop is unrolled to a factor of four in Figure 3.6.

3.3.2 Cyclic Buffer Implementation

This implementation employs shift register inference optimization, as explained in the OpenCL optimization guide [3]. Listing 3.3 is an implementation of shift registers in OpenCL using compile pragma (*#pragma unroll*). A shift register is a powerful tool and it is used extensively in the hardware design. An accelerated implementation for dynamic programming [65] uses shift registers to resolve complex dependencies among elements in the Smith-Waterman algorithm for sequence matching. A small shift register can be implemented in the register space (ALMs) while the larger ones can be implemented using block-RAMs (BRAM) on the FPGA.

```
#define N 32
#define SIZE (2*N*N +1 )
__attribute__((reqd_work_group_size(1,1,1)))
__kernel kernel_stencil(in_d,out_d,dim_z)
    __private double sh_mem[SIZE];
    for(int i = 0; i < dim_z; i++)
        for(int j = 0; j< N; j++) {
            #pragma unroll <UNROLL_FACTOR>
            for(int k = 0; k< N; k++) {
            __private double neighbors[6];
            __private double in;
            in = in_data[index + k];
            get_neighbors_from_buffer();
            out_d[index+k-N*N] =
                compute_stencil(neighbors,in);
            sh_mem[size -1] = in;
            shift_registers;
        }
     }
```

Listing 3.4: Pseudo-code for 3D-stencil implementation

3.3.3 Loop Unrolling

The FPGA implementation of the above algorithm without any loop unrolling suffers from low memory bandwidth utilization. Unrolling the innermost loop (variable k) results in better utilization of memory bandwidth, as more compute units are generated. A Higher number of compute units, in turn, increase the number of buffer reads per clock cycle and results in diminishing bandwidth gains. An unroll factor of eight results in a high number of memory stalls, however, performance still increases due to increased memory bandwidth. The increase in bandwidth utilization does not scale with unrolling.

Insight: *The loop-unroll can help improve utilization of under-utilized memory bandwidth if unrolling does not increase local memory access latency, after which there are diminishing returns.*

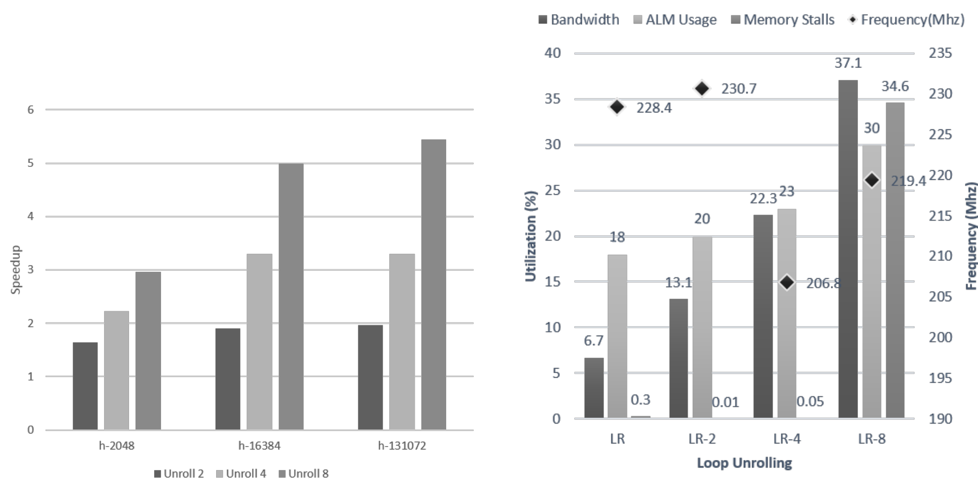


Figure 3.6: Speedup and profiling information for unroll

3.3.4 Evaluation

The implementation of the 3D-stencil algorithm without any unrolling achieves the near-maximum possible throughput for a single pipeline. A pipeline that computes one point of the grid in each clock cycle and runs at a clock frequency of 228.4 Mhz requires memory bandwidth of $228.4 * 8 = 1.872$ Gbps. The presented accelerator utilizes 6.678% of memory bandwidth or $1.709 GBps$, translating to an efficiency of 91.32% of theoretical throughput. This design performs 3D-stencil computation for one point per cycle, and each calculation does six floating-point operations turning into a performance of 1.25 GFlops. This floating-point performance increases to 7 GFlops with an unroll factor of eight. Figure 3.6 shows the speed-up attained using unrolling for heights of 2048, 16384, and 131072 elements. This speed-up does not scale linearly with unrolling because a larger unrolling factor results in the increased stalls introduced by memory banks. The accelerator discussed here is a preliminary work with shift registers and a more detailed approach is presented in Chapter 5 that uses advanced memory optimization.

3.4 Aligned Data Transfers: Host-side Optimization

The target FPGA platform (Stratix V) supports a maximum DRAM data transfer bandwidth of 25 Gbps with a burst length of sixteen and word size of 64-bits. The Altera manual [3] recommends using 64-byte boundary aligned buffers in the host memory to accelerate the data transfer time from host to device (H2D) and device to host (D2H) by utilizing direct memory access (DMA) engines. The time to move aligned data in either direction (H2D or D2H) is the same when using the DMA. On data size, the speeds differ by a factor of six for $8 * (64)^3$ bytes of unaligned data and the difference increases with data size (Figure 3.7). The speedup in using aligned access reduces as data size grows. Aligned data transfers are the only host-side code optimization that does not require any algorithmic changes.

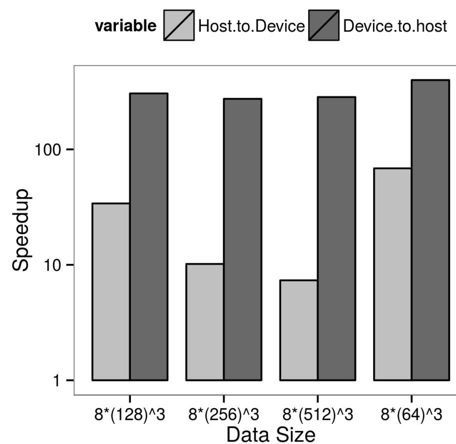


Figure 3.7: Data transfer time

.

Chapter 4

A Scalable Framework for Profiling

In this chapter, we introduce two fundamental patterns, one for timestamps and the other for sequence numbers, to gain information on hardware execution. We then propose an intelligent trace buffer, referred to as iBuffer, to collect and process the run-time information. Figure 4.1 illustrates the generic iBuffer framework. As shown in the figure, an iBuffer contains both logic function blocks and a trace buffer. The logic function blocks provide data processing capabilities while the trace buffer serves as a flight recorder, storing the information from the data input channel(s). The command channel configures the state of the trace buffer and how the data are to be processed. The data output channel is used to forward the data in the iBuffer to the user for analysis.

We examine two use cases to show how our proposed iBuffer framework [66] can be utilized. First, we use it to collect latencies and analyze pipeline stalls to reason about the performance. Second, we construct watch points for selected memory locations to realize the watch capability in conventional software debugging tools such as gdb. In the meanwhile, we also perform run-time address bound checking and value invariance checking for memory operations. Our experiments show that our proposed scheme works well in these use cases

and incurs limited overhead in the FPGA area and clock frequency.

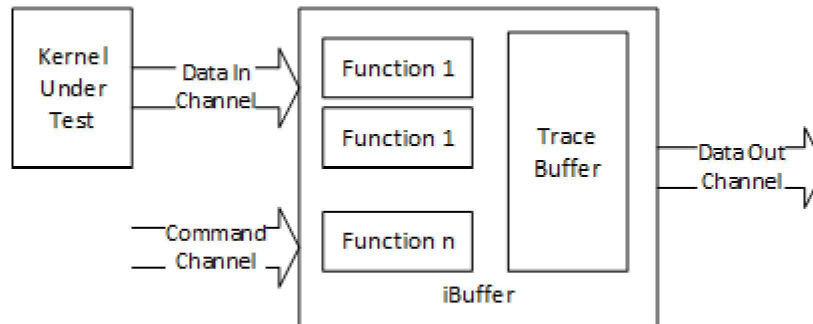


Figure 4.1: An overview of the iBuffer structure.

4.1 Primitive Patterns

4.1.1 Timestamp

Timestamps are useful to reveal run-time information on hardware execution. To enable the timestamp functionality, we propose the following two schemes. The first is to use a persistent autorun kernel, which contains a free-running counter and feed the counter value through an Altera channel with depth zero, meaning that the channel always contains the most up-to-date counter value. The code of this persistent kernel is shown in Listing 4.1. As shown in the code, the counter value is written to the channel in a non-blocking manner, which will not affect the logic to increment the counter each cycle. For retrieving the timestamp at a particular instance, a read channel function call is inserted into the kernel under test, as illustrated in Listing 4.2. In this example, there are two read sites of the timestamp and the difference between the two timestamps would show the latency of the event of interest, a vector dot-product in this case. Note, that since each channel can only support one producer and one consumer, multiple channels are used for multiple read sites.

```
channel int time_ch1 __attribute__((depth(0)));
__attribute__((autorun))
__attribute__((max_global_work_dim(0)))
kernel void timer_srv(void) {
  int count = 0;
  while(1) {
    bool success;
    count++;
    success = write_channel_nb_altera(time_ch1,
        count);   } }
```

Listing 4.1: The timestamp pattern using a persistent autorun kernel with a free-running counter.

```
int start_t, end_t;
//Read site 1
start_t = read_channel_altera(time_ch1);
//Event of interest
sum = 0;
for(int i = 0; i < num; i++) {
  sum += x[i+l]*y[i]; }
z[k] = sum;
//Read site 2
end_t = read_channel_altera(time_ch2);
```

Listing 4.2: Read site(s) of the timestamp.

The potential limitation of the persistent kernel based timestamp is two folds. First, the OpenCL compiler may try to optimize the channel depth although it is explicitly set to zero, which may result in stale timestamps. In our experiments, we found that we have to use one persistent kernel to drive one channel rather than letting multiple channels driven by the same free-running counter in a single persistent kernel. This may be a problem if different persistent kernels are not launched in the same cycle and there could be offsets among the separate free-running counters. Second, the read sites of timestamps have no data dependency upon any variables in the computation of interest. Therefore, the OpenCL compiler might move the read sites of the timestamps to optimize pipeline schedules, although

such movement, i.e., moving the `read_channel` functions, has not been observed in our experiments.

Our second approach for timestamps aims to overcome the aforementioned limitations. Instead of using a persistent kernel containing a free-running counter, we resort to hardware-design language (HDL) to initiate such a counter and embed it into the kernels under test. Listing 4.3 and 4.4 shows the code of this approach and how it is used. As shown in Listing 4.3 , we define an OpenCL function `get_time`. The function defined in OpenCL has one input parameter, *command*, and simply returns the sum of (*command*+1) as its output. Such an OpenCL definition is used for emulation while the actual implementation for synthesis is defined in a Verilog module. All such information is encapsulated in a library to be integrated during the OpenCL compilation.

```
In the file "timer.h"
ulong get_time(ulong command);
In the file "timer.cl"
ulong get_time(ulong command) {
    return (command + 1); }
In the file "timer.v"
module get_time (input clk, ..)
always @(posedge clk)
  if(~rstn) counter_time <= 'h0;
  else counter_time <= counter_time + 1;
...
```

Listing 4.3: The timestamp pattern using a verilog module containing a free-running counter and its OpenCL interface.

Listing 4.4 shows the read sites of the timestamp in the same kernel as in Listing 4.2. The variable *sum* is passed as the input parameter to our defined `get_time` function. The reason is to create dependency so as to avoid the compiler accidentally moving the read sites during scheduling.

We analyze the area and frequency overhead of the two approaches to implement the times-

```
int start_t, end_t;
start_t = get_time(sum); //read site1
//event of interest
sum = 0;
for(int i = 0; i < num; i++) {
  sum += x[i+l]*y[i]; }
z[k] = sum;
end_t = get_time(sum); //read site2
```

Listing 4.4: The read sites(s) of the timestamp using HDL counters.

tamp pattern. The synthesis reports show that both methods have a small area and frequency overhead. Between the two approaches, the HDL implementation has lower cost in register usage and logic unit (1.1% logic overhead including a trace buffer) than the persistent kernel approach (1.3% logic overhead with the same trace buffer). The frequency of an un-profiled kernel, which performs intensive pointer-chasing operations, reaches 233.3MHz while the one adding the OpenCL free-running counters runs at 227.8MHz, and the one including the HDL counter runs at 229.2 MHz. As it does not use the channel, thereby free from the channel depth issue, the HDL approach is preferred to implement the timestamp pattern.

4.1.2 Sequence Number

Besides timestamps, a sequence number[1] can be used to establish the order of run-time events. A sequence number can also be constructed using a persistent kernel, as shown in Listing 4.5. Similar to the code in Listing 4.1, a persistent autorun kernel is used to maintain the sequencing counter as shown in Listing 4.5. Rather than a free-running counter for timestamps in Listing 4.1, the sequencing counter will not be incremented until the blocking channel write function is finished. In other words, only after the consumer reads out the counter value from the channel, the counter is incremented.

[1]Sequence number logic was proposed and designed by Huiyang Zhou and added here for completion of this thesis.

```
#pragma OPENCL_EXTENSION   cl_altera_channels:enable
channel int seq_ch __attribute__((depth(0)));

__attribute__((max_global_work_dim((0)))
__attribute__((autorun))
kernel void seq_srv(void) {
  int count = 0;
  while(1) {
    count++;
    write_channel_altera(seq_ch,count);
  }  }
```

Listing 4.5: Sequencing number: The persistent kernel containing a sequence counter.

Listing 4.6 and Listing 4.7 show how we use the sequence number to reveal the execution/scheduling order of the loops/work-items in a matrix-vector-multiplication kernel. The Altera OpenCL for FPGA compiler supports both single-task and NDRange kernels. In the single-task mode, the matrix-vector-multiplication is implemented as a nested loop as shown in Listing 4.6. In the NDRange mode, each work-item computes one dot-product, as shown in Listing 4.7. The compiler extracts the loop-level parallelism in the single-task kernel while it relies on the explicit thread-level parallelism in the NDRange kernel. Here, we use the sequence number to reveal how the synthesized hardware executes/schedules such loops/work-items. The sequence number here is used as addresses for the profiling buffers. We also use the timestamp pattern to collect the clock cycle information, from which the execution order can also be constructed/confirmed. Since there is one read site and the data dependency due to the variable *seq* prevents the compiler from moving the `read_channel function`, we use the OpenCL free-running counter for the timestamps. A segment of the collected results is shown in Figure 4.2, Figure 4.2(a) for the single task kernel in Listing 4.6 and Figure 4.2(b) for the NDRange kernel in Listing 4.7.

From Figure 4.2, we can see that the single-task kernel in Listing 4.6 and the NDRange kernel in Listing 4.7 result in different execution orders. For the single-task kernel, all iterations in

```
for(int k = 0; k < N; k++) { //N=50
  l = k*num; sum = 0;   //num = 100
  for(int i = 0; i < num; i++) {
    sum += x[i+l]*y[i];
    if (i < 10) {
      int seq = read_channel_altera(seq_ch);
      info1[seq] =
        read_channel_altera(time_ch[0]);
      info2[seq]  = k;
      info3[seq]  = i;
    }  }
  z[k] = sum; }
```

Listing 4.6: Sequencing number: The read site of the sequence number in a single-task kernel.

```
k = get_global_id(0); //50 work-items
l = k*num; sum = 0;   //num=100
for(int i = 0; i < num; i++) {
  sum += x[i+l]*y[i];
  if (i < 10) {
    int seq = read_channel_altera(seq_ch);
    info1[seq] =
      read_channel_altera(time_ch[0]);
    info2[seq]  = k;
    info3[seq]  = i;
  } }
```

Listing 4.7: Sequencing number: The read site of the sequence counter in an NDRange kernel.

the inner loop are executed first before going to the next iteration of the outer loop, the same as sequential execution. For the NDRange kernel, however, different work-items (equivalent to outer loop iterations) get into the pipeline before they go to the next iteration of the (inner) loop. Such different execution orders lead to different memory access patterns. In the case of the single task kernel in Listing 4.6, the memory access order of the array x is x[0], x[1], x[2], . . . , for the first iteration of the outer $for(k)$ loop and x[100], x[101], x[102], . . . , for the second iteration, etc. In contrast, the access order for array x in the NDRange kernel in Listing 4.7 becomes x[0], x[100], x[200], . . . , for the first iteration of the $for(i)$ loop, and

then x[1], x[101], x[201], . . . , for the second iteration. Such different memory access patterns contribute to the different execution times (as revealed in the timestamps in Figure 4.2) of the two kernels.

(a)

```
               Timestamp    k    i
info_seq[51]:  260911       5    0
info_seq[52]:  261002       5    1
info_seq[53]:  261003       5    2
info_seq[54]:  261004       5    3
info_seq[55]:  261063       5    4
info_seq[56]:  261066       5    5
```

(b)

```
               Timestamp    k     i
info_seq[51]:  289634       0     1
info_seq[52]:  289635       1     1
info_seq[53]:  289636       2     1
info_seq[54]:  289637       3     1
info_seq[55]:  289638       4     1
info_seq[56]:  289639       5     1
```

Figure 4.2: The execution/scheduling order of the loop iterations or work-items, (a) for Listing 4.6 and (b) for Listing 4.7.

4.2 Framework

As shown in Figure 4.1, our proposed iBuffer contains logic functions and a trace buffer. The logic functions control how the incoming data are processed while the trace buffer stores the pertinent information for debugging and profiling. The command channel configures the state of the iBuffer to process the incoming data on a data channel. The output channel is used to forward the data stored in the iBuffer to the user. An iBuffer can be in one of the following states, `reset, sample, stop`, and `read`. The transitions among the states are

presented in Figure 4.3. A state transition occurs either when there is control information provided through the command channel, or when an event completes in the state machine. Data are written into the trace buffer during the `sample` state, using one of the two ways, linear or cyclic. In the linear scheme, writes to the trace buffer stop when it is full, while in a cyclic mechanism, writes continue until a stop command is issued through command channel. A read command through the command channel moves the state to `read`, during which the data are sent on the output channel. The state moves to `stop` when all the data in the trace buffer are read. Sampling is restarted by resetting the state machine.

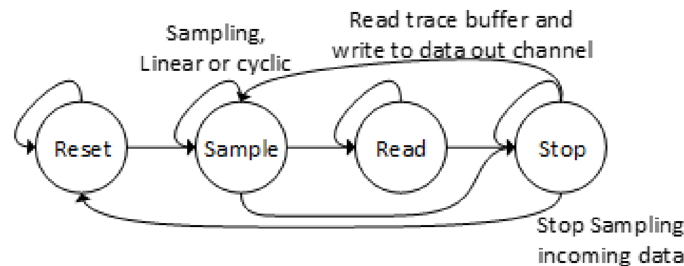


Figure 4.3: The state machine for an iBuffer. States can be changed by writing commands into the command channel, or by completion of tasks in the iBuffer.

As an iBuffer is used for dynamic debugging or profiling, it must not affect the designs under test. This requirement imposes the following challenges. First, writes to the input data channel of the iBuffer should not block the calling site. Second, reads from and writes to the iBuffer should not affect global memory accesses latency of the kernel under test, i.e., debugging or profiling shall not alter the memory behavior of the design under test. The first challenge is addressed with a stall-free pipeline design as shown in Listing 4.8. The outer loop is an infinite loop, and one iteration is launched every cycle. The single-cycle launch of each iteration of the outer loop ensures that incoming data is read from the input data channel each cycle and processed without incurring any data loss or stalls in the caller site pipeline. We guarantee single-cycle launch by avoiding any dependence on the outer loop

variable and by either unrolling all the inner loops or avoiding inner loops. The launching schedule of the outer loop is confirmed with the compiler-generated log that provides details about compiled kernels. The second challenge is addressed by having a trace buffer in local memory, hence writes to this memory do not affect global memory accesses. Users, however, need to ensure that the trace buffer is read out and written to global memory when the kernel under test is not running.

An iBuffer can store information from a particular calling site where it writes data into the input data channel of the iBuffer. Users may want to probe into multiple kernels or have multiple calling sites inside a kernel. This requires multiple iBuffer instances, which are accomplished using the Altera attribute `num_compute_units` `(x,y,z)`. This attribute replicates the kernel in up to three dimensions, where the number of units in each dimension is specified by the variable x, y, and z. The depth (or size) of an iBuffer can be controlled by changing the define $DEPTH$ as in Listing 4.10. This makes iBuffer scalable, for both the depth of the trace buffer and the number of instances, while providing separate command channels that can control each iBuffer instance.

4.3 Use Cases

In this section, we present two use cases of our proposed iBuffer. The first collects the latency to show how a pipeline is stalled. The second provides smart watchpoints, which can do address bound checking and value invariance checking on the fly for specified memory locations.

```
__attribute__((max_global_work_dim(0)))
__attribute__((autorun))
__attribute__((num_compute_units(N,1)))
__kernel void state_machine(void) {
  bool r, rvalid;
  state_e curr_state, next_state;
  uchar id  = get_compute_id(0);
  #pragma acc kernels loop independent
  for(ulong i = 0; i < ULONG_MAX ; i++) {
    take_stamp =
      read_channel_nb_altera(data_in[id], &r);
    next_state =
      read_channel_nb_altera(cmd_c[id], &rvalid);
    if(rvalid) {
      switch(next_state) {
        case RESET: ...
        case STOP: ...
        case SAMPLE: ...
      }   }   }
```

Listing 4.8: Autorun iBuffer persistent kernel.

4.3.1 Pipeline Stall Monitors

Pipeline stalls may occur because of loads or stores accessing global memory, or a throughput difference between a producer and a consumer connected through a channel. A pipeline stall monitor is useful to profile the kernel in such scenarios. By using the HDL-based timestamps and iBuffer framework, we develop a stall monitor as shown in Figure 4.4 that stores timestamps at points of interest in a kernel. A timestamp is taken inside the iBuffer when there is data available to be read at the data input channel. Such information is then written into the trace buffer in either a cyclic or linear fashion. Listing 4.9 shows an example of measuring the load latency of $a = data_a[i * col_a + k]$ in a matrix multiply kernel using the function `take_snapshot`. This example is very similar to Listing 4.4 while the function `take_snapshot` sends in the variable `in` through the data input channel and the timestamp is taken implicitly inside the iBuffer when the data input channel is read. As the iBuffer is stall

```
void take_snapshot(uint id, int in) {
    (void) write_channel_nb_altera(data_in[id],
                                   (int) in);
    mem_fence(CLK_CHANNEL_MEM_FENCE);}
.......
for(int k = 0; k < col_a; k++) {
  take_snapshot(0,k);//snapshot site 1
  int a = data_a[i*col_a + k];
  take_snapshot(1,a);//snapshot site 2
  int b = data_b[k*col_b + j];
  int acc  += a*b;
  ..... }
```

Listing 4.9: Measuring load latency using stall monitor.

free, the latency of the load can be computed as the difference between the two snapshots and the processed trace contains the latency of the load in an execution window determined by the trace buffer depth.

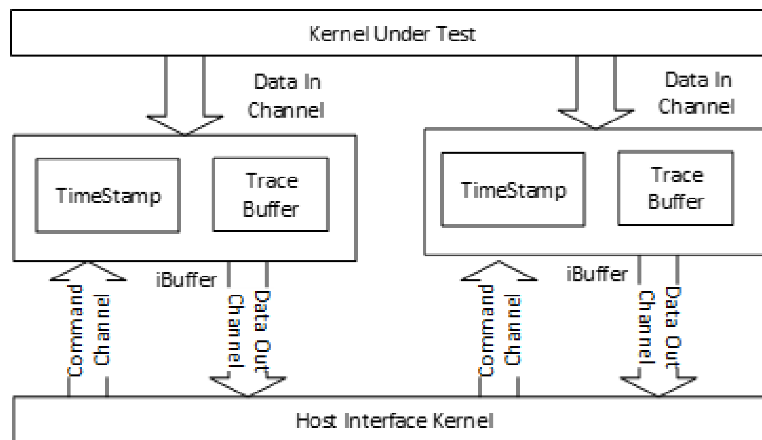


Figure 4.4: Monitoring pipeline-stalls using timestamps and the iBuffer.

For facilitating the host communication with our proposed iBuffer, so as to initiate monitoring and collect the monitored results, a host interface kernel is introduced as shown in Figure 4.4. The implementation of this host interface kernel is shown in Listing 4.10. It works as an agent to forward the command from the host to the iBuffer through the command channel. When the command is a `read`, it then reads the data out channel until all the elements in

```
#define N 10 //iBuffer Count
#define DEPTH 1024 //Trace buffer depth
channel cmd_e cmd_c[N]; //Command channels
channel out_e out_c[N]; //Data channels
__attribute__((max_global_work_dim(0)))
kernel
void read_host (cmd_e cmd, int id, out_e out) {
  #pragma unroll
   for(int i = 0; i < N; i++) {
     if(i == id)
         write_channel_altera(cmd_c[i], cmd);}
 if(cmd == READ) {
    for(int k = 0; k < DEPTH; k++) {
      #pragma unroll
      for(int i = 0; i < N; i++) {
        if(i == id) {
          output[k] =
          read_channel_altera(out_c[id]);
        }   }   }   }   }
```

Listing 4.10: Host interface kernel to forward control commands from host to iBuffer and to read data from iBuffer.

the trace buffer are read. This data is written to global memory, which can be accessed by the host for further post processing.

4.3.2 Smart Watchpoints

A watchpoint monitors how the value at a user-specified location in memory changes over time. As proposed in a prior work [67], additional functionality such as invariance checking or address bound checking can be included to make watchpoints more intelligent. Our design supporting such intelligent watchpoints uses the timestamp function and iBuffer framework, with augmented logic for address monitoring and/or value checking as shown in Figure 4.5. In our design, after setting up the watchpoints, we monitor the memory accesses of interest explicitly. In other words, a user needs to explicitly insert a `monitor_address` function for

```
void add_watch(uint id, size_t address) {
  write_channel_nb_altera(addr_in_c[id], address);}
void monitor_address(uint id, size_t addr,
            ushort tag) {
  in.addr = addr; in.tag = tag;
  write_channel_nb_altera(data_in[id], in);}
.....
add_watch(0,(size_t) &data_a[0]);//Add watch point
for(int k = 0; k < M; k++) {
  b = ...; ...
  a     = addr_a[k];
  //Monitor the read address for bound checking
  monitor_address(0,(size_t) &addr_a[k], a);
  *a    = b;
  //Monitor the write address for
  //     bound checking and value updates
  monitor_address(1,(size_t)a, b);
  ...} ...
```

Listing 4.11: Adding a watchpoint for a specified address and monitoring memory operations.

every possible memory operation that may access the location under watch. Such an example is shown in Listing 4.11. The address to be watched is provided using an additional channel `addr_in_c`. The watchpoint logic compares this address against the addresses sent on the data input channel through the `monitor_address` functions. On a match, the tag specifying the value at the memory location and the timestamp are written into the trace buffer, if the iBuffer is in the `sample` state. The additional checking functions such as address bound checking or value invariance are supported by simply changing the code of iBuffer, where the information is read from the data input `data_in` channels. The user can control the iBuffer through a host interface kernel similar to Listing 4.10.

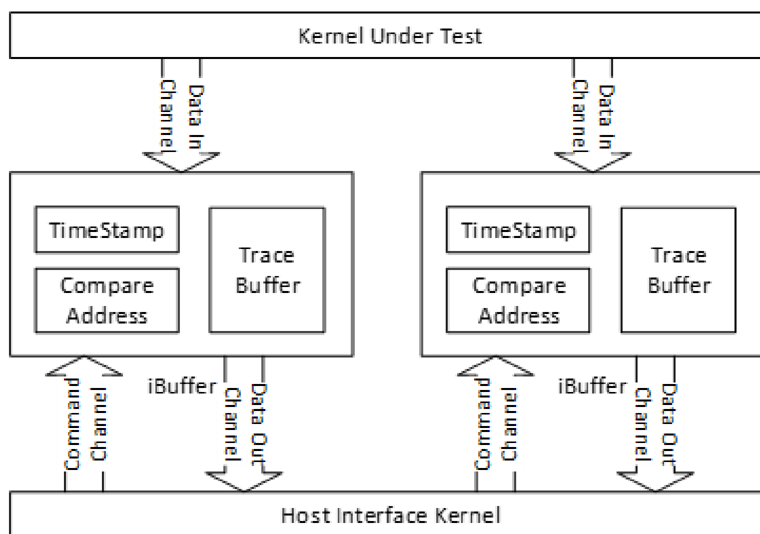


Figure 4.5: Smart watchpoints using timestamp and the iBuffer.

4.4 Evaluation

In this work, we use three different FPGA platforms to validate and evaluate our proposed framework. They include a discrete Stratix V FPGA, a discrete Arria 10 FPGA, and an integrated Arria 10 FPGA in an Intel Broadwell-EP processor. We use Altera SDK for OpenCL, 64-Bit Offline Compiler V16.0 to compile and synthesize circuitry for the FPGAs. We mainly report the results using the Stratix V system as other platforms show similar trends.

In our experiments, we implemented our proposed profiling and debugging support on different kernels. Table 4.1 lists logic and memory usage for generated designs upon a matrix multiplication kernel. The base shows the utilization for the kernel without any debugging/profiling support. When the same kernel is profiled using a stall monitor (SM), it results in small memory overhead and similar logic utilization while the clock frequency is reduced by 20.5%. We observed the similar results with a single watchpoint (WP) and a combination of a watchpoint and a stall-monitor. On a different kernel which performs

Table 4.1: The logic, memory usage, and frequency results.

Type	Clock Freq. (Mhz)	Logic Utilization	Memory Bit	Memory Blocks
Base	245.32	42.02K	2.97M	396
SM	194.96	41.92K	4.16M	414
WP	198.53	42.57K	4.03M	407
SM + WP	198.09	45.84K	4.16M	416

pointer-chasing operations, however, the frequency overhead is less than 3%, as discussed in Section 4.1.1. Therefore, it seems that the overhead is kernel dependent. If the kernel is simple as matrix multiplication which can reach high baseline frequency, the overhead is more evident. Also, as we can see from Table 4.1, the design with a stall monitor has lower logic utilization than the baseline, which implies that the baseline matrix multiplication kernel may benefit from some synthesis optimizations, which trade logic for higher frequency, while the kernels with debugging/profiling support do not.

Chapter 5

An Accelerator for 3D-stencil

Acceleration for 3D-stencil has been mainly done using HDLs. Developing accelerators in HDL requires significant development time and a detailed understanding of logic design, thus adversely impacting productivity. Recent advances in the HLS compilation tools seek to address this issue by allowing a user to generate designs for FPGA by describing it in a high-level language, namely OpenCL. While OpenCL dramatically enhances programmability (when compared to HDL), it does so at the expense of increased area utilization and performance degradation.

In this chapter, we address this design trade-off via a comparative study of our hand-coded design in an HDL (Verilog) and a similar tool-generated design using the OpenCL compilation tool in the application area of 3D-stencil (i.e., structured grid) computation. Compared to the peak performance of a micro-benchmark kernel that assesses the maximum attainable performance, the OpenCL accelerator achieves approximately 95% of the micro-benchmarks' performance for multiple problem sizes. The hand-coded HDL counterpart of the OpenCL accelerator results in approximately 50% less memory usage, about 10% lesser ALM utilization, and only 2.3% better performance on average against the tool generated design.

Following sections in this chapter discuss the peak-performance of a microkernel benchmark, accelerator design in OpenCL and HDL, followed by their evaluation, and comparison addressing the involved area-performance trade-off.

5.1 Peak Performance for a Memory-Bound Kernel

To determine the peak attainable performance for a 3D-stencil problem on FPGA using the OpenCL tool, we design a micro-benchmark kernel (Listing 5.1) that copies all the cells of a problem from one location to another. It copies eight double-precision cells (512-bit) in each iteration of the outer loop (inner loop is unrolled). Copying eight cells per iteration of outer loop provides the best performance (Figure 5.1) because the kernel interface to memory is 512-bit wide, limiting the number of bits read or written in each clock cycle to 64-bytes (8 double floating-point cells).

We report the experimental results on a Bittware board that uses Stratix V FPGA (details listed in Table 2.1) and the Altera offline compiler (aocl) v16.0 to compile the OpenCL kernels.

```
void kernel
benchmark_kernel(__global double *in..){
  for(i=0; i<size/8; i++){
#pragma unroll <Factor>
    for(k=0; k<8;k++){
      out[8*i+k] = in[8*i+k];}}}
```

Listing 5.1: Micro-benchmark kernel for assessing peak performance

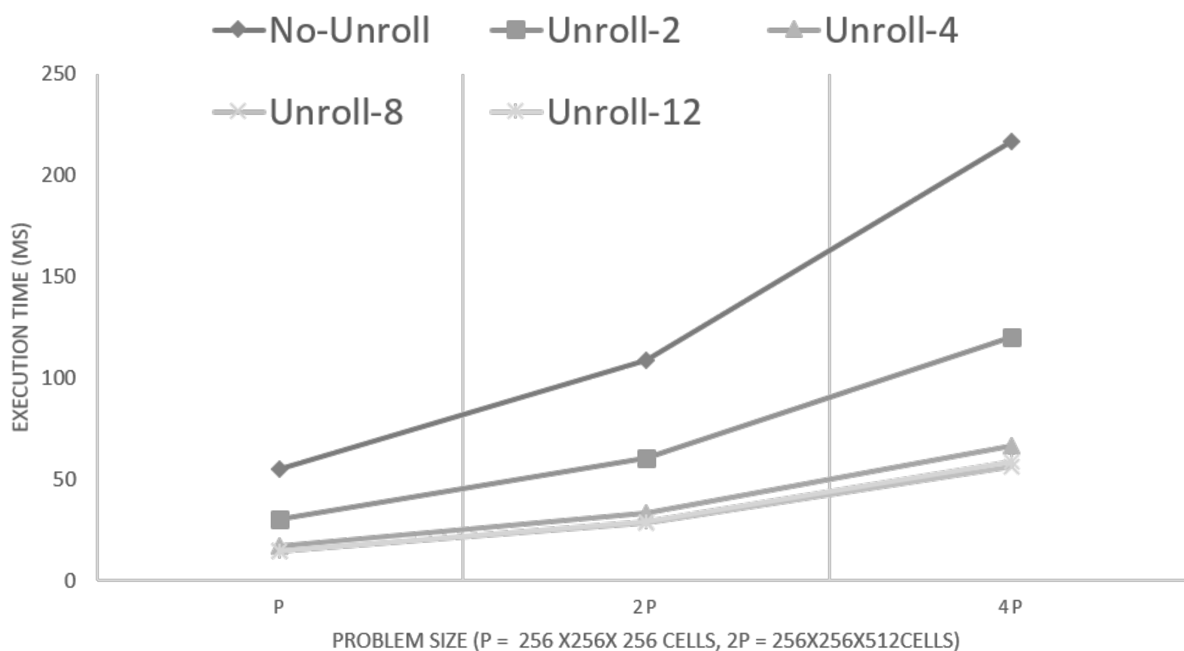


Figure 5.1: Performance of micro-benchmark kernel with different unroll factors (one, two, four, eight, and twelve) for problem sizes 256x256x256 (P), 256x256x512 (2P), 256x256x1024(4P). It performs best with unroll factor of eight, after which unrolling inner loop has diminishing returns.

5.2 OpenCL Accelerator

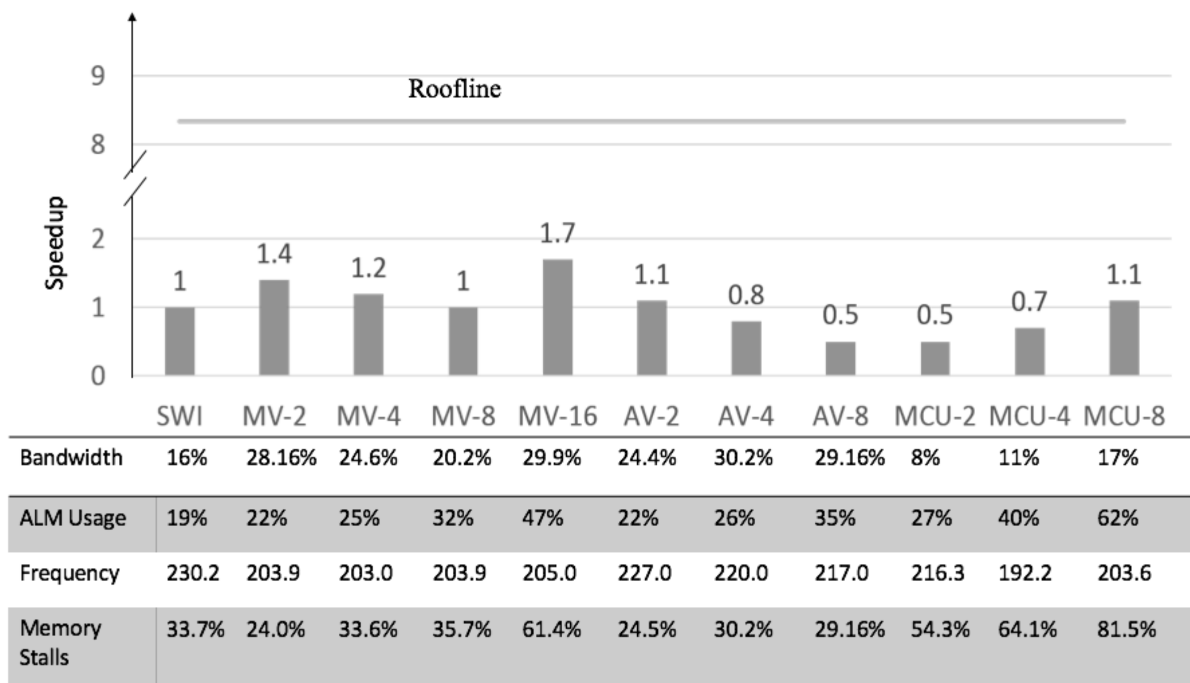
The compiler optimizations discussed in Chapter 3 result in a maximum speed-up of 1.7-times over a simple kernel implementation (Figure 5.2). Even with the compiler optimizations, there remains a performance gap (about 5x) between this enhanced performance and that of the benchmark kernel. The performance gains in SWI (single work-item kernel for 3D-stencil) through compiler optimizations are modest. This modest increase in the performance can be understood from the memory access patterns of the 3D-Stencil kernel and the hardware design constraints that restrict the creation of an efficient stall-free pipeline by the compiler explained further in this section.

The accelerator accesses elements across three layers of a problem to compute a single cell. Having the elements from these layers present in the on-chip memory would provide significant performance gains by reducing the accesses to DRAM, and thereby keeping the pipeline stall-free. In the designs generated by Altera OpenCL compiler, there is a small on-chip cache present as part of load units but it cannot contain all the cells required for computation as the problem size grows. This results in a significant number of on-chip cache-misses, therefore, the higher number of DRAM accesses. Although we cannot confirm the internal details of the loading machine and quantify the cache miss-rate, the cache miss-rate depends on its data-replacement policy and the size of on-chip load cache. Altera tool reports the size of this on-chip cache for load unit as 512Kb.

To improve the kernel performance, and to close this performance gap between attainable peak performance and that of the compiler optimized kernel, we design a hardware accelerator that stores all required cells for computation in an on-chip memory. The design of this accelerator is discussed in the current section followed by its evaluation in Section 5.4.

5.2.1 Problem Fragmentation

We provide the details of an accelerator for a fixed plane size (length-in-cells(L) x width-in-cells(W)) that can solve a 3D-stencil problem whose length-in-cells and width-in-cells are multiple of L and W, respectively. The accelerator does so by fragmenting a problem bigger than its plane size (LxW) into smaller chunks of equal size as shown in Figure 5.3. These equal sized chunks are processed on accelerator sequentially. For example, subproblem (0,2) gets processed before (0,3) followed by (1,0). A subproblem is solved when all of its cells are computed, and then the accelerator proceeds to solve the next subproblem. The limited resource availability on FPGA encouraged us to fragment a problem into multiple



	SWI	MV-2	MV-4	MV-8	MV-16	AV-2	AV-4	AV-8	MCU-2	MCU-4	MCU-8
Bandwidth	16%	28.16%	24.6%	20.2%	29.9%	24.4%	30.2%	29.16%	8%	11%	17%
ALM Usage	19%	22%	25%	32%	47%	22%	26%	35%	27%	40%	62%
Frequency	230.2	203.9	203.0	203.9	205.0	227.0	220.0	217.0	216.3	192.2	203.6
Memory Stalls	33.7%	24.0%	33.6%	35.7%	61.4%	24.5%	30.2%	29.16%	54.3%	64.1%	81.5%

Figure 5.2: Speed-up with respect to a naive single work-item 3D-stencil kernel and corresponding area, bandwidth utilization, and clock frequencies using different compiler optimizations (discussed in Chapter 3). Horizontal line at speedup 8.3x represents the performance of micro-benchmark kernel for the same problem size.

subproblems. The accelerator that has a plane size dimension of 512x512 does not fit on Stratix V board due to the shortage of memory resources (M20K RAMs). It means that in the absence of fragmentation, it would not be possible to solve problems of plane size 512x512 or higher.

5.2.2 Column Transformation

The computation of a boundary element in each subproblem requires cells from neighboring subproblems. Fetching these cells one-by-one results in underutilization of DRAM bandwidth. Therefore, the host transforms the boundary elements represented by dashed lines in Figure 5.3 into rows. Each column of the boundary cells is transformed into a linear array

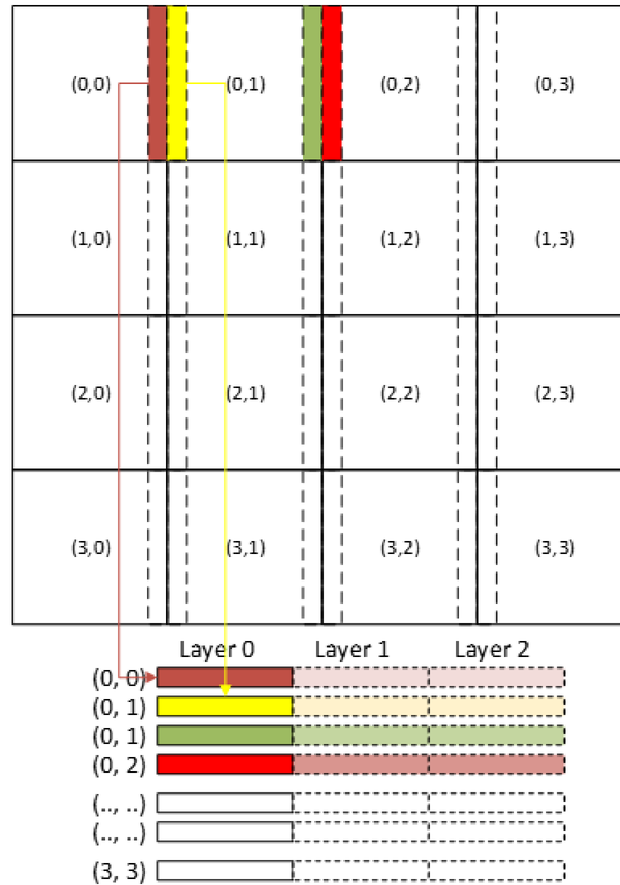


Figure 5.3: Dividing 3D-stencil problem bigger than accelerator plane size into subproblems. Dashed lines represent boundary elements for each subproblem that are transformed by the host.

for each layer of a subproblem. This transformation allows the cells to be accessed in bursts of coalesced memory accesses improving the bandwidth utilization. An element sharing a horizontal boundary with another subproblem does not require any alteration since these are fetched in burst-coalesced accesses. The accelerator discussed in next subsection uses this transformed data and problem fragmentation to accelerate 3D-stencil computation.

5.2.3 Accelerator Design

The OpenCL accelerator has three tasks or single work-item kernels namely `divide_problem`, `compute_stencil`, and `fetch_boundary_elements` communicating with each other using the Altera-channels [3] as represented in Figure 5.4. The task `divide_problem` fragments the problem into subproblems and pushes their corresponding commands into a channel. A command has the subproblem's coordinates, start address of the first cell in this subproblem, the number of layers in the problem, and flags that specify if this subproblem requires boundary elements from an adjacent subproblem. The divide kernel sets a terminate flag in this command to notify the compute kernel if it is the last subproblem, and marks the computation complete upon receiving the response from the compute kernel through a "finish" channel.

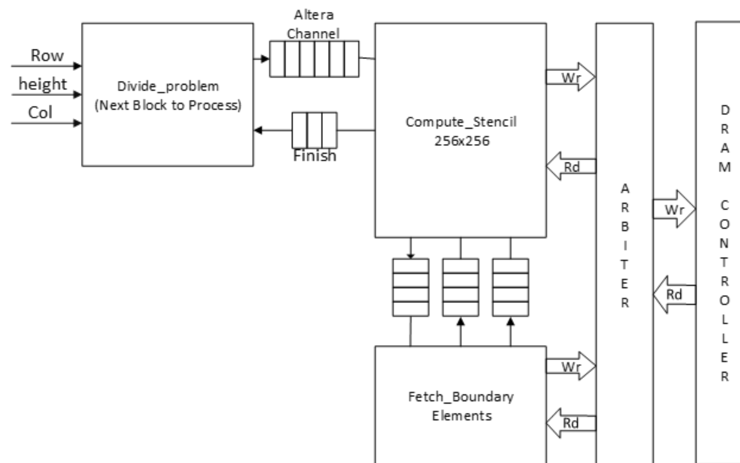


Figure 5.4: Stencil accelerator comprising of single work-item kernels that communicate with each other using the Altera-channels.

The `compute_stencil` kernel reads the command provided by the divide kernel (Listing 5.2), after a successful read it issues a request to fetch boundary elements by writing a request command "req_cmd" to channel "req". This command has all the necessary information needed for the kernel fetch boundary elements. When a write to "req" channel is successful

the compute kernel proceeds to solve the subproblem on its own. It fetches the data from DRAM and stores it into on-chip memory. Once two layers of subproblem are available in the memory, the computation for a cell start. After fetching the two bottom layers, each newly fetched cell provides all the required neighbors for a cell at same (X, Y) coordinates in a lower layer. To process a vertical boundary element, the compute kernel reads the border neighbor via response channels from `fetch_boundary_elements`. Once the subproblem is solved, the compute kernel checks if it was the last subproblem then it issues a finish request to divide kernel, else it elicits a new subproblem by reading the `cmd` channel and issues another request for boundary elements by writing into "req" channel.

The kernel `fetch_boundary_elements` keeps polling channel "req" by non-blocking reads to check if there is an available "req_cmd". After a successful read of "req", the fetch kernel checks if the compute kernel has requested boundary data. If the data is requested, it proceeds to read data from DRAM. Thereafter, it writes the fetched data to either one or both the response channels. If a subproblem requires both the left and right vertical boundary elements, the fetch kernel pushes cell data into both the channels, otherwise it writes into either the left or right response channel. After receiving a new "req_cmd" request, `fetch_boundary_elements` kernel fetches vertical boundary cells and writes the data into its corresponding channel before proceeding to the next boundary. This scheme results in large burst sizes and faster external-memory access time when fetching boundary elements.

The computation of a problem is complete when the divide kernel issues a terminate request to the compute kernel and receives a finish command from the compute kernel in response. The compute kernel issues the finish command after it has calculated all the cells for the last subproblem, and all the outstanding memory write requests are complete. It is ensured by the use of `mem_fence`, which guarantees that all the pending pipelined writes to memory are complete.

```
void kernel
compute_stencil(double *in, double *out,..)
{
   //Buffers P0
   double buf_P0_0[BUF_DEPTH];
   ...
   double buf_P0_7[BUF_DEPTH];
   //Buffer P1
   double buf_P1_0[BUF_DEPTH];
   ...
   double buf_P1_7[BUF_DEPTH];
   //Buffer CB0, CB1
   double buf_cb0, ...
  //infinite counter
  for(ulong i = 0 ; i < ULONG_MAX ; i++) {
    switch(curr_state) {
    case NEXT_SUBPROB:
    bool rvalid;
//channel read to get next subproblem
      read_channel_altera_nb(cmd, rvalid);
      next_state = rvalid ? BOUNDARY_ELEM: NEXT_SUBPROB;
    case BOUNDARY_ELEM:
//channel write to request boundary cells
      write_channel_altera(req, req_cmd);
      next_state = COMPUTE;
    case COMPUTE:
      //Provide the next fetch address
      address = fetch_address();
#pragma unroll 8
      for(int k=0; k<8;k++)
        top_data[k] = in[address+k];
      write_into_local_mem(top_data);
      center[..] = read_frm_local_mem();
      ....
      left[..]= subprob_boundary ? read_channel():
                !prob_boundary   ? read_frm_local_mem() : 0;
      right[..] = ...;
#pragma unroll 8
      for(int k=0; k<8; k++)
        computed[k] = c1*center[k] +
                      c2*top[k]...
      waddress = compute_write_addr();
      if(write) //If valid compute
#pragma unroll 8
        for(int k=0; k<8; k++)
          out[waddress+k] = computed[k];
        next_state = last_cell? NEXT_SUBPROB: COMPUTE;
        if(terminate) { //Last compute
            mem_fence(CLK...); //barrier
            write_channel_altera(end)
        }
    default: break;
    }
    curr_state = next_state;
  }
}
```

Listing 5.2: Pseudo-code for compute 3D-stencil kernel

The compute and the fetch kernels are persistent kernels, which run forever and ensure a single-cycle launch for the outer loop (Listing 5.2). In each iteration of the outer loop, accelerator processes eight adjacent cells (8x64-bit) to take the benefit of a 512-bit wide data bus. The OpenCL code for the solution is parameterized for different plane sizes, and it is used to design accelerators for three different plane sizes 64x64 (OCL-64x64), 128x128(OCL-128x128), and 256x256 (OCL-256x256). The performance and resource utilization of these accelerators are reported in Section 5.4.

5.2.4 On-chip Memory Management

The on-chip memory on the Stratix V board is at most a two-port memory, and processing eight points in each cycle require a significant number of the read ports to access cells individually. The Altera compiler [3] can create banked memory configuration with additional ports. Also, the compiler generated design runs memory at twice the clock frequency (2x-synchronous clock) of kernel-logic. This mechanism enables four reads or writes per clock cycle resembling a four-port memory. A kernel requiring more than four ports in implementation may result in inefficient designs due to pipeline stalls. To overcome this limitation the accelerator uses two local memories (P0, P1) (as shown in Figure 5.5) for storing last two layers of cells for a subproblem. These work as a ping-pong buffer, e.g., P0 stores layer0, followed by layer1 in P1, after that P0 stores layer2. The P0 buffer is a combination of eight smaller equal-sized memories that stores one of the eight cells being processed in that clock cycle.

To further reduce the required number of memory ports, it uses two additional memories (CB0, CB1) that store the last two rows of cells from the current computation layer. This memory partition results in at most two read ports and one write port for each of the memories and thereby avoiding any pipeline stalls emanating from memory read or writes.

If the accelerator is designed to solve only one problem that matches its plane size, then such a complex scheme is not required. On-chip memory can be managed using a simple shift register [65] working as a cyclic buffer (Section 3.3), and neighbors can be read from fixed memory locations from a shift register. Since the problem is fragmented and during computation, some cells require boundary elements, this memory scheme is necessary to create a stall-free design.

Memory contents between two consecutive fetch cycles are represented in Figure 5.5. In

clock cycle k, elements $T[0:7]$ (Figure 5.5 (a))are fetched from DRAM. At this instance location of cells required for computation are shown in the memory tile by gray cells. These locations are read for computation in the current clock cycle and data $D[0:7]$ is stored in buffer CB1 during the cycle $k+1$. In the next clock cycle $(k+1)$, data $T[8:15]$ is fetched (Figure 5.5(b)) which would then be stored to on-chip memory in clock cycle $k+2$.

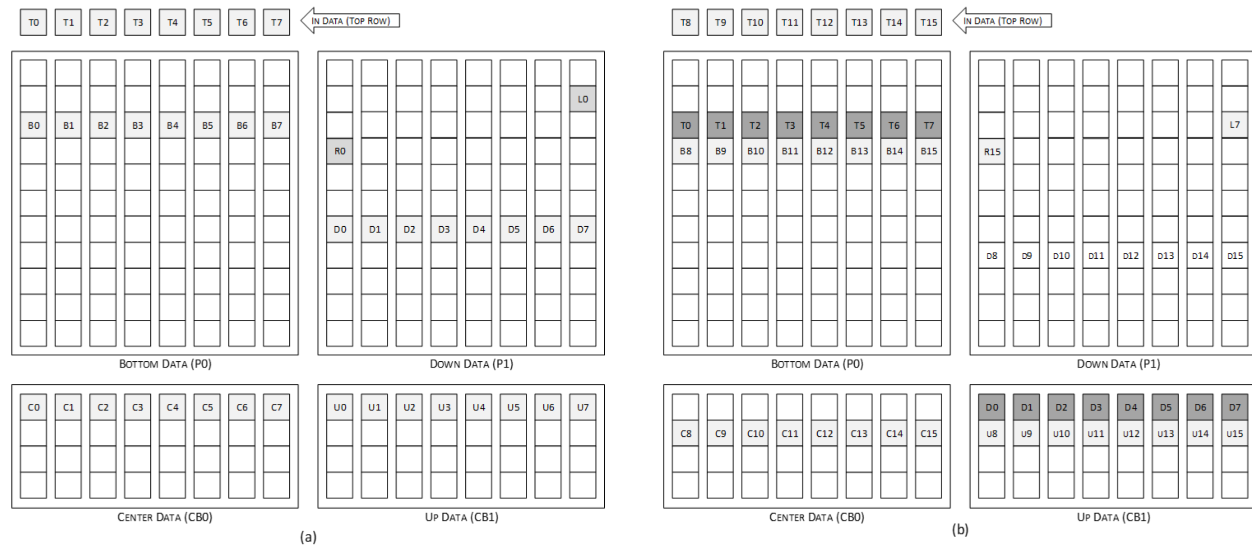


Figure 5.5: On-chip memory data layout for two consecutive fetch cycles. (a) Memory P1 provides the down data, memory P2 provides bottom data, top data is fetched from DRAM. Memories CB0 and CB1 provide center and north data. (b) Top data and south data fetch in the previous cycle gets stores in memory P0, and CB1 respectively. All other elements required for computation are fetched from the memory location next to previously accessed location.

5.3 HDL Accelerator

This section discusses an accelerator (HDL-256x256) for a 256x256 double-precision plane size, designed in a standard widely used HDL namely, Verilog. To detail a fair comparison between the two design choices, HDL and OpenCL; its architecture remains same as OpenCL accelerator discussed in Section 5.2. For reducing the development time, in particular, the

development time of device-drivers for host PCIe, and memory controllers integration, the available advanced features from Altera OpenCL compiler are used. These advanced features allow the user to create a library function for an HDL design and use it in a kernel. The Altera Megafunctions (FIFOs, memories, floating-point adders, and multipliers) are also used as intellectual property (IPs) to cut-down the development time. The HDL accelerator is used as a library function in an empty kernel as shown in Listing 5.3. The compiler integrates this HDL design to compile the kernels using a user-specified "XML" bindings. This XML file contains the information about available ports in HDL design and how these input and output ports map to a PCIe and DRAM interfaces. The HDL module interfaces with DRAM using Avalon [68] interface and connects to the kernel pipeline using a `valid` and `ready` interface. A simple valid/ready protocol qualifies the data-bus when both the signals are high.

```
__attribute__((max_global_work_dim(0)))
__attribute__((num_compute_units(1,1)))
void kernel
stencil_kernel(__global double *ptr_in..){
    stencil_rtl(ptr_in, ptr_col, ptr_out,
                row, col,height);}//RTL Lib
```

Listing 5.3: Stencil kernel with RTL library

The HDL accelerator (Figure 5.6) uses two load-control units to fetch the cells and transformed columns from memory. It also has a store-control unit to store the computed stencil value for a cell to memory. Each of the load-control and the store-control units has an address generation unit that iterates over each of the subproblem depicted in Figure 5.7(a) and provides the computed address for the store and fetch. After going through the reset, these units wait for a valid input from the pipeline and then proceed to computation. The load-control and store-control units connect to 128-deep FIFOs to store or fetch the data. This ensures that there is no data loss or unnecessary pipeline stalls during an ongoing com-

putation. A start/finish controller inside the HDL design manages the interface with the kernel; this controller provides the start signal and data necessary for stencil calculation such as pointer addresses, and dimensions of the grid to accelerator module.

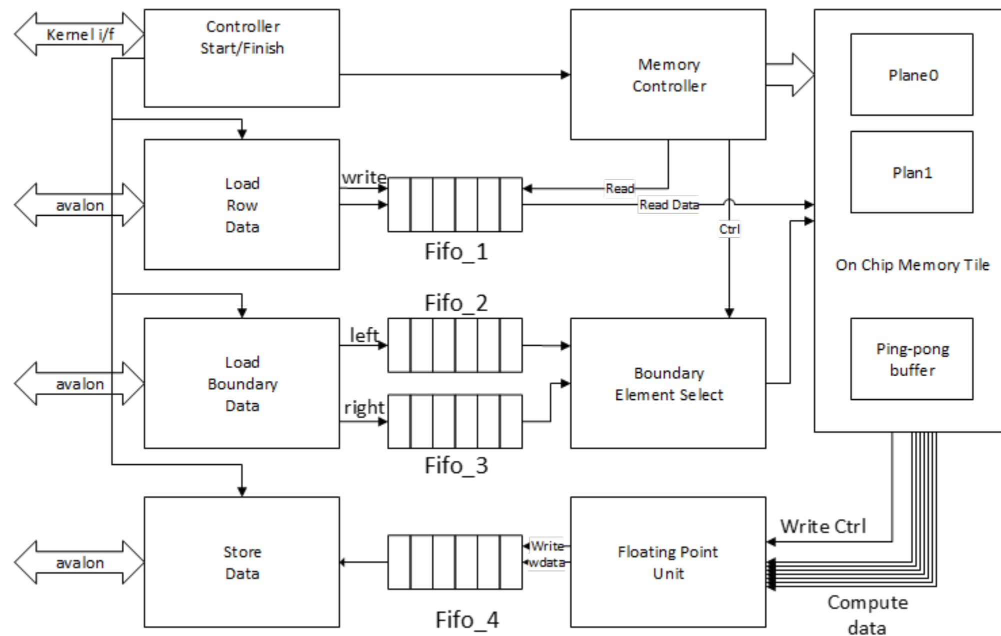


Figure 5.6: Block diagram for the HDL implementation of stencil kernel that connects to memory arbiter using Avalon memory ports

Once start/finish controller indicates a start, after a delay, each of the load-control machines begins to access memory in the maximum allowed bursts of sixteen, fetching 1K-bytes in each burst. The on-chip memory controller modules read the data from fifo_1 when data is available in the FIFO, and then provides necessary control signals to the on-chip-memory-tile to write and/or read this data. This on-chip-memory-tile (Figure 5.5) design is same as discussed in subsection 5.2.4. The outputs from the on-chip-memory-time, along with other control signals go to a pipelined floating point unit (FPU), which computes the stencil values as shown in Figure 5.7(b). If the output of the FPU is a valid calculated stencil cell, it is written into in fifo_4. Once fifo_4 has accumulated sufficient data required for a burst-write transfer to DRAM, it issues to write request to DRAM-controller. The pipeline stalls when

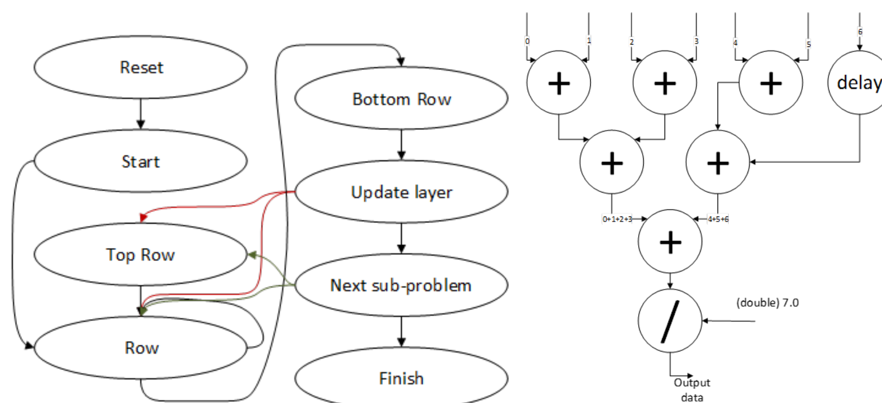


Figure 5.7: (a) State machine for address generation (b) Floating point unit for 3d-stencil computation

the number of empty spaces in the store-FIFO equals to the latency in clock cycle count for data to travel from "A" to "B". Here, "A" refers to the instance when the FPU reads data from on-chip memory, and "B" relates to the instance when corresponding computed cell is written into fifo_4. This stall in pipeline avoids any overflow at the store-FIFO (when the DRAM controller is busy servicing load requests or memory-refresh) and ensures that it has sufficient empty spaces to store cells already in computation pipeline. The load-boundary-data module fetches transformed column data (if required) whenever there is enough empty space (equal to or more than burst size in the corresponding FIFOs).

In the next section, we discuss performance for the OpenCL and HDL implementations.

5.4 Evaluation

Performance

This section discusses the performance of the stencil accelerator created using the OpenCL compiler for three different plane sizes namely 64x64, 128x128, and 256x256 (OCL-64x64,

OCL-128x128, and OCL-256x256, respectively) and its HDL (HDL-256x256 for plane size 256x256) implementation. Figure 5.8 shows the speed-up for the accelerators OCL-64x64, OCL-128x128, and OCL-256x256 with respect to the performance of a micro-benchmark kernel (Listing 5.1). The accelerators OCL-64x64 and OCL-128x128 perform best when the problem plane-size matches the accelerator plane-size precisely (P_64_cubed for OCL-64x64, P_128_cubed for OCL-128x128) due to the benefit received from linear memory accesses. As the problem plane size grows, the performance of OCL-64x64 and OCL-128x128 deteriorates due to the access of different DRAM row-buffers. This results in inefficient bandwidth utilization which further degrades the overall performance.

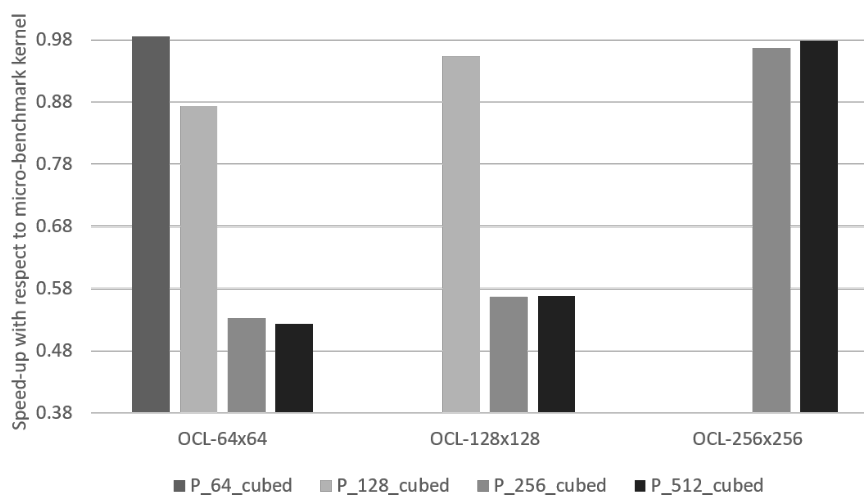


Figure 5.8: Speed-up for the OpenCL accelerators for different plane size with respect to micro-benchmark kernel. Results are reported for problem sizes of 64x64x4096 (P_64_cubed), 128x128x128 (P_128_cubed), 256x256x256 (P_256_cubed), and 512x512x512 (P_512_cubed). (Note: Bars corresponding to plane sizes smaller than accelerator plane size are not reported)

A similar observation in deterioration of performance is not noted in the case of OCL-256x256 since the burst-size perfectly matches the DRAM row-buffer size.

Figure 5.9 provides the detailed information about the usage of on-chip resources and its as-

sociated clock frequencies for different accelerators. The accelerator OCL-256x256 consumes the maximum on-chip memory because it has the largest on-chip-memory-tile. Additionally, high resource utilization impacts the maximum operating clock frequency since routing the signals becomes tough for the placement tool. This is evident with the OCL-256x256 that has the least F_{max} in our experiments.

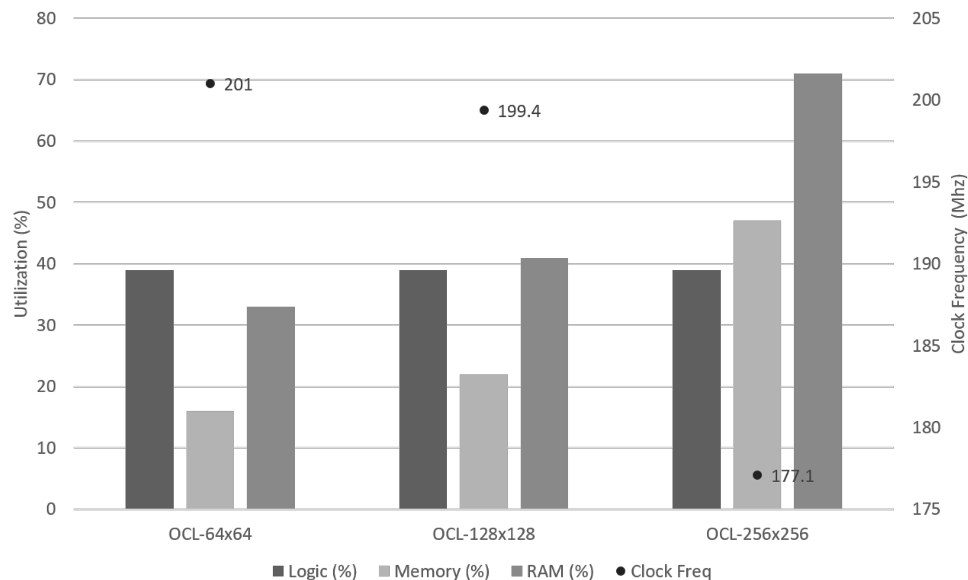


Figure 5.9: Memory, ALM, RAM usage, and clock frequencies for the OpenCL-accelerators with different plane sizes

Figure 5.10 provides the performance comparison of OCL-256x256, HDL-256x256, and a naive single-work item kernel (SWI discussed in the chapter 3) for various problem sizes. The HDL-256x256 performs 2.24% better on an average, and at most 4.4% better than the OCL-256x256. This performance difference can be attributed to a lower kernel clock frequency of OCL-256x256 listed in Table 5.1. The accelerator HDL-256x256 uses 47% lesser on-chip memory blocks, 53% lower memory bits, and 10% lesser logic utilization. Additionally, it runs at a kernel clock frequency 22% faster than that of OCL-256x256. Disregarding the board support package area would result in the higher resource usage savings for the HDL-256x256. On an average, the performance of the HDL-256x256 (or OCL-256x256) is approximately 8x

better than the SWI implementation.

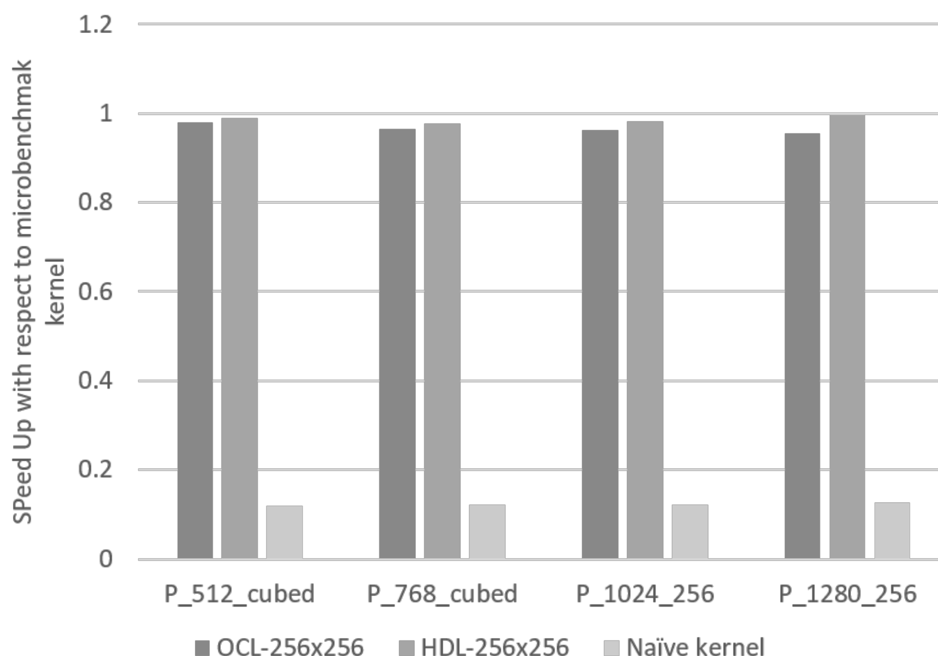


Figure 5.10: Speed-up for OpenCL (OCL-256x256 and naive task implementation SWI) and RTL (HDL-256x256) accelerators with respect to micro-benchmark kernel performance for problem sizes of 512x512x512 (P_512_cubed), 768x768x768 (P_768_cubed), 1024x1024x256 (P_1024_256), and 1280x1280x256 (P_1280_256).

Table 5.1: The logic, memory usage, and frequency comparison for OpenCL and RTL accelerators for plane size 256x256.

Type	Clock Freq. (Mhz)	Logic Utilization	Memory Bit	Memory Blocks
OCL-256x256 (OpenCL)	177.115	101.635K	24.65M	1815
SWI (OpenCL)	261.09	43.383K	6.896M	584
HDL-256x256 (RTL)	215.93	90.69K	11.582M	965

The memory bandwidth utilization for the different accelerators is represented in Table 5.2, as reported by the Altera profiling tool. The accelerator HDL-256x256 has a slight advantage in bandwidth utilization than OCL-256x256 resulting in marginal gains in the overall performance. These results indicate that the compiler can still be improved further to address this

performance gap to generate highly efficient pipelines for simple kernel implementations in the absence of guidance, either from compiler directives or hand-tuned algorithm refactoring.

Table 5.2: The memory bandwidth utilization for different accelerators.

Type	Bandwidth Usage (in percentage)	Problem size
OCL-64x64	68.81%	128x128x128
OCL-128x128	70.80%	128x128x128
OCL-256x256	72.29%	256x256x256
HDL-256x256	73.21%	256x256x256

Scalability

The performance gains on FPGA come from the deep-pipelining. The best performance gains occur when the accelerator for 3D-stencil can compute maximum available cells in each clock cycle. Any stalls in this pipeline would deteriorate the performance. In this thesis, we discussed a 3D-7Point Stencil algorithm and designed a deep pipelined system that could process 8-double floating-point cells per clock cycle (the maximum available). A stall-free and high-performance design was possible because of the efficient memory tiling that could provide all the cells required for computation in each clock cycle. As the required cells for the computation of a point in 3D-stencil would increase, this tiling mechanism may need change to run without any stalls. This change would also be constrained by the limited number of read-write ports on memory and will require efficient banking of the memory tiles, for example, a 27-point stencil algorithm may need a much more complex scheme for on-chip memory to produce a highly efficient pipeline on FPGA. For a 2D-stencil problem, a shift register or sliding window approach [35, 34, 36] is better because of its low memory requirement.

Programmability

To address enhanced programmability, we provide a rough sketch of the development time and the source line of code (SLOC). The kernel development time for the HDL accelerator was significantly higher (about 4-5x) than the OpenCL counterpart. The SLOC is 650 lines for OCL-256x256, in contrast, the HDL accelerator (HDL-256x256) has the SLOC (excluding Altera IPs) of 3350. Additionally, an HDL accelerator development requires a verification environment and test-suite for functionality checks in the RTL simulation. This overhead has an approximate SLOC of 2000, in Verilog and System-Verilog (excluding the Altera bus functional models [69]).

We should note that much of the SLOC in an HDL design is used in the declaration of variables and module interconnections. Additionally, HDLs are primarily design languages and inherently different from programming languages. Even among the HDL models, the SLOC varies significantly by the design choices. For example, a structured HDL design or netlist, that instantiates basic gates and flops would have a significantly large amount of code than an equivalent behavioral HDL. Hence, a direct comparison using the SLOC between a hardware design and an equivalent program is not a fair comparison.

The quantification of programmability in itself is a complex research problem. The cyclomatic complexity [70] provides the complexity of a program by computing linearly independent paths in it, employing a control flow graph. This approach to compare a procedural programming paradigm such as OpenCL, with a hardware design language that is inherently parallel and concurrent would not yield the correct comparison. Addressing the methods to compare the programmability is outside the scope of this thesis, and hence we resorted to the simplest approach; by reporting the development time and source lines of code comparison.

Chapter 6

Conclusions and Future Directions

6.1 Conclusion

The Chapter 3 discusses the performance-programmability gap in employing the OpenCL to program an FPGA. It starts with the GPU-favorable implementations of two applications from the OpenDwarfs benchmark ported to an FPGA without architectural considerations and explores the compiler optimizations to accelerate them. Furthermore, it provides a refactored 3D-stencil algorithm and its implementation as a highly efficient accelerator using the OpenCL compiler. The results indicate that OpenCL programming model can indeed be used to create efficient accelerators. Additionally, this exhibits the potential of the reconfigurable architectures, their increased programmability with the OpenCL which does not require the detailed knowledge of digital design.

The Chapter 4 presents a source-level scalable framework for debugging and profiling OpenCL for FPGA designs. It includes efficient timestamps and sequencing primitives and an intelligent trace buffer that can process data besides recording them on the fly. The experiments

in this chapter show that the proposed framework has a small overhead in area and frequency, and provides a software-like interface to enable the user to profile and debug the kernel functions efficiently.

Finally in the Chapter 5, the experiments demonstrate that a carefully designed kernel in a high-level language for a regular memory access pattern algorithm can match the performance of a manually designed HDL accelerator at the expense of area utilization and maximum frequency. However, shorter design and verification time for the OpenCL is lucrative for the user. The experiments show that there is still a substantial performance gap for designs generated from a simple kernel implementation using compiler tools when compared to the architecture-aware kernel. This chapter also discusses the efficacy of compiler optimization in the context of a naive kernel implementation and provides a contrast in productivity for the OpenCL and HDL approaches.

We sincerely hope that the findings of this thesis and the open-source profiling framework would help the future development of the OpenCL designs targeted for FPGAs as well as the HDL compiler improvements.

6.2 Future Directions

The debug and profiling library presented in this thesis can provide a fine-grained information but that comes with additional overhead as this debug logic is generated by the compiler. Instead, if this logic is optimized in an RTL module and added to the already existing profiling tools, the area overhead and frequency impact on design can be mitigated to a large degree.

This work does not discuss the energy efficiency for the FPGA when compared to other fixed

architectures such as CPUs or GPUs. It can be shown in future that the FLOPS/watt for FPGA is significantly higher for fundamental algorithms while striving for performance that is at least comparable to GPUs.

The experiments in this use an old generation of the reconfigurable hardware. Currently available FPGAs such as Stratix-10 [71] have a considerably large number of resources, hardened floating-point units, and improved memory bandwidth. These would significantly improve the performance of the FPGA designs. The additional available area on the Stratix-10 can be used to put multiple accelerators on-chip and compute various iterations of the stencil algorithm before the host starts another iteration.

Additionally, the presented stencil accelerator in this work that can be used to create a parameterized library for high-performance computation of stencil applications. The implementation in this thesis supports a 7-point implementation for 3D-stencil, which can be generalized to build a library for 2D or 3D stencil problems supporting different configurations that can readily be employed by the programmers.

Bibliography

[1] V. Stratix, "Device handbook, volume 1: Device interfaces and integration," *Altera, June*, 2012.

[2] K. O. W. Group *et al.*, "The opencl extension specification, version 2.0.(2014)," *Document revision*, vol. 26, 2014.

[3] Altera, *Altera SDK for OpenCL: Best Practices Guide*, 2015.

[4] A. Putnam, A. M. Caulfield, E. S. Chung, D. Chiou, K. Constantinides, J. Demme, H. Esmaeilzadeh, J. Fowers, G. P. Gopal, J. Gray, *et al.*, "A reconfigurable fabric for accelerating large-scale datacenter services," in *Computer Architecture (ISCA), 2014 ACM/IEEE 41st International Symposium on*, pp. 13–24, IEEE, 2014.

[5] M. Lavasani, H. Angepat, and D. Chiou, "An fpga-based in-line accelerator for memcached," *IEEE Computer Architecture Letters*, vol. 13, no. 2, pp. 57–60, 2014.

[6] D. Chiou *et al.*, "Cryptoraptor: High throughput reconfigurable cryptographic processor," in *Proceedings of the 2014 IEEE/ACM International Conference on Computer-Aided Design*, pp. 154–161, IEEE Press, 2014.

[7] K. Ovtcharov, O. Ruwase, J.-Y. Kim, J. Fowers, K. Strauss, and E. S. Chung, "Accelerating deep convolutional neural networks using specialized hardware," *Microsoft Research Whitepaper*, vol. 2, no. 11, 2015.

[8] D. Mahajan, J. Park, E. Amaro, H. Sharma, A. Yazdanbakhsh, J. K. Kim, and H. Esmaeilzadeh, "Tabla: A unified template-based framework for accelerating statistical machine learning," in *High Performance Computer Architecture (HPCA), 2016 IEEE International Symposium on*, pp. 14–26, IEEE, 2016.

[9] S. A. Fahmy, K. Vipin, and S. Shreejith, "Virtualized fpga accelerators for efficient cloud computing," in *Cloud Computing Technology and Science (CloudCom), 2015 IEEE 7th International Conference on*, pp. 430–435, IEEE, 2015.

[10] C. De Schryver, I. Shcherbakov, F. Kienle, N. Wehn, H. Marxen, A. Kostiuk, and R. Korn, "An energy efficient fpga accelerator for monte carlo option pricing with the heston model," in *Reconfigurable Computing and FPGAs (ReConFig), 2011 International Conference on*, pp. 468–474, IEEE, 2011.

[11] K. Sano, T. Iizuka, and S. Yamamoto, "Systolic architecture for computational fluid dynamics on fpgas," in *Field-Programmable Custom Computing Machines, 2007. FCCM 2007. 15th Annual IEEE Symposium on*, pp. 107–116, IEEE, 2007.

[12] J. W. Lockwood, N. Naufel, J. S. Turner, and D. E. Taylor, "Reprogrammable network packet processing on the field programmable port extender (fpx)," in *Proceedings of the 2001 ACM/SIGDA ninth international symposium on Field programmable gate arrays*, pp. 87–93, ACM, 2001.

[13] D. F. Bacon, R. Rabbah, and S. Shukla, "Fpga programming for the masses," *Communications of the ACM*, vol. 56, no. 4, pp. 56–63, 2013.

[14] H. Ren, "A brief introduction on contemporary high-level synthesis," in *IC Design & Technology (ICICDT), 2014 IEEE International Conference on*, pp. 1–4, IEEE, 2014.

[15] J. Bachrach, H. Vo, B. Richards, Y. Lee, A. Waterman, R. Avižienis, J. Wawrzynek, and K. Asanović, "Chisel: constructing hardware in a scala embedded language," in *Proceedings of the 49th Annual Design Automation Conference*, pp. 1216–1225, ACM, 2012.

[16] C. Catapult, "Synthesis overview," 2015.

[17] L. Wirbel, "Xilinx sdaccel: A unified development environment for tomorrows data center," *The Linley Group Inc*, 2014.

[18] S. Sirowy and A. Forin, "Wheres the beef? why fpgas are so fast," *Microsoft Research, Microsoft Corp., Redmond, WA*, vol. 98052, 2008.

[19] G. Martin and G. Smith, "High-Level Synthesis: Past, Present, and Future," *IEEE Design Test of Computers*, vol. 26, pp. 18–25, July 2009.

[20] S. A. Edwards, "The Challenges of Synthesizing Hardware from C-Like Languages," *IEEE Design Test of Computers*, vol. 23, pp. 375–386, May 2006.

[21] S. Windh, X. Ma, R. Halstead, P. Budhkar, Z. Luna, O. Hussaini, and W. Najjar, "High-Level Language Tools for Reconfigurable Computing," *Proceedings of the IEEE*, vol. 103, pp. 390–408, March 2015.

[22] G. Inggs, S. Fleming, D. Thomas, and W. Luk, "Is High Level Synthesis Ready for Business? An Option Pricing Case Study," in *FPGA Based Accelerators for Financial Applications*, pp. 97–115, Springer International Publishing, 2015.

[23] D. Chen and D. Singh, "Invited paper: Using OpenCL to Evaluate the Efficiency of CPUS, GPUS and FPGAS for Information Filtering," in *2012 22nd International Conference on Field Programmable Logic and Applications (FPL)*, pp. 5–12, Aug 2012.

[24] G. Ndu, J. Navaridas, and M. Luján, "CHO: Towards a Benchmark Suite for OpenCL FPGA Accelerators," in *Proceedings of the 3rd International Workshop on OpenCL*, IWOCL '15, (New York, NY, USA), pp. 10:1–10:10, ACM, 2015.

[25] K. Hill, S. Craciun, A. George, and H. Lam, "Comparative Analysis of OpenCL vs. HDL with Image-Processing Kernels on Stratix-V FPGA," in *2015 IEEE 26th International Conference on Application-specific Systems, Architectures and Processors (ASAP)*, pp. 189–193, July 2015.

[26] D. Chen and D. Singh, "Fractal Video Compression in OpenCL: An Evaluation of CPUs, GPUs, and FPGAs as Acceleration Platforms," in *2013 18th Asia and South Pacific Design Automation Conference (ASP-DAC)*, pp. 297–304, Jan 2013.

[27] B. Vermeulen and S. K. Goel, "Design for debug: Catching design errors in digital chips," *IEEE Design & Test*, vol. 19, no. 3, pp. 37–45, 2002.

[28] J. Goeders and S. J. Wilton, "Effective fpga debug for high-level synthesis generated circuits," in *FPL'2014*, pp. 5–8, ACM, 2014.

[29] J. Goeders and S. J. Wilton, "Using dynamic signal-tracing to debug compiler-optimized hls circuits on fpgas," in *FCCM'2015*, pp. 5–8, ACM, 2015.

[30] E. Hung and S. Wilton, "Speculative debug insertion for fpgas," in *FPL'2011*, pp. 524–531, IEEE, 2011.

[31] N. Calagar, S. D. Brown, and J. H. Anderson, "Source-level debugging for fpga high-level synthesis," in *Field Programmable Logic and Applications (FPL), 2014 24th International Conference on*, pp. 1–8, IEEE, 2014.

[32] L. Yang, M. others Ikram, S. Gurumani, S. Fahmy, D. Chen, and K. Rupnow, "Jit trace-based verification for high-level synthesis," in *FPT'2015*, pp. 228–231, IEEE, 2015.

[33] J. Monson and B. Hutchings, "Using source-level transformations to improve high-level synthesis debug and validation on fpgas," in *FPGA 2015*, pp. 5–8, ACM, 2015.

[34] H. M. Waidyasooriya, Y. Takei, S. Tatsumi, and M. Hariyama, "Opencl-based fpga-platform for stencil computation and its optimization methodology," *IEEE Transactions on Parallel and Distributed Systems*, 2016.

[35] K. Sano, Y. Hatsuda, and S. Yamamoto, "Multi-fpga accelerator for scalable stencil computation with constant memory bandwidth," *IEEE Transactions on Parallel and Distributed Systems*, vol. 25, no. 3, pp. 695–705, 2014.

[36] Q. Jia and H. Zhou, "Tuning stencil codes in opencl for fpgas," in *Computer Design (ICCD), 2016 IEEE 34th International Conference on*, pp. 249–256, IEEE, 2016.

[37] H. R. Zohouri, N. Maruyamay, A. Smith, M. Matsuda, and S. Matsuoka, "Evaluating and optimizing opencl kernels for high performance computing with fpgas," in *High Performance Computing, Networking, Storage and Analysis, SC16: International Conference for*, pp. 409–420, IEEE, 2016.

[38] S. Che, M. Boyer, J. Meng, D. Tarjan, J. W. Sheaffer, S.-H. Lee, and K. Skadron, "Rodinia: A benchmark suite for heterogeneous computing," in *Workload Characterization, 2009. IISWC 2009. IEEE International Symposium on*, pp. 44–54, Ieee, 2009.

[39] Z. Wang, J. Paul, B. He, and W. Zhang, "Multikernel data partitioning with channel on opencl-based fpgas," *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, 2017.

[40] K. Hill, S. Craciun, A. George, and H. Lam, "Comparative analysis of opencl vs. hdl with image-processing kernels on stratix-v fpga," in *Application-specific Systems, Architectures and Processors (ASAP), 2015 IEEE 26th International Conference on*, pp. 189–193, IEEE, 2015.

[41] M. Gort and J. Anderson, "Design re-use for compile time reduction in fpga high-level synthesis flows," in *Field-Programmable Technology (FPT), 2014 International Conference on*, pp. 4–11, IEEE, 2014.

[42] C. Lavin, B. Nelson, and B. Hutchings, "Impact of hard macro size on fpga clock rate and place/route time," in *Field Programmable Logic and Applications (FPL), 2013 23rd International Conference on*, pp. 1–6, IEEE, 2013.

[43] C. Lavin, M. Padilla, J. Lamprecht, P. Lundrigan, B. Nelson, and B. Hutchings, "Hmflow: accelerating fpga compilation with hard macros for rapid prototyping," in *Field-Programmable Custom Computing Machines (FCCM), 2011 IEEE 19th Annual International Symposium on*, pp. 117–124, IEEE, 2011.

[44] R. Tessier, "Fast placement approaches for fpgas," *ACM Transactions on Design Automation of Electronic Systems (TODAES)*, vol. 7, no. 2, pp. 284–305, 2002.

[45] C. Kao, "Benefits of partial reconfiguration," *Xcell journal*, vol. 55, pp. 65–67, 2005.

[46] M. Feilen, A. Iliopoulos, M. Ihmig, and W. Stechele, "Partitioning and context switching for a reconfigurable fpga-based dab receiver," in *Design and Architectures for Signal and Image Processing (DASIP), 2012 Conference on*, pp. 1–8, IEEE, 2012.

[47] R. Wiśniewski, M. Wiśniewska, and M. Adamski, “Effective partial reconfiguration of logic controllers implemented in fpga devices,” in *Design of Reconfigurable Logic Controllers*, pp. 45–55, Springer, 2016.

[48] A. Verma, A. E. Helal, K. Krommydas, and W.-C. Feng, “Accelerating workloads on fpgas via opencl: A case study with opendwarfs,” tech. rep., Department of Computer Science, Virginia Polytechnic Institute & State University, 2016.

[49] K. Krommydas, A. E. Helal, A. Verma, and W.-C. Feng, “Bridging the performance-programmability gap for fpgas via opencl: A case study with opendwarfs,” in *Field-Programmable Custom Computing Machines (FCCM), 2016 IEEE 24th Annual International Symposium on*, pp. 198–198, IEEE, 2016.

[50] A. Verma, H. Zhou, S. Booth, R. King, J. Coole, A. Keep, J. Marshall, and W.-c. Feng, “Developing dynamic profiling and debugging support in opencl for fpgas,” in *Proceedings of the 54th Annual Design Automation Conference 2017*, p. 56, ACM, 2017.

[51] Altera, *Design Debugging Using the SignalTap II Logic Analyzer*, Nov 2013.

[52] Xilinx, *ChipScope Pro Software and Cores: User Guide*, Apr 2012.

[53] J. Shalf, “The new landscape of parallel computer architecture,” in *Journal of Physics: Conference Series*, vol. 78, p. 012066, IOP Publishing, 2007.

[54] H. Sutter, “The free lunch is over: A fundamental turn toward concurrency in software,” *Dr. Dobbs journal*, vol. 30, no. 3, pp. 202–210, 2005.

[55] R. H. Dennard, F. H. Gaensslen, V. L. Rideout, E. Bassous, and A. R. LeBlanc, “Design of ion-implanted mosfet’s with very small physical dimensions,” *IEEE Journal of Solid-State Circuits*, vol. 9, no. 5, pp. 256–268, 1974.

[56] A. OpenGL, M. Woo, J. Neider, and T. Davis, "Opengl programming guide," *Addison-Wesley*, 1999.

[57] C. Nvidia, "Compute unified device architecture programming guide," 2007.

[58] O. S. Committee *et al.*, "The openacc application programming interface, version 2.0," *Standard document, OpenACC-standard. org*, 2013.

[59] K. O. W. Group *et al.*, "The opencl specification," *version*, vol. 1, no. 29, p. 8, 2008.

[60] T. S. Czajkowski, U. Aydonat, D. Denisenko, J. Freeman, M. Kinsner, D. Neto, J. Wong, P. Yiannacouras, and D. P. Singh, "From opencl to high-performance hardware on fpgas," in *Field Programmable Logic and Applications (FPL), 2012 22nd International Conference on*, pp. 531–534, IEEE, 2012.

[61] J. Tompson and K. Schlachter, "An introduction to the opencl programming model," *Person Education*, vol. 49, 2012.

[62] S. V. Adve and M. D. Hill, "Weak orderinga new definition," in *ACM SIGARCH Computer Architecture News*, vol. 18, pp. 2–14, ACM, 1990.

[63] Altera, *Altera SDK for OpenCL Custom Platform Toolkit User Guide 1*, May 2016.

[64] K. Krommydas, W.-c. Feng, C. D. Antonopoulos, and N. Bellas, "Opendwarfs: Characterization of dwarf-based benchmarks on fixed and reconfigurable architectures," *Journal of Signal Processing Systems*, vol. 85, no. 3, pp. 373–392, 2016.

[65] S. O. Settle, "High-performance dynamic programming on fpgas with opencl," in *Proc. IEEE High Perform. Extreme Comput. Conf.(HPEC)*, pp. 1–6, 2013.

[66] A. Verma, H. Zhou, S. Booth, R. King, J. Coole, A. Keep, J. Marshall, and W.-c. Feng, "Developing dynamic profiling and debugging support in opencl for fpgas," in *Proceedings of the 54th Annual Design Automation Conference 2017*, p. 56, ACM, 2017.

[67] P. Zhou, F. Qin, W. Liu, Y. Zhou, and J. Torrellas, "iwatcher: Efficient architectural support for software debugging," in *ACM SIGARCH Computer Architecture News*, vol. 32, p. 224, IEEE Computer Society, 2004.

[68] A. Corporations, "Avalon interface specification," *Application note, United States of American*, pp. 11–15, 2006.

[69] Altera, *Avalon Verification IP Suite*, 2016.

[70] T. J. McCabe, "A complexity measure," *IEEE Transactions on software Engineering*, no. 4, pp. 308–320, 1976.

[71] D. Lewis, G. Chiu, J. Chromczak, D. Galloway, B. Gamsa, V. Manohararajah, I. Milton, T. Vanderhoek, and J. Van Dyken, "The stratix 10 highly pipelined fpga architecture," in *Proceedings of the 2016 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, pp. 159–168, ACM, 2016.