EMPOWERING RECONFIGURABLE PLATFORMS FOR MASSIVELY PARALLEL APPLICATIONS

by

Arnab A Purkayastha

A dissertation submitted to the faculty of
The University of North Carolina at Charlotte
in partial fulfillment of the requirements
for the degree of Doctor of Philosophy in
Electrical Engineering

Charlotte

2021

Approved by:

Dr. Hamed Tabkhi

Dr. Arun Ravindran

Dr. Erik Saule

Dr. Nigel Zheng

ABSTRACT

ARNAB A PURKAYASTHA. Empowering reconfigurable platforms for massively parallel applications. (Under the direction of DR. HAMED TABKHI)

The availability of OpenCL for FPGAs along with High-Level Synthesis tools have made them an attractive platform for implementing compute intensive massively parallel applications. FPGAs with their customizable data-path, deep pipelining abilities and enhanced power efficiency features offer the most viable solutions for programming and integrating them with heterogeneous platforms. However, OpenCL for FPGAs raise many design challenges which require an in-depth understanding to better utilize their enormous capabilities. Inefficient routing of data, high number of memory stalls exposed to execution and under-utilization of FPGA resources are significant execution bottlenecks that overshadow the advantages of data-path customization. Furthermore, leveraging OpenCL parallelism abilities and throughput oriented principles is paramount to the success of FPGAs in the high performance computing environment.

In this research we identify, analyze and categorize the architectural differences between the OpenCL parallel programming model and FPGA execution semantic. We propose a generic taxonomy for classifying FPGA parallelism potential to the fullest. To benefit massive thread-level parallelism, we introduce a unique LLVM based automation tool to decouple memory access from computation, thereby hiding memory stalls from the execution path. We further present a novel parallelism granularity that separates kernels to split them into data-path and memory-path (memory read-/write) that work concurrently to overlap the computation of current threads with the memory access of future threads. We validate these principles on the Xilinx based AWS Cloud FPGA platform.

We then conduct a thorough investigation into the scalability of OpenCL coarse-grain parallelism, as well as an examination of Compute Unit(CU) replication, Double Data Rate (DDR) and Burst Transfer (BT) optimizations on Cloud FPGAs. To address the issue of programming challenges, we present generic template(s) and a front end design tool to aid the programmer in rapid exploration and testing. Overall, this dissertation is an amalgamation of principles and techniques to improve the performance and programmability of OpenCL on FPGAs when running massively parallel applications on 'Edge' as well as the 'Cloud'.

DEDICATION

This dissertation is dedicated to my family and friends who have been a constant source of support, encouragement and motivation during my graduate school studies. I feel fortunate enough for having them in my life.

ACKNOWLEDGEMENTS

I'd like to share my heartfelt gratitude to my advisor, Dr. Hamed Tabkhi for his continuous support all through my research. His constant motivation, enthusiasm and immense knowledge of the subject aided in significantly developing my logical and reasoning faculties that are critical for research. If it was not for his continuous support, this work wouldn't have been a reality.

I would like to again offer my heartfelt appreciation to Dr. Hamed Tabkhi, the TeCSAR lab's principal investigator, for providing the critical infrastructure that enabled me to conduct my research. Thanks to the Xilinx University Program (XUP) team at Xilinx Inc. for supporting our work on AWS cloud services. I am grateful to all of my TeCSAR lab colleagues and co-authors who have always supported and assisted me. Special thanks to Samuel Rogers and Mohammadreza Baharani, you guys are the best!

I would like to express a ton of thanks to my commitee members, Dr. Arun Ravindran for his valuable insights on writing this dissertation and Dr. Erik Saule for his ideas on data visualization. A special thanks to Dr. Nigel Zheng who agreed to preside over my doctoral dissertation as an external committee member.

I would at this point like to pay special gratitude to the ECE department of UNC Charlotte, for providing me with the opportunity to pursue my PhD. Thank you for assisting me financially with GASP award and allowing me to use the lab and research facilities in the department.

Finally, I would like to thank my family and friends for cheering me up and standing by me through all times good and bad.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

LIST OF ABBREVIATIONS

ALU	Arithmetic Logic Unit.
API	Applications Programming Interface.
CPU	Central Processing Unit.
CU	Compute Unit.
DP	Data-Path.
FPGA	Field Programmable Gate Array.
GPU	Graphical Processing Unit.
HLS	High-level Synthesis.
HPC	High Performance Computing.
ISA	Instruction Set Architecture.
LLVM	Low Level Virtual Machine.
MCUMDP	Multiple Compute Unit Multiple Data-Path.
MCUSDP	Multiple Compute Unit Single Data-Path.
OpenCL	Open Compute Language.
PE	Processing Elements.
RTL	Resister Transistor Logic.
SCUMDP	Single Compute Unit Multiple Data-Path.
SCUSDP	Single Compute Unit Single Data-Path.
SDK	Software Development Kit.

SIMD	Single Instruction Multiple Data.
SIMT	Single Instruction Multiple Thread.

CHAPTER 1: Introduction

The advent of new technologies like Big Data, Internet of Things (IoT), and Artificial intelligence (AI) have made a significant impact across various sectors ranging from healthcare, transportation, manufacturing, education etc., The combined effect of increased internet usage along with the availability of affordable infrastructure and continuous monitoring across the globe has led to an avalanche of data that needs to be stored, processed and decisioned. Design of powerful and efficient computing engines that can handle large volumes of data is therefore essential.

Instruction Specific Architectures (ISA) such as CPUs have been traditionally employed to deal with this massively increasing workload. However, consistent decrease in transistor sizes have resulted in saturation of Dennard scaling [4] and stagnation of Moore's law [5]. This has lead to a paradigm shift in computing machinery. Modern architectures like Graphics Processing Units (GPUs), Application Specific ICs (ASICs) and Reconfigurable platforms like Field Programmable Gate Array's (FPGAs) are commonplace in today's computing environment. GPUs have thousands of general-purpose cores that can deliver teraflops of performance and work on multitude of tasks at once. Nonetheless, being primarily ISA based, GPUs are power-hungry devices (Figure 1.1). ASICs, on the other hand are power-efficient devices that offers limited design flexibility.

FPGAs have bolstered their position in the high performance computing field by improving performance outside the limits set by slowdown of Moore's Law [5]. Specialized hardware employed to accelerate an application is becoming an essential part for improving application efficiency. FPGAs can provide custom data paths and memory hierarchies meeting the needs of compute intensive workloads, while being

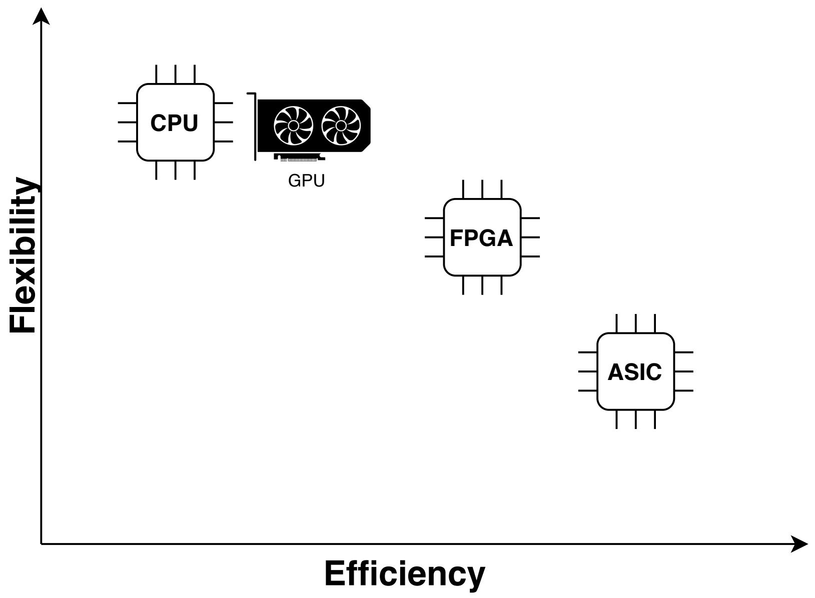


Figure 1.1: Flexibility vs Efficiency of different platforms

power and energy efficient. The demand for FPGAs is thus projected to grow from $63 billion in 2019 to $117.97 billion in 2026 [6] [7].

Conventional FPGA design requires expertise in complex hardware description languages (HDLs) and proprietary synthesis tools that suffer from lack of portability and programming complexity. To counter this shortcoming, the Open Computing Language (OpenCL) was introduced for heterogeneous computing platforms as a vendor neutral programming language. OpenCL for FPGAs offers a wide range of possibilities for increased programmability benefits without getting bogged down by the complexities of HDL programming. The abstraction and convergence of OpenCL [8] provide a novel opportunity for massively parallel applications to be accelerated using FPGAs. Moreover, OpenCL High-Level Synthesis (OpenCL-HLS) allows programmers to create a reconfigurable data path on FPGAs that is ideally suited to the application without going into the low-level implementation details [8].

Despite such promising benefits, the OpenCL language is still in development. In addition to this, both FPGA developors and application programmers face a slew of new design challenges. These difficulties stem primarily from the architectural dif-

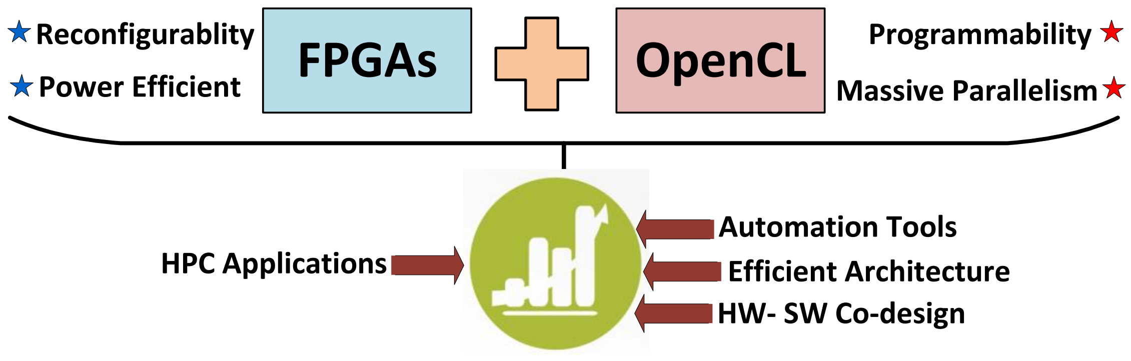


Figure 1.2: Overall philosophy of this dissertation

ferences between GPUs and FPGAs. GPUs are multi-threaded instruction flexible architectures with massive spatial parallelism. Thus, they benefit from the OpenCL Single Instruction Multiple Threads (SIMT) semantic [9, 10]. Unfortunately, the potential and difficulties of OpenCL programming have not been considered extensively for the FPGAs [11, 12, 13, 14, 15, 16, 17, 18]. FPGAs offer customizable data path, operation-level parallelism and the ability to use deep pipelining or temporal parallelism- aspects that are not captured well in OpenCL. Additionally, in the absence of a dedicated run time scheduler or an advanced memory hierarchy engine, execution often stalls due to a lack of availability of data. This results in data-path under utilization for complex OpenCL kernels with high memory access demands.

Furthermore, OpenCL has been highly optimized for GPUs due to their dominant position in the heterogeneous computing industry. Thus there is an insufficient lack of understanding of the impact of design decisions made at the source and synthesis levels for the created data-path on FPGAs. To fully exploit the flexibility of FPGAs, new research is imperative to better understand and formalize the design dimensions while using OpenCL abstractions.

Figure 1.2 shows the overall philosophy of this dissertation. The broad scope of this research is to identify and mitigate several key problems that hinder the maximum performance potential of FPGAs while taking into consideration several limitations in terms of synthesis tools, resources and programming challenges. We combine best-of-both-worlds of both FPGAs and OpenCL programming platform to provide several different solutions ranging from efficient data-path creation to maximizing parallelism, architectural solutions to minimize memory access latency, design automation tools and provide various solutions across the entire computing stack. We have open sourced all our work through different contributions linked to our previous and ongoing publications.

1.1 Contribution and Dissertation Outline

The aim of this dissertation is to explore the execution of massively parallel applications on reconfigurable devices using novel design methodologies. We employ a plethora of techniques to improve the performance and energy efficiency of OpenCL when mapped on to FPGAs (Figure 1.3).

The contributions of this dissertation are summarized as follows.

1. A taxonomy proposed at OpenCL abstraction to improve FPGAs execution efficiency. The taxonomy guides OpenCL programmers and tool developers to achieve maximum throughput by identifying and applying the right granularity of spatial parallelism (Chapter 4)

2. A novel memory decoupling approach at OpenCL abstraction combined with LLVM-based design automation tool to overcome the bottleneck of memory stalls exposed to data-path. We validate and scale our approach on a AWS cloud based FPGA with a new set of applications (Chapter 5) .

3. A generic template for Compute Units (CUs) replication and memory access parallelism combined with an automation tool to enable programmers achieve maximum throughout with respect to available FPGA resources and memory bandwidth (Chapter 6).

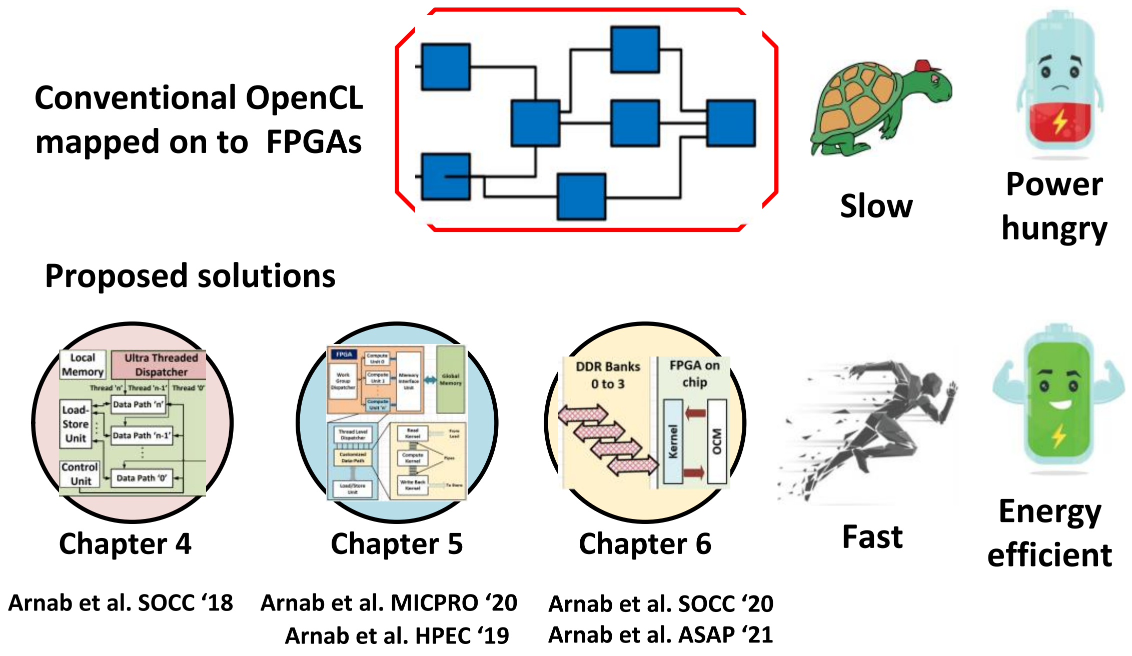


Figure 1.3: Dissertation contributions with publications

1.2 Motivation

"Divide and conquer"

-Gaius Julius Caesar

The basic philosophy of this research is to capitalize upon the numerous potentials of OpenCL programming platform and leverage its parallelism faculties to benefit reconfigurable platforms using their inherent deep-pipeling and power efficient characteristics. We identify several of the critical issues that hinder OpenCL performance when mapping massively parallel applications to FPGAs. We investigate each of the following problems in-depth and systematically present ways and means to counter them.

Motivation for FPGA taxonomy Parallelism principles on CPUs and GPUs [19, 20, 21] has been an active research topic. These techniques, while capable of capturing instruction, data and task level parallelism, were designed for ISA-specific architectures. FPGAs and other reconfigurable systems are ISA-independent platforms, so the same classifications does not apply to them. Recent advancements in OpenCL-HLS tools have provided programmers with new opportunities to work with massively parallel applications and map it on FPGA's. Therefore, it is important that a standardized taxonomy be adopted to fully exploit OpenCL parallelism benefits when mapped to FPGAs.

Motivation for efficient architecture An important observation regarding OpenCL performance on FPGAs vs CPUs and GPUs is the large disparity in performance due to inefficient data path. One of the main drawbacks of fine-level temporal parallelism on FPGAs is memory wall. Memory stalls are explicitly exposed to the execution path since there is no run-time thread scheduling as in GPUs. Additionally, unlike CPUs, FPGAs do not have sophisticated data caching or specialized hardware prefetching to reduce memory access latency. Memory stalls and bandwidth utilization are thus at odds, severely affecting performance. While there are quite a few available techniques like use of scratchpad memory, double buffering etc., to mitigate memory stalls, these methods suffer from additional memory copy time and gross over-allocation of FPGA on-chip memory.

An interesting mechanism to mitigate stalls is *Memory decoupling*, that operates at a finer granularity of data access. By separation of individual memory accesses into different kernels it overlaps memory access and computation. This approach does not have the same ability to amortize memory access costs as tiling and double buffering, but it can be beneficial for certain data that does not follow a streaming access semantic.

Moreover, modern cloud based FPGAs [22] have a sizeable number of on-chip resources and increased bandwidth capacities, allowing many opportunities to boost performance. OpenCL-HLS allows us to explore various optimization techniques that can be employed in consistence with existing methods to exploit FPGA potential to the fullest

Motivation for generic framework and automation tool Replicating Compute Units (CUs) has been found to improve OpenCL performance by increasing the scope of spatial parallelism employed on top of existing temporal parallelism [8, 23]. However, replicating CUs comes with its own set of challenges. Since multiple parameters (such as FPGA tools, workgroup sizing, and additional costs associated with CU setup) are involved, CU replication is an NP Hard problem. This necessitates design space exploration, which can be time consuming and costly in terms of resources, particularly if the workload is run on cloud-based FPGAs. Moreover, multiple CUs do not always improve efficiency because over-splitting the job size results in redundant CU setup costs, which can either stabilize or increase the kernel execution time. Furthermore, there is always the possibility of exhausting FPGA resource utilization and bandwidth. This, in fact, necessitates the obligation of programmers to use a ’trial and error’ approach because it is difficult to make even an informed guess in such situations.

Finally, the outline of this dissertation is as follows.

Chapter 2 introduces the traditional FPGA datapath and gives a brief context of OpenCL programming platform. It overviews OpenCL execution model on FPGAs and discusses the motivation behind each of the core contribution areas as well as current and future research direction.

Chapter 3 briefly reviews previous related work in the field upon which we base our research. We look at several existing approaches of OpenCL parallelism models and provide hints captivating the reader to question and argue the current approaches.

Chapter 4 presents our first contribution of taxonomy of spatial parallelism on FPGAs. We build on our understanding of spatial parallelism as opposed to the default temporal parallelism on FPGAs and formalize these concepts.

Chapter 5 proposes the second contribution of LLVM based memory decoupling approach to hide memory access latency on FPGA devices. We further validate our design principles for Xilinx based AWS cloud FPGAs.

Chapter 6 describes two ideas elaborated in two separate sections. The first section proposes a generic template for Compute Unit (CU) replication and proposes a novel tool to automatically identify optimum number of CUs. The second section explores Double Data Rate (DDR) and Burst transfer optimization (BT) and formalizes both these approaches.

In **Chapter 7** we summarize the contributions of this dissertation, list our publications and talk about future research directions.

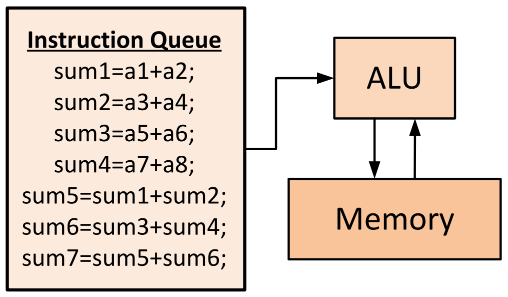
CHAPTER 2: Background

In this chapter, we look at the trends in programming models starting from traditional CPUs introduced in the early 60's to modern multicore devices. We understand the basic architectural differences between general purpose CPUs, multicore devices and reconfigurable platforms. We next move on to see the OpenCL programming model and its execution semantic when mapped onto FPGAs.

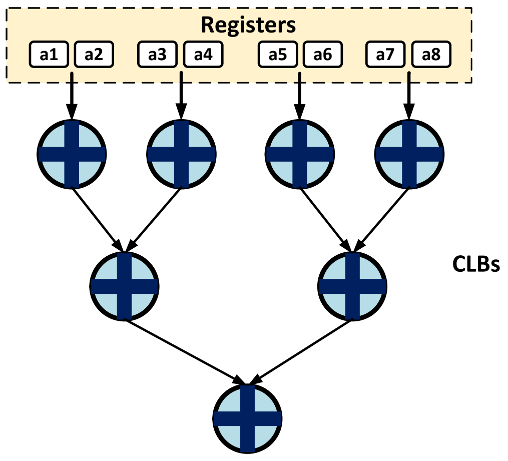
2.1 Current Trends in Programming Technology

Programming technology in the initial computing era was driven majorly by two popular approaches, the single core CPU and fine-grained array architectures. The general-purpose CPU on one end is an instruction specific architecture where programmable software instructions are sequentially executed (traditionally), while on the other end, fine-grained architectures like FPGA are reconfigurable hardware devices that can execute instructions in parallel using programmable Hardware Description Languages (HDLs). Figure 2.1 shows the basic architectural difference of CPUs and FPGAs. CPUs benefit from the inherent sequential data-path while FPGAs benefit from the customizable hardware.

Consider a sequence of instructions *sum1* through *sum7* as shown in Figure 2.1a. The instructions are stored in the instruction queue before being moved sequentially to the ALU via functional units for execution. The resulting instruction data-path is therefore a collection of ALUs, registers, buses and control units inside the CPU. The main components of an FPGA are Configurable Logic Blocks (CLBs), Programmable Interconnects and Programmable I/O blocks. These logic blocks can be configured internally to generate the desired data-path to result into the output logic (Figure



(a) CPU execution semantic



(b) FPGA execution semantic

Figure 2.1: CPU and FPGA execution semantic

2.1b). In an ideal case with limitless registers, parallelism on FPGA can therefore be infinite. CPU and FPGA architectures have thus coexisted in the two extremes for several years, with each form of programmability being extended to various application domains.

Increased demands in computation and performance have steered CPU research in two directions; *one*- increasing the operating frequency with supply voltage and power density trade off, and *two*-extracting Instruction Level-Parallelism (ILP) from the sequential execution semantic. Both these approaches have provided enormous benefits for several decades. However by making device dimensions smaller and smaller, we have reached the "Power Wall" [24], beyond which it is not feasible to reduce the operating frequency anymore. Moreover, approaches to extracting ILP by various techniques like memory prefetching have slowly started to show diminishing returns after several years of performance improvements thus hitting the "Memory Wall" [25].

Due to the saturation of the previous two approaches, current CPU technologies have moved from instruction level parallelism to thread level parallelism. There is a general trend toward highly parallel multicore CPUs (8, 16 or 32 cores) with a

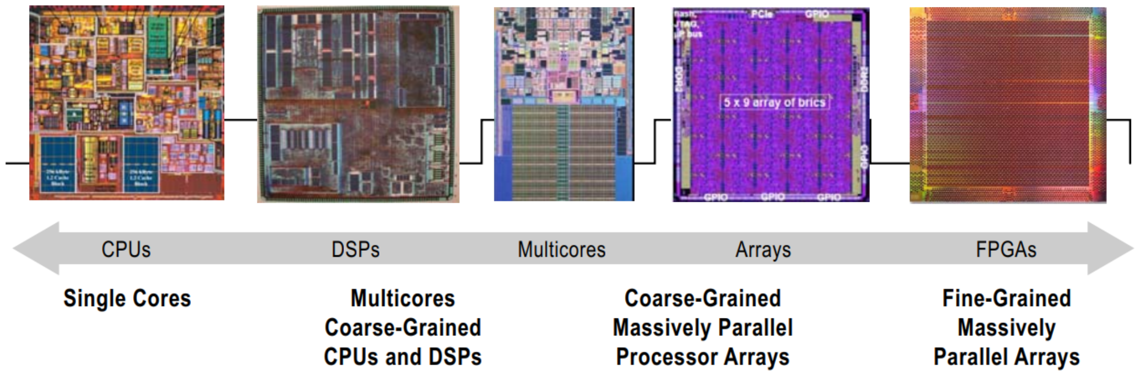

CPUs
DSPs
Multicores
Arrays
FPGAs
Single Cores
Multicores Coarse-Grained CPUs and DSPs
Coarse-Grained Massively Parallel Processor Arrays
Fine-Grained Massively Parallel Arrays
5 x 9 array of brics

Figure 2.2: Recent trends in programming models[1]

focus on several simpler processors with significant number of transistors dedicated to computation. Following the same lines, data-parallel devices like GPUs were designed with hundreds to thousands of simple cores. In any case, the programmer must efficiently code their applications to extract parallelism and achieve high performance on these multicore machines. Each core must be given work in such a way that all cores will work together to complete a task. FPGA designers do the same thing when creating high-level system architectures. Latest technology scaling patterns and increased computational demands have thus led to favourable technologies that are both programmable and parallel (Figure 2.2).

2.2 The OpenCL Programming Language

Heterogeneous computing needs across various architectural frameworks have led to the rise of efficient programming technologies. However, one of the main conflicts between technology and users is the differences between components that don't match well with others, resulting in integration issues and the need for complicated, time-consuming solutions. As an example, popular programming language like CUDA is vendor specific only for NVIDIA GPUs making it hard to translate and fit well with the AMD GPU platform. Therefore, it was realized that a standardized model for writing programs that can run across all of these platforms is essential. Thus the OpenCL programming language was born as a close collaboration of Khronos Group and Apple Inc.,

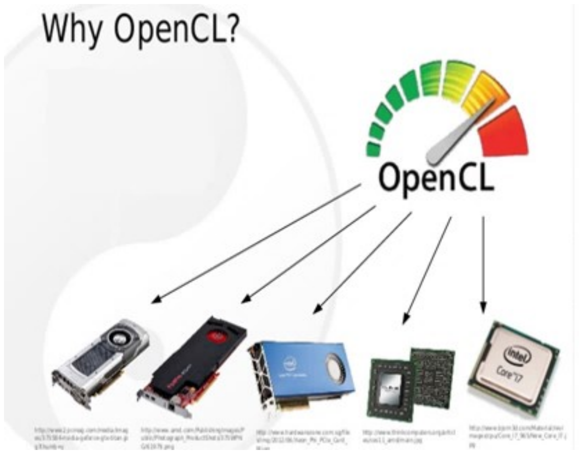


Figure 2.3: OpenCL programming models [2]

The Open Computing Language (OpenCL) (Figure 2.3) is a heterogeneous programming platform for developing applications that run on a variety of vendor-neutral devices ([26, 27]). It is a cross-platform API that enables the creation of portable parallel applications approaching dynamic memory hierarchies and data-parallel execution in the same way as CUDA does. However, unlike CUDA, OpenCL has a more complex system management model to support its multiplatform and multivendor portability.

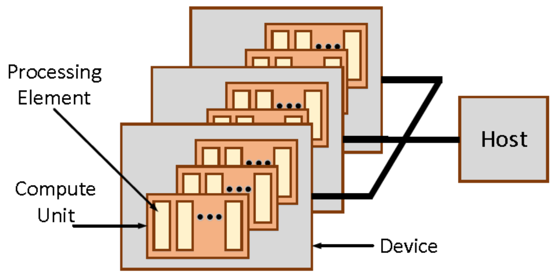


Figure 2.4: OpenCL Platform Model

OpenCL offers a very promising lexical semantic to work with massive threads in parallel, especially when a fixed routine over large volumes of data is executed per each thread. It embraces a broad variety of parallel levels at a higher level of abstraction

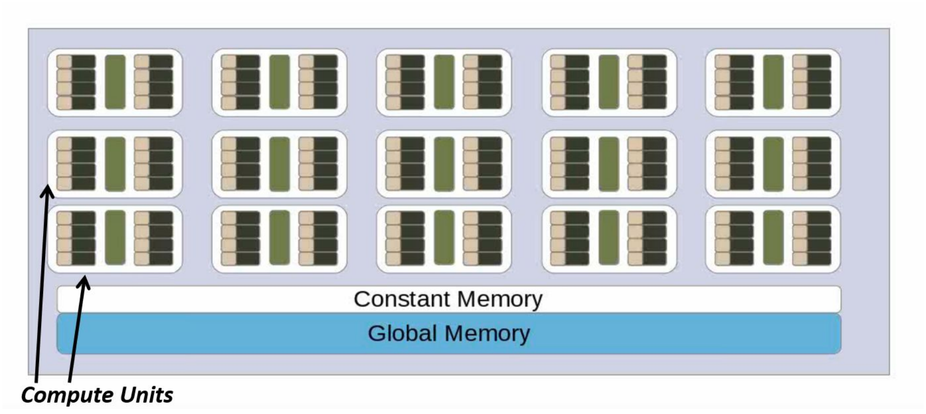


Figure 2.5: OpenCL Device Model

than HDLs and maps heterogeneous networks with CPUs, GPUs, FPGAs and others.

The OpenCL platform model (Figure 2.4) includes a host processor that coordinates program execution and one or more accelerators that can execute OpenCL code (Figure 2.4) (called kernel). The host is typically an x86 CPU that runs the serial portions of the program with GPUs or FPGAs that run the parallel part. The host is also in charge of configuring the systems and overseeing host-to-device and device-to-host connectivity.

Figure 2.5 shows the internal architecture of OpenCL device. An OpenCL device can be a CPU, GPU or an FPGA. Any such device consists of Compute Units (CU) that the main processing blocks similar to execution units and ALUs on multi-core CPUs or compute cores in GPU. Each Compute Unit internally contains Process Elements (PE) with its own private memory as shown in Figure 2.6. The PEs are where a device's computations take place. CUs are generally called as 'work-group' (a.k.a Block in CUDA) while PEs are called as 'work-item' (a.k.a Thread in CUDA) which execute the flow of instructions.

The 3 major memory types in OpenCL are the Global memory, Local memory and Constant memory.

Global memory region allows read/write access to all work-items in all work-groups on any computer in a context. Any part of a memory object can be read from or

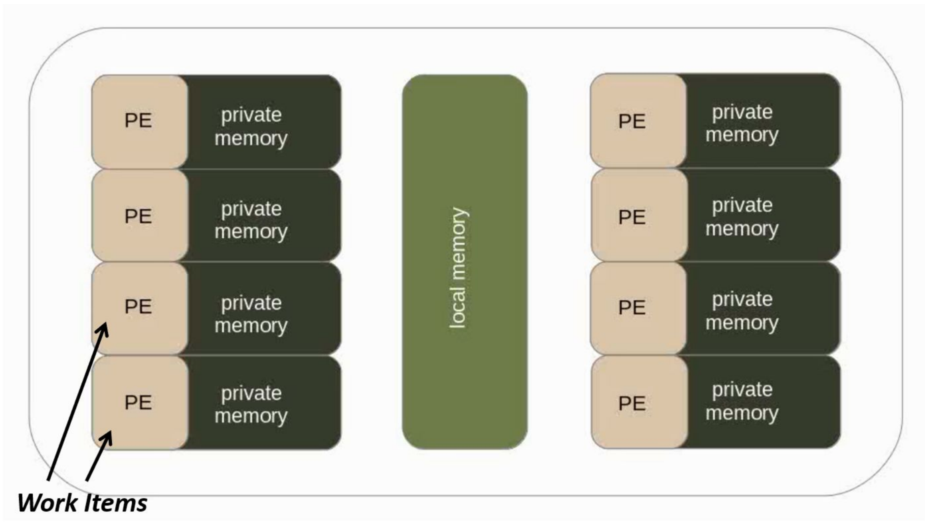


Figure 2.6: OpenCL Compute Unit model

written to by work-items. Depending on the device's capabilities, reads and writes to global memory can be cached.

During the execution of a kernel-instance, an area of global memory that remains constant is called the *Constant memory*. Memory objects stored in Constant memory are allocated and initialized by the host.

Finally, *Local memory* is a Work-group specific memory area. This memory region can be used to store variables that are shared by all work-items in a work-group. Local memory is thus shared across multiple process elements within a compute unit.

2.3 OpenCL Execution on FPGAs

In this section, we look at the OpenCL abstraction and execution models when mapped on to FPGA. OpenCL execution utilizes High Level Synthesis (HLS) tools (vendor-specific) to generate the desired data-path on the FPGA. The HLS toolchain is the common link between the high level OpenCL language to the low level bit-stream. Figure 2.7 shows a generalized HLS toolflow.

The programming model of OpenCL is based on the C programming language. It provides an API that allows host-based programs to load the OpenCL kernel on

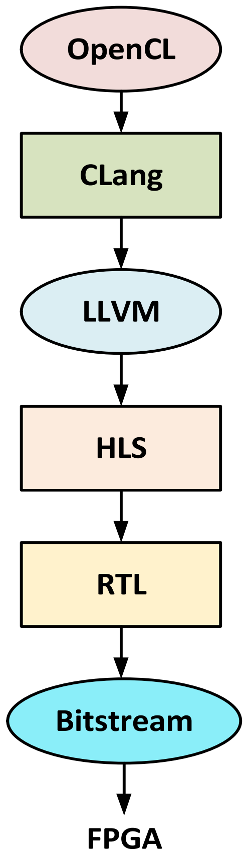


Figure 2.7: OpenCL HLS toolflow

computing devices. The API is also used to manage system memory independent of host memory and interact directly with GPU resources. The host and kernel code written in OpenCL are packed together and fed to CLang compiler that converts high-level OpenCL code to Low Level Virtual Machine (LLVM) which is an Intermediate Representation (IR) of the instructions fed to CLang. This IR is then fed to the HLS toolchain that generates the RTL (Register Transfer Logic) down to the programmable bitstream. The bitstream is mapped to the FPGA in the form of programmable logic that is customizable in nature and can be executed. This entire process is called "Synthesis".

Next, we look at the OpenCL execution model as depicted in Figure 2.8. In OpenCL, a work-item is the unit of parallelism inside each kernel. The same ker-

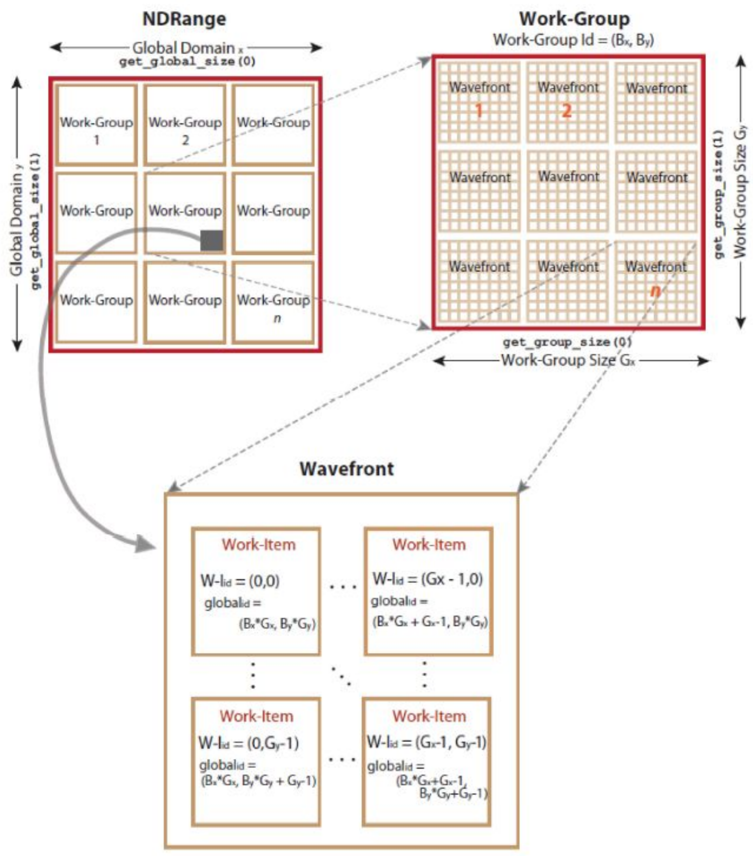


Figure 2.8: OpenCL Execution Model

nel is used for all OpenCL work objects, but the data is different. An 'NDRange' (a.k.a Grid in CUDA) is the total number of work-items executing kernel code as specified by the programmer in the host code. The NDRange is a work-item index space with N dimensions of one, two, or three. The NDRange is divided into work-groups, each of which includes several work-items, as seen in Figure 2.8.The compiler defines the NDRange size (also known as global size) and work-group size (also known as local size) in the host code. A wave-front is a set of work objects that run at the same time on the system. The system vendor determines the wave-front distance, which is architecture-dependent.

2.4 Maximizing the Parallelism on FPGAs

Parallelism concepts on CPUs and GPUs have been extensively studied before [19, 20, 21]. While these techniques are able to capture instruction and data level parallelism, they were developed for ISA specific architectures. Reconfigurable platforms like FPGAs are ISA independent architectures and the same classifications cannot be justifiably applied on them. Moreover, in contrast to GPUs that have massively parallel fixed ALU cores, an FPGA's reconfigurable design allows for the creation of a customized data-path suited for each application. Thus by eliminating instruction fetch and streamlined thread execution, a custom data-path can be deeply pipelined to improve throughput. Deep pipelining also allows FPGAs to take advantage of temporal parallelism through a large number of hardware threads while sharing a single data path.

Recent improvements in OpenCL-HLS have opened up new opportunities for the FPGA programmers to work with massively parallel applications on FPGAs and utilize the capabilities of the FPGAs to their full potential. An application developed at OpenCL abstraction can guide the synthesis tools by explicitly exposing the parallelism. While there are quite a few techniques developed to aid the programmer in this field, what is missing is the lack of insight on the impact of source level design decisions on the generated data-path on FPGAs. In particular, spatial parallelism when mapping massively parallel applications on FPGAs has not been well understood and formalized. To fully leverage the benefit of FPGA's reconfigurability, novel research is required to understand and formalize the design dimensions when running OpenCL abstraction on FPGAs.

2.5 Memory Bottleneck

One of the primary limitations of fine-level temporal parallelism on FPGAs is the issue of *Memory Wall*. In the absence of a dedicated run-time scheduler like in

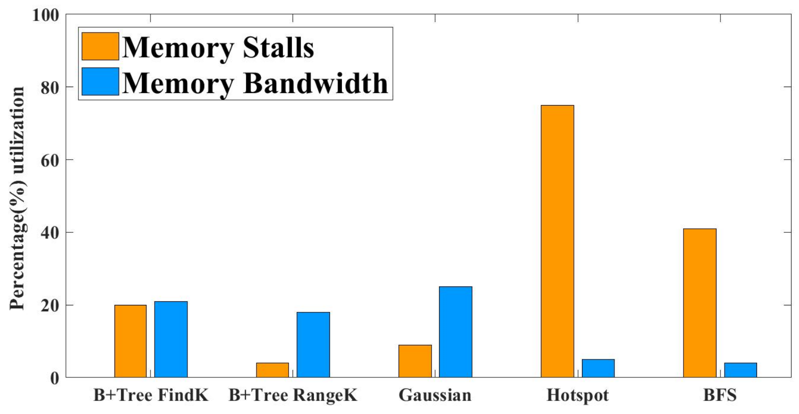


Figure 2.9: Memory stalls and bandwidth utilization for Rodinia applications

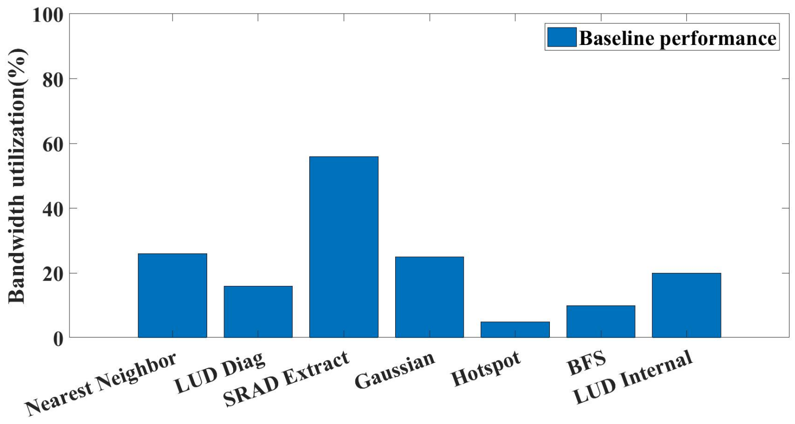


Figure 2.10: Baseline bandwidth utilization(%) for Rodinia applications

GPUs, or a sophisticated memory hierarchy (multiple levels of cache), execution on FPGAs is often stalled due to a lack of data. As a result, if one thread is waiting for memory, all other threads will be stalled before the current thread receives the data. This results in data-path under-utilization in complex OpenCL kernels with strong memory access demands. Memory stalls is also therefore a major cause for under-utilization of memory bandwidth. To illustrate this we look at baseline profiling information of some of the applications from the Rodinia Benchmark Suite [11] as shown in Figure 2.9. For 2 cases, namely BFS (40% memory stalls) and Hotspot(75% memory stalls)leads to massive under- utilization of available memory bandwidth (4% in BFS and 5% in Hotspot).

Further to this, figures 2.10 and 2.11 show the low baseline bandwidth utilization

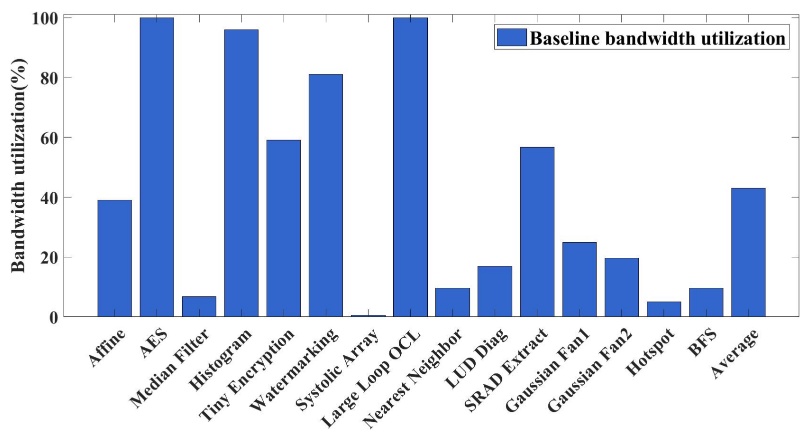


Figure 2.11: Baseline bandwidth utilization(%) for SDAccel applications

for a set of applications from the Rodinia Benchmark suite and the SDAccel design suite. Average bandwidth utilization for the applications are less than 25% and 50% respectively. This significantly limits their achievable performance while opening new opportunities for optimizations at the same time. As a result, for effective execution of massively parallel applications on FPGAs, reducing the memory bottleneck is crucial.

2.6 Programmability Challenges and Design Time Complexity

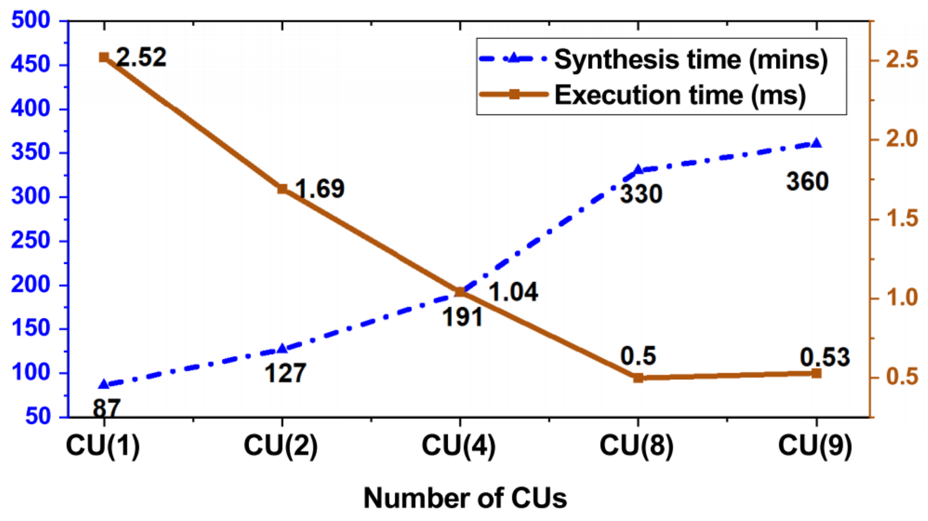


Figure 2.12: DSE of Histogram application.

Many new parallelism possibilities and optimizations are available with OpenCL. However, applying these methods necessitates a thorough understanding of the device's design, and programmers often settle for a "trial and error" solution, which is cumbersome and may not yield maximum parallelism benefits. This is partly due to the time-consuming nature of extensive design space exploration, and partly due to the lack of generic programming constructs to serve as a reference for FPGA programmers. Apart from that, the long synthesis times make the process of development to deployment very tedious. Figure 2.12 illustrates this problem for a demonstrative *Histogram* application. The Design Space Exploration time of *Histogram* for Compute Unite replication (More on that in Chapter 4) is close to 347 mins! This costs a lot of valuable time and resources and is therefore a long standing issue with parallel programming on FPGAs hindering performance efficiency. On the flip side, it also makes way for tremendous research opportunities in this field.

CHAPTER 3: Related Work

The introduction of OpenCL for FPGAs combined with numerous capabilities of HLS tools, has sparked a tremendous amount of interest and influenced a large number of researches in this field. OpenCL for FPGAs has benefited a wide range of high-performance massively parallel applications. Furthermore, OpenCL abstraction and unification has the potential for widespread deployment of FPGAs to revolutionize them as an omnipresent component of heterogeneous platforms. Overall, despite numerous studies in the field, OpenCL for FPGAs is still in its infancy. In-depth analysis and generalized solutions to improve OpenCL execution performance on FPGA devices are lacking. The main emphasis has been on building an effective application-specific data-path rather than eliminating critical bottlenecks like memory latency, optimum resource utilization etc., In this chapter we look across the spectrum of research that has already been conducted upon which we lay the foundation of this dissertation.

3.1 OpenCL Parallelism Across Various Architectures

The advent of OpenCL framework has become a very popular topic of interest in the high-performance community. Performance optimization using OpenCL framework for massively parallel applications on CPUs/GPUs has also become very popular [28, 29, 12]. Execution of OpenCL on FPGAs has become a prominent topic [12, 13, 30, 14, 31, 17, 23] in recent times. These approaches primarily focus on the application-specific performance optimization techniques [13, 30], or basically making a performance comparison between FPGAs and GPUs.

Many researches [31, 17, 32] have been conducted on OpenCL programming capabilities to improve FPGAs efficiency. The energy efficiency benefits of FPGA devices along with the ability to employ pipelined parallelism properties make the evaluation of OpenCL kernels on these devices very fascinating [13, 14]. Further, [33, 34, 35] have worked on exploiting the parallelism on FPGAs with OpenCL attributes as well as by suggesting new architectural modifications to improve performance across varied applications. In particular, we observe a significant interest in accelerating neural networks and deep learning applications on FPGAs using OpenCL programming abstraction [36, 37, 38, 39, 40].

The subject of multi-threaded execution on FPGAs has also been explored [41] [42, 43, 44, 45, 46]. A small number of researchers have also looked into multi-threaded execution on single kernels [47, 48]. A framework was proposed to explore parallelization of instructions and data on ISA-based architectures, that is, CPUs and GPUs [19] by Michael J Flynn during the early 70's. Many other works [20, 21] have been studied to provide an extension over Flynn's taxonomy for different multiprocessor architectures.

In contrast to existing categories for ISA based machines, the execution model of FPGAs make them different which leads to a dire need for a classification of their own. OpenCL-HLS gives the programmer such flexibility by introducing various optimization techniques. FPGA design space is not yet well investigated and systematized. This applies in particular to Compute Unit replication with enormous potential performance. It is therefore important not only to design scalable accelerators but to design tools that can reduce synthesis design times significantly, which make them cost-effective infrastructures.

3.2 Memory Wall

Memory stalls as a result of unavailability of data for computation is a major bottleneck as the cost of data transfer from memory is very expensive. The most intuitive solution to this problem is to make data available prior to execution. The simplest approach to furnish this data is called as prefetching [49, 50]. As the name suggests, prefetching is to understand the behaviour and keep the data available for the device prior to computation. There has been a lot of prior research done in this field. Based on the behavior prefetching can be acheived by hardware called as hardware prefetching [51, 52, 53] or software called as software prefetching [54, 55, 56]. There have been relevant experiments done to merge both hardware and software prefetching to achieve maximum performance [57]. Also recently, the focus has shifted towards various cache prefetching approaches [58, 59].

In the reconfigurable computing community, multi-thread execution on FPGAs is a very rich topic [41, 60, 42, 43, 44, 45, 61, 62, 46]. The primary focus is on context switching and partial reconfigurability across multiple applications. However, there is a less focus on addressing the execution challenges of massively parallel applications on FPGAs when many threads are sharing same kernel (data-path) over many data (as in OpenCL). The approach of decoupling has been well elaborated [63] and has been explored and analyzed more on different variety of devices [64, 65, 66]. Memory decoupling has made a significant contribution to our understanding of memory access latency in different system models.

Several other papers, for example [67], introduce a novel Domain-Specific DSE (DS-DSE) approach for domain-specific computing with an emphasis on streaming applications, as well as Function-Level Processors (FLPs) [68] to bridge the gap between ILPs and dedicated hardware accelerators. While performance gains are demonstrated, these approaches often do not explore the static analysis of application for memory access behavior or are restricted to well defined accesses with fixed strides. with no massive thread-level parallelism.

Finally, we also note the interest in using OpenCL pipe semantics for efficient communication between kernels across several OpenCL kernels [35, 18, 34, 69]. [69], for example have used pipes for efficient DNA and RNA sequencing on FPGAs.

Elsewhere [33], proposed analytical models for power and performance evaluation and optimization of OpenCL kernels on FPGAs. Xilinx in particular, have also taken advantage of the pipe build to stream data from one kernel to another and thus reduce latency [22]. These serve as a backbone to our decoupling approach to mitigate the memory wall problem.

3.3 OpenCL Optimizations on Cloud FPGAs

Designers have used FPGAs for the past 30 years as a quick and easy way to create specialized hardware for applications that need more performance than software-programmed CPUs can provide. In recent years, we observe integration of FPGAs in many cloud platforms ranging from AWS cloud [70] to Microsoft Brainwave project focused on real-time AI [71] to Xilinx Zynq platforms [72, 73] for real-time stream processing at the edge. Today's cloud-based FPGA instances, also known as FPGA-based acceleration-as-a-service, enable users to rent time on an FPGA-equipped server rather than buying a board or an integrated server.

The two major players in the FPGA market viz., Intel and Altera have both improved their High Level Synthesis (HLS) tool-chains to suit a variety of scalable options on the cloud. As a result a number of optimization tools have sprung up from both these vendors. Compute Unit replication is a well know optimization [23, 74, 75] that has shown significant performance benefits although limited due to lack of systematic approach. Other optimizations like Caching/Tiling, Customized Pipeline, and Double Buffering [76] do provide benefits along with programmability challenges. Furthermore, techniques like explicit data caching, communication overlap, and scratchpad organization have also shown to improve the performance [75]. Optimizations such as adjusting the work-group, CU replication and memory coalescing were performed to get 1.5X improvement on two dimensional Mandelbrot factorial algorithm [77] employed on the cloud.

Even though current accelerators outperform generic processors, the FPGA design space has yet to be fully explored and systematized. This is particularly true for CU replication, which has a lot of potential in terms of efficiency. As a result, it's critical to develop not only scalable accelerators but also tools that can dramatically reduce synthesis design time while remaining economically viable across all infrastructure.

Overall, despite many interesting researches in the area, OpenCL for FPGAs is still at its early stages. There is a visible lack of in-depth analysis, generalized tools, frameworks and solutions to enhance the OpenCL execution efficiency on FPGA devices. Here in the course of this entire dissertation, we propose a plethora of generic design solutions. We identify, analyze and mitigate various bottlenecks affecting FPGA performance. Finally, we propose new automation tools to reduce the gap between programming challenges and performance efficiency.

CHAPTER 4: OpenCL Parallelism Taxonomy

4.1 Introduction

In this chapter, we begin our exploration of OpenCL spatial parallelism when mapping OpenCL kernels to FPGAs. This work presents a systematic study to define, analyze, and categorize the spatial parallelism. We investigate the effects of Data-Path (DP) and Compute-Unit (CU) replication of OpenCL execution on FPGAs. We further propose a generic taxonomy for classifying spatial parallelism to be applied across any application. We have utilized the Intel Altera Stratix V FPGA of the DE5 family to carry out the experiments in this part of the thesis. Our findings on eight applications from the Rodinia benchmark suite show that FPGA-aware OpenCL codes boost performance by 3.4X, 2.2X, and 2.6X on average for SCUMDP, MC-SDP, and MCUMDP versions over SCUSDP as the baseline implementation.

4.2 Taxonomy Grid

Primary motivation behind this work lies in identifying and classifying the OpenCL HLS programming techniques and mapping them into a framework which we believe could provide better programmability to FPGA programmers and help formalize OpenCL research in near future. To identify all possible spatial parallelism benefits and maximize parallelism potentials on FPGAs, we propose a taxonomy that provides a clear classification based on the type of parallelism that can be employed on the FPGAs. Our proposed taxonomy is classified into four categories. Figure 4.1 shows the grid depicting our taxonomy. OpenCL work-groups typically get mapped to compute unit while work-items get mapped to data-path. We show one through many compute units on the X-axis of the grid while variation of data-paths are shown on the Y-axis. Figure 4.2 through 4.5 shows all possible classifications of OpenCL parallelism that we have proposed in our work.

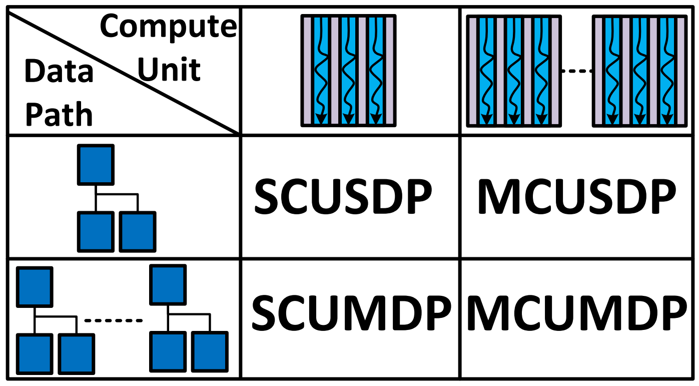


Figure 4.1: Taxonomy Grid

4.2.1 Single Compute Unit Single Data-Path

The Single Compute Unit Single Data-Path [SCUSDP] is the default synthesis generated by the HLS tool. We introduce this as a starting point for comparison with other categories. The SCUSDP which has its own memory, load-store and control units. SCUSDP does not utilize any spatial parallelism capability when implemented on the FPGA although the data-path generated across the compute unit does enjoy temporal pipelining benefits.

4.2.2 Single Compute Unit Multiple Data-Path

Figure 4.3 is a finer grained classification that involves replicating data-paths inside a single compute unit without replicating the thread dispatcher and the load/store units. The CU is able to run multiple threads at the same time by replicating the data-path. SCUMDP resembles the Single Instruction Multiple Threads (SIMT) paradigm used in GPUs in terms of semantics. On top of the temporal parallelism, this allows for spatial parallelism. Furthermore, since each compute unit is involved, each CU has multiple ALUs to execute the same instruction through multiple threads and data while sharing the same control signals.

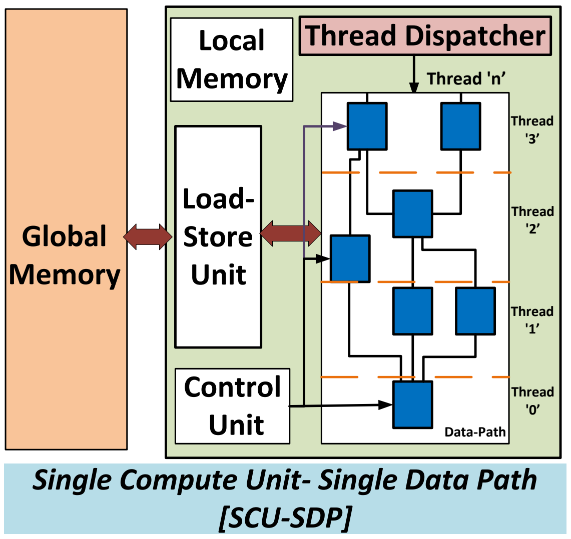


Figure 4.2: Single Compute Unit Single Data-Path

One disadvantage of SCUMDP is that it executes in lock-step between replicated data-paths, which may cause execution stalls due to a lack of data. Due to the fact that repeated data paths share the same control signals, they must run in synchronous lock-step mode. This necessitates the availability of data for all threads; otherwise, all threads would be stalled. Overall efficiency enhancement is hampered as a result of this.

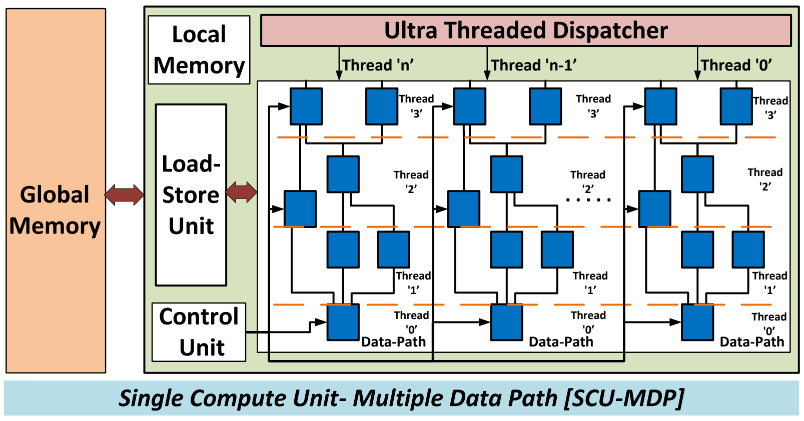


Figure 4.3: Single Compute Unit Multiple Data-Path

4.2.3 Multiple Compute Unit Single Data-Path

The Multiple Compute Unit Single Data-Path [MCUSDP] (Figure 4.4) is a coarser level granularity of implementing spatial parallelism over temporal parallelism. This method replicates the entire data path, thread dispatcher, and load/store units of a CU. The dispatcher divides the workload among several CUs, with each CU handling a group of threads. As a result, control signals within each CU are independent of one another. Employing multiple compute units however can lead to contention for global memory which in turn might lead to undesired memory access patterns affecting performance. This can be attributed to the fact that vectorizing a kernel gives an opportunity to the HLS tool to apply memory coalescing [78].

Further, due to the limitations imposed on FPGA resources, MCUSDP is not always feasible for complex kernels with large code sizes. In terms of resource consumption, MCUSDP is less effective than SCUMDP. Furthermore, the most significant disadvantage of MCUSDP is the added memory burden on off-chip memory. Increased off-chip memory accesses degrade performance in memory-bound kernels due to CU contention over the device's limited memory bandwidth.

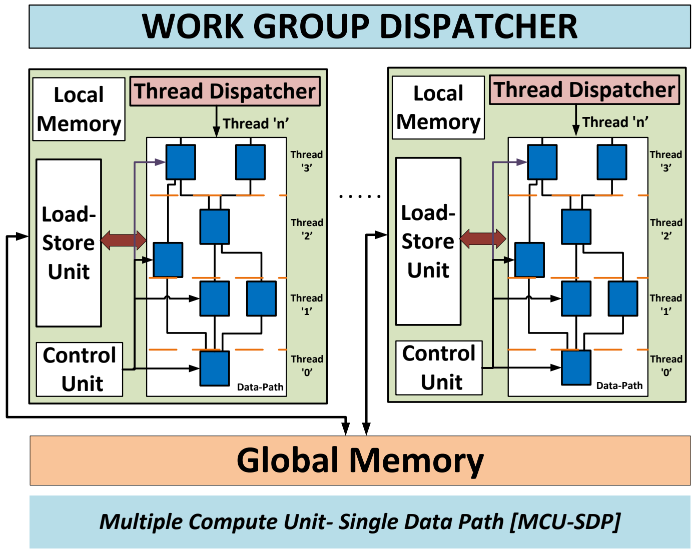


Figure 4.4: Multiple Compute Unit Single Data-Path

4.2.4 Multiple Compute Unit Multiple Data-Path

Another method to maximize spatial parallelism benefits is the use of total spatial parallelism across all data paths. This concept is utilized in Multiple Compute Unit Multiple Data-Path [MCUMDP] (Figure 4.5) which is a hybrid model developed from the previous two classifications. This technique not only makes use of pipelining potentials but also exploits massive parallelism across each compute unit. This is the maximum parallelism potential that can be exploited on the FPGAs. MCUMDP strives to offer a balance between the major bottlenecks of resource utilization and memory contention in MCUSDP along with stalls observed due to lock step execution in SCUMDP.

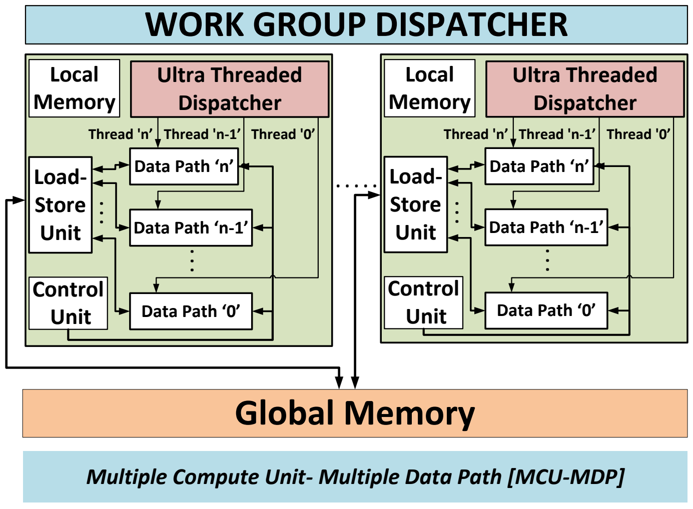


Figure 4.5: Multiple Compute Unit Multiple Data-Path

4.3 Experiments

To evaluate the taxonomy benefits and validate them we employ eight standard OpenCL applications from the Rodinia benchmarks suite [31]. They namely are Nearest Neighbors, Srad_base(Srad extract application), Gaussian, B+Tree, Needleman Wunsch(NW), Breadth First Search(BFS), Hotspot and Stream cluster. Our FPGA implementations are synthesized on the Stratix-V FPGA while we have used the AMD Firepro W7100 device for our GPU implementation. Table 6.7 lists the

Table 4.1: System characteristics employed for the study.

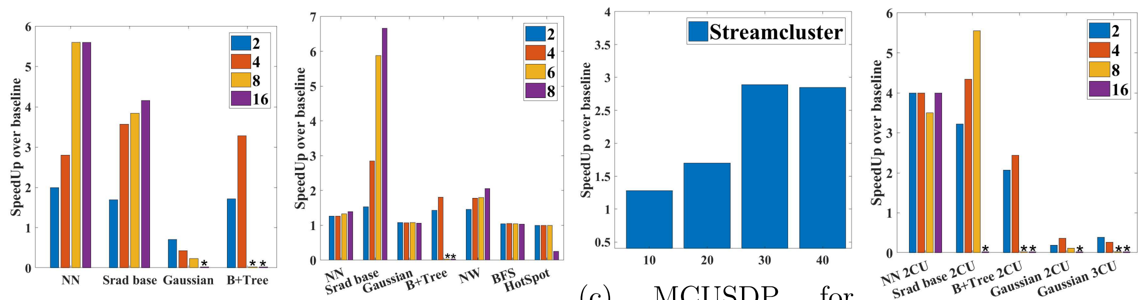
Host	Intel(R) Core(TM) i7-7700K
Host clock	4.2 GHz
FPGA Family	Stratix-V
FPGA Device	5SGXMA7H2FE35C2
CLBs	234,720
Registers	939K
Block Memory bits	52,428,800
DSP Blocks	256
GPU Device	AMD Firepro W7100

parameters of our FPGA platform. We use Intel SDK for OpenCL [8] based off on OpenCL version 1.0 for compiling and synthesizing OpenCL code.

We use eight massively parallel OpenCL applications from the Rodinia benchmarks suite [31] to test and verify the taxonomic benefits. Nearest Neighbors, Srad base(Srad extract application), Gaussian, B+Tree, Needleman Wunsch(NW), Breadth First Search(BFS), Hotspot, and Stream cluster are the chosen candidates. We synthesized all the applications on the Stratix-V FPGA, and our GPU implementation was created on the AMD Firepro W7100. Our FPGA platform's parameters are described in Table 6.7. For compiling and synthesizing OpenCL code, we use Intel SDK for OpenCL [8], which is built on OpenCL version 1.0. The Intel SDK-OpenCL(AOCL) profiler is also used to extract accurate execution performance. The AOCL profiler collects kernel output statistics, global memory bandwidth efficiency, and stalls (Percentage time that memory access caused the stall in kernel execution).

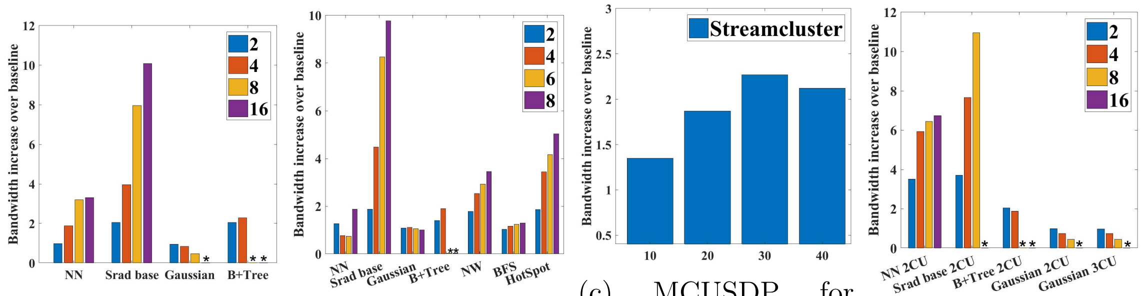
4.4 Discussion

In this section we present and evaluate our experimental results. At first, we compare FPGA taxonomy classifications based on Speedup, Bandwidth utilization, and Absolute stalls parameters with respect to the baseline numbers. Following that, we compare the best FPGA performance obtained from taxonomy against GPU execution.



(a) SCUMDP (b) MCUSDP (c) MCUSDP for Stream-Cluster (d) MCUMDP

Figure 4.6: Performance Improvements over Single compute-unit Single data-path



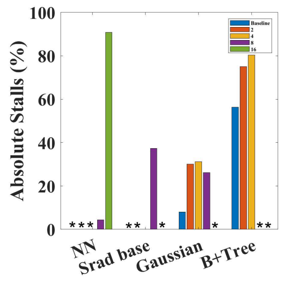
(a) SCUMDP (b) MCUSDP (c) MCUSDP for Stream-Cluster (d) MCUMDP

Figure 4.7: Bandwidth Variations

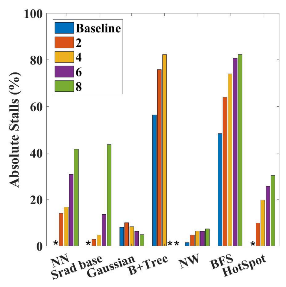
4.4.0.1 Performance Analysis

In our experiments we used the OpenCL attributes to set the number of compute units and data-paths along with sizing the work group dimensions for the purpose of our design. For the SCUSDP and the MCUSDP configurations we were able to synthesize all eight selected applications. On the other hand for SCUMDP and MCUMDP classifications we could synthesize only four applications namely, NN, Srad_base, Gaussian and B+Tree. The use of several data-paths, which is common to each of these types, has a drawback of requiring a lock-step execution pattern, which restricts data-path replication to basic kernels with no data based or conditional branches. The OpenCL-HLS tool fails to vectorize such applications.

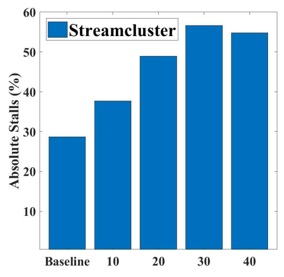
Figure 4.6a through Figure 4.6d show the performance improvement(increase in times speedup over baseline) for each of the taxonomy categories for every single application. Legends of SCUMDP and MCUMDP indicate the number of data-paths



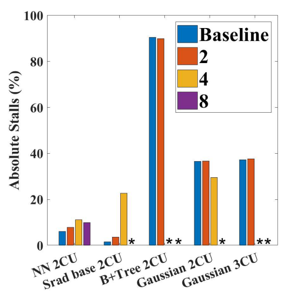
(a) SCUMDP



(b) MCUSDP



(c) MCUSDP for Stream-Cluster



(d) MCUMDP

Figure 4.8: Absolute Stalls

from '2' through '16' while that of MCUSDP indicates number of compute units from '2' through '8'. The graphs for compute units and data-paths labels marked 'asterisk(*)' are the ones which could not be compiled due to FPGA resources reaching its maximum possible capacity.

Speed up due to SCUMDP can be attributed to two basic reasons:- 1. Vectorizing the kernel allows the creation of multiple data-paths that can execute in a single instruction multiple thread (SIMT)fashion. 2. It also avails an opportunity for the HLS tool to introduce memory coalescing to further increase efficiency. On the other hand performance either saturates or decreases owing to the increase in number of stalls with every additional data-path due to lock step execution model. Apart from this, the application itself can affect the parallelism potential.

SCUMDP in Figure 4.6a shows a maximum speed up of 5.6X over baseline for nearest neighbor application when employing '8' data-paths while saturating for '16' data-paths. The speed up for Srad_base and B+Tree show similar trends as well. The Gaussian application on the contrary shows reducing trends, this is due to the fact that low temporal locality hinders its ability to take full advantage of memory coalescing thereby reducing its overall performance despite employing data-path parallelism. Figure 4.7a and Figure 4.8a show a similar correlation between the results obtained. While bandwidth utilization tends to improve with increasing number of data-paths(maximum being at 10.2X for Srad base with 16 data-paths) the effect is more or less limited to increase in stalls due to lock step execution observed in SCUMDP.

MCUSDP in Figure 4.6b shows a maximum speed up of 6.7X over baseline for srad_base application after replicating '8' compute units. The speed up however isn't that effective for all other applications maintaining an average of about 1.5X speed up. Overall more resource utilization in MCUSDP accounts for more memory contention leading to reduced performance for most of the benchmarks. Figure 4.7b and Figure 4.8b provide similar characteristics with increasing number of compute units however with more number of stall percentage compared to SCUMDP attributed to memory contention.

The stream cluster application for MCUSDP in Figure 4.6c offered negligible improvements at low number of compute units while showing a considerable improvement only on increasing the number of CUs from 10 through 40 while achieving a maximum speed up of 2.89X over baseline before getting limited by resources.

Figure 4.6d shows the performance improvement of MCUMDP. In this case we could experiment with two compute units and a maximum data-path of four. Anything over this was not compilable due to the major bottleneck of FPGA resources. However, the performance improvement observed is once again a trade-off between stalls (Figure 4.8d) due to memory contention(MCUSDP) and lockstep execution(SCUMDP). We attained a maximum speed up of 5.6x for the srad_base application using this approach.

The following two-fold approach was used to quantify and compare the performance of FPGA and GPU in terms of normalized performance/watt.

1. We used the PIN Tracer tool[79] to find the total number of instructions, memory and branch accesses per application. We then calculated the total number of instructions utilized for computation related access as in Equation 5.1. Table 4.2 gives us a comprehensive idea of all the instruction accesses.

Table 4.2: Number of instructions per application

Application	**Instructions**		
	Memory Access	**Branch Access**	**Computation**
NN	971583	842826	2825645
Srad_extract	2534227	1613410	4149347
B+Tree	791962001	1431982510	2990223309
NW	37965825	12701717	63215476
BFS	131783847	99622929	135156055
HotSpot	9599430	5732485	9888922
Streamcluster	1086883070	561570230	2041351051

$$\begin{aligned}\textbf{No. of computation access instructions} &= \textbf{No. of total instructions} - \\ &(\textbf{No. of memory access instructions} + \textbf{No. of Branch access})\end{aligned} \tag{4.1}$$

2. Next, we used the CodeXL Power Profiler version 2.5 [80] to give us the GPU power consumption per application. For the FPGA power we used the Intel SDK-OpenCL(AOCL) profiler to collect kernel stalls(%) information and bandwidth(MBs) utilization. We then used the StratixÂ® IV and StratixÂ® V PowerPlay Early Power Estimator tool to find the total thermal power consumption(Watts) of the device. The device's total thermal power is determined by including the Static Power (PSTATIC), Dynamic Power ($\mathrm{P_{DYNAMIC}}$), and I/O Power [78]. The leakage power dissipated by the chip, which is independent of consumer clocks, is referred to as static power. The DC bias power and transceiver DC bias power are the I/O power. The dynamic power is measured using equal lumped capacitance's calculated from internal nodes shifting logic levels within the system, as seen below.

$$\begin{aligned}DynamicPower = V_{\mathrm{CCINT}} \times \sum I_{\mathrm{CCINT}}(&LE/ALM, RAM, \\ &DSP, PLL, Clocks, HSDI, Routing)\end{aligned} \tag{4.2}$$

where power is calculated from dynamic power consumed across Adaptive Logic Modules(ALMs), RAM Blocks(RAM), DSP blocks(DSP), Phase Lock loops(PLLs), Clock, High Speed Differential I/Os(HSIO) and related routing modules.

$$Performance/Watt = \frac{No.of computation access instructions/sec}{Power consumption(Watts)} \tag{4.3}$$

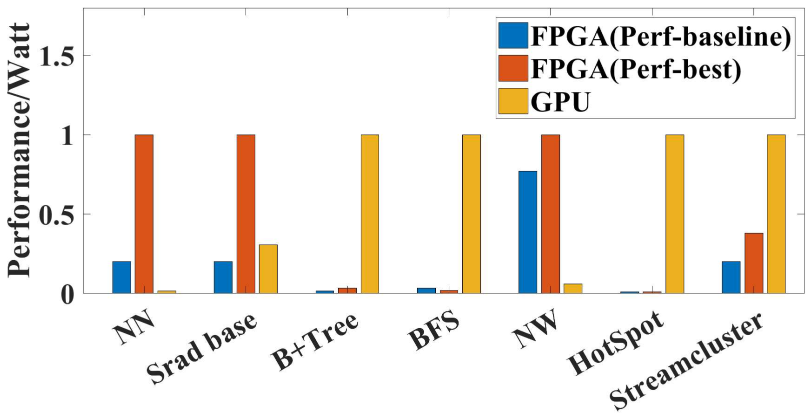


Figure 4.9: FPGA vs GPU Normalized Performance/Watts

From these results we finally calculate the Performance/Watt for every individual application as shown in Equation 5.2. We believe that Performance/Watt is an ideal standard for comparing FPGAs vs GPUs on the same scale. We normalize the values obtained for curve fitting purposes. Figure 5.14 shows a comparison between Baseline FPGA vs Best FPGA performance obtained after applying our taxonomy vs GPU. We observe that after using our taxonomy we get improved FPGA performance as against baseline FPGA for all the cases. However the FPGA performs fairly better than GPU only for three applications which have more number of regular accesses and are embarrassingly parallel. NN, Srad base and NW exhibit such properties. B+Tree, BFS, Hotspot and Streamcluster on the other hand have a large number of random accesses and suffer from stalls due to unavailability of data affecting the entire FPGA pipeline. Such applications although do perform better after applying the taxonomy, they cannot compare to the level of performance of GPUs.

4.5 Conclusion

This work's main contribution is the proposed association between OpenCL parallelism abstraction and execution of FPGAs with a new taxonomy. The goal is to give OpenCL programmers and OpenCL synthesis tools an early look at formalizing OpenCL written codes on FPGAs in order to improve their performance. FPGA-conscious OpenCL codes reach up to 6.7X maximum speedup at the same time, showing an average improvement of 3.4X, 2.2X and 2.6X for SCUMDP, MCUSDP and MCUMDP over SCUSDP. Furthermore, we also observe that the performance/watt numbers of at least 3 out of 7 applications fare far better than GPUs.

CHAPTER 5: LLVM Based Memory Decoupling

5.1 Introduction

This chapter introduces our second and most important research contribution. We present a scalable automatic LLVM based framework to decouple memory access from computation, effectively 'hiding' the memory access latency of applications. This novel LLVM-based tool introduces a new parallelism granularity that breaks down kernels to distinguish data-path and memory-path (memory read/write), resulting in a split-kernel approach that creates concurrency between current threads, computation and memory access with future threads memory access. Simultaneously, this paper proposes an LLVM-based static analysis method for detecting dynamic variable access patterns (beyond a constant stride) and controlling data dependencies through sub-kernels. LLVM analysis senses and distinguishes prefetchable (computable) and non-prefetchable (run-time dependent) patterns of data access.

5.2 FPGAs Memory Wall

One of the many drawbacks of fine-level temporal parallelism on FPGAs is the memory wall. Memory stalls are immediately exposed to the execution path since there is no run-time single-cycle thread scheduling (as in GPUs). In addition, unlike CPUs, FPGAs lack sophisticated data caching and specialized hardware prefetching to limit or mask memory access latency. Large numbers of delay buffers are often used in OpenCL-HLS software to partly mask memory access latency.

Using scratchpad memory or local memory (in the case of FPGAs) is a common way to alleviate memory stalls. Prior to launching a work-group granularity, OpenCL-HLS tools transfer data from global memory (off-chip) to local memory (on-chip) via local memory allocation (groups of thread that share same local memory exactly similar to the concept of thread blocks in CUDA memory model). The high number of memory stalls may thus be reduced by using local memory. However on the flip side, memory copy time for transferring the next work-group data from global memory space to local memory space is immediately revealed to the execution.

Memory stalls on FPGAs can also be hidden using *double buffering*, also known as ping-pong buffering, and *memory decoupling*. Wide arrays of data are separated into chunks, or tiles, and loaded into on-chip memories in the double buffering method. Off-chip memory access can be overlapped with computing by reducing the granularity of data transmission from the whole array to smaller tiles. At the same time, bulk data transfers between tiles help to amortize much of the setup and transition costs associated with accessing individual data from off-chip memory. However, this does not take into account the complexity of an application's working range, which can result in gross over-allocation of on-chip memory. Figure 5.1 shows an example of stencil computation. The blue plane is calculated along with the green planes in each iteration. The prefetched plane is yellow, and its transition overlaps the computation [3].

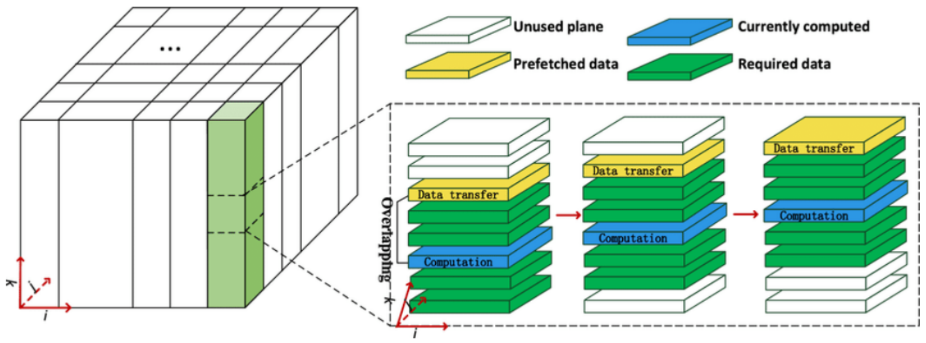


Figure 5.1: Double buffering for stencil computation [3]

Ping-pong buffering optimization necessitates an appreciation of working set size, such that only enough OCM is allocated for the present and next working sets. Due to the common occurrence of data duplication across consecutive working sets, this usually entails sizing buffer tiles to a fraction of the working set, depending on the data access stride. Moreover, one of the big drawbacks of this method is that it necessitates data with a predictable, streaming connection pattern. Despite this drawback, double buffering on FPGA will result in considerable performance gains [81, 82]. This method also necessitates a major reorganization of the application's memory as well as manual synchronization.

Memory decoupling operates at a finer granularity of data access, separating individual memory accesses into different kernels, allowing memory access and computation on FPGAs to overlap. This approach does not have the same ability to amortize memory access costs as tiling and double buffering, but it can be used with data that does not obey a streaming access semantic. Memory decoupling can be achieved with automated synchronization and minimal memory consolidation by using certain OpenCL constructs discussed later in this article.

5.2.1 Qualitative Comparison of Various Approaches

Double buffering has the largest potential to minimize stalls from off-chip memory access while keeping the smallest on-chip memory buffer of the memory optimizations mentioned above. Unfortunately, achieving high performance for double buffering necessitates a lot of hand-crafted optimization on the developer's side, making it difficult to automate. Simpler implementations don't always outperform most memory optimization techniques. Furthermore, the restriction on streaming data forbids the use of double buffering for more irregular and control-intensive programs that do not need streaming data or have a high spatial localization.

Memory decoupling, on the other hand, can be used to hide memory latency for the vast majority of data at the expense of more off-chip memory transactions. The system is only subject to the energy cost of these transfers, with the access delay of possible compute threads being overlapped with the computation of current threads. Both stream data and data with run-time dependent addresses will benefit from this memory optimization. It is also unaffected by memory data sparsity because data access granularity is limited to only the data used by a single thread.

Further, this method has trouble dealing with inner loop data access, where the per-thread memory requirement is high and repeated through threads. This scenario causes several threads to repeatedly retrieve the same data. This method is relatively easy to automate, thanks to built-in OpenCL constructs that can automatically handle synchronization through parallel producer-consumer kernels and the ability to identify complicated patterns.

In comparison to other optimization approaches, local memory optimization has the smallest performance and reliability gains. It is, however, the easiest optimization to automate, as it can be used in combination with other optimizations to cover data that isn't covered by the other approaches.

5.2.2 OpenCL Pipes

In this section, we briefly introduce the OpenCL *Pipe* semantic. We use the OpenCL Pipes to implement 'split-kernels,' a kernel temporal parallelism solution that we introduce in this work for memory decoupling approach. The pipe semantic was first implemented in OpenCL 2.0, and it was later merged into Altera's (now Intel's) OpenCL1.0 environment [8, 83]. It provides an effective data communications model for OpenCL kernels sharing data in a producer-consumer manner (with built-in synchronization).

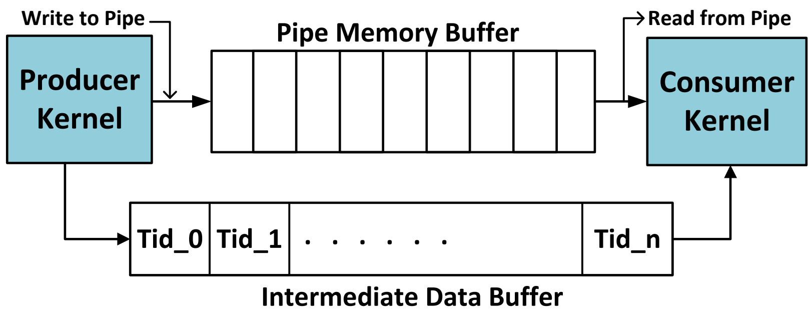


Figure 5.2: OpenCL Pipe semantic

A simple pipe construct is depicted in the diagram (Figure 5.2). It has a *pipe memory buffer* for storing inter-kernel communication data and an *intermediate buffer* for synchronizing data communication with thread id. The pipe semantic is often synthesized in OpenCL-HLS in two ways: (1) *Pointer-based* and (2) *Channel-based.* The real Pipe build is created in the global memory space using a pointer (off-chip). Channel-based, on the other hand, creates an on-chip Pipe construct assuming FPGA resources are available. Channel is a fast way to keep inter-kernel data communication on-chip by eliminating on-chip memory access between kernels.

5.3 Low Level Virtual Machine(LLVM) Framework

LLVM is a compiler framework built to provide a source and target independent intermediate representation (IR) that enables general purpose code optimization [84]. The LLVM IR provides a level of abstraction similar to that of a traditional assembly language, while also preserving high-level control semantics through an explicit control and data flow graph (CDFG). This makes it a useful abstraction for generation of application-specific hardware as part of a high-level synthesis toolchain like LegUp [85]. These features can also be used to derive algorithm-specific characteristics including memory-access patterns [86] and arithmetic intensity.

5.3.1 Decoupled Access

This work uses the concept of decoupled access to overlap memory access and thread execution at run-time to solve memory stalls in OpenCL kernels operating on FPGAs. Decoupling is applied using the OpenCL pipe semantic at a finer granularity, allowing OpenCL applications to take advantage of a new degree of parallelism while operating on FPGAs.

FPGAs' reconfigurability allows them to take advantage of parallelism at different levels in terms of processing and memory access patterns. The degrees of parallelism for OpenCL abstraction on FPGA devices are classified in Figure 5.3. The default

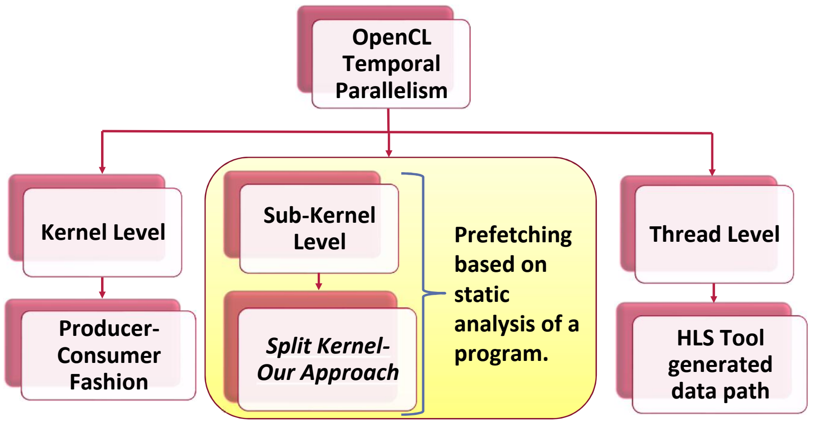

OpenCL Temporal Parallelism
Kernel Level
Producer-Consumer Fashion
Sub-Kernel Level
Split Kernel-Our Approach
Prefetching based on static analysis of a program.
Thread Level
HLS Tool generated data path

Figure 5.3: Temporal parallelism at multiple levels

parallelism inherent to OpenCL is thread-level parallelism, as seen on the right. On the left, task-level parallelism is often used by programmers to effectively schedule several kernels by defining the application's longest running kernel. We take advantage of current kernel-level parallelism and use LLVM-based automated static analysis of a program to decouple memory accesses from real computation. The suggested solution, which is a finer degree of parallelism for running massively parallel applications on FPGAs and hiding the long latency memory accesses, is applied as an extension over kernel-level parallelism illustrated in the middle.

In order to synchronize execution through parallel producer and user kernels, we make use of the OpenCL pipe and its channel dependent automatic synchronization dynamics. Decoupling memory access (read and write) from real computing is the main insight here. While certain kernels, such as the data producer and data collector kernels, are only responsible for memory accesses, others execute computations. As a result of the kernel separation, parallelism is used, which necessitates a formalized data communication and synchronization model through the kernels. We use the channel build, which automatically synthesizes on-chip memory buffers based on channel width, to preserve the current in-order thread execution model on FPGAs.

Figure 5.4 shows a conceptual architecture model of the kernel parallelism approach that we employ using kernel splitting. We propose to divide the kernel running per

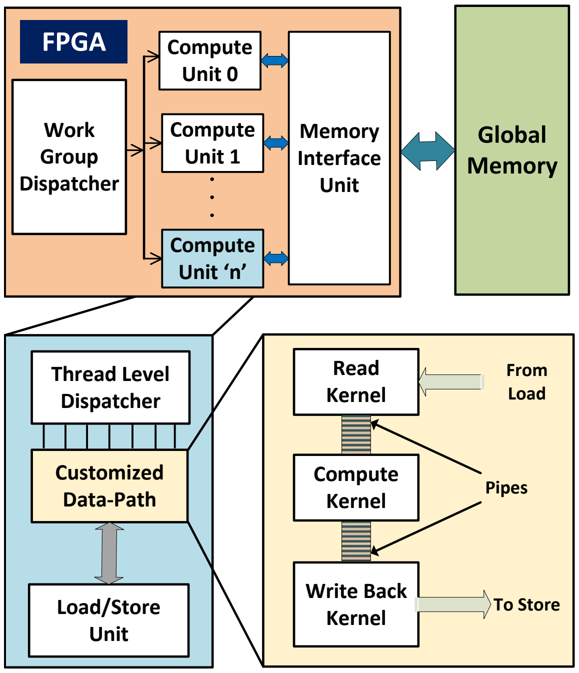


Figure 5.4: Kernel splitting for decoupled access

each Compute Unit (CU) to three major kernels: (1) Read kernel, (2) Compute kernel, (3) Write-back kernel. The kernels execute concurrently but in an asynchronous fashion. The Read and Write back kernels are responsible for loading from and storing to the global memory while the Compute kernel only deals with computation.

The Figure 5.4 depicts a computational design model for the kernel parallelism technique we use for kernel splitting. We put forward that each Compute Unit (CU) be divided into three main kernels: (1) Read kernel, (2) Compute kernel, and (3) Write-back kernel are the three types of kernels. The kernels run in parallel but in an asynchronous manner. The Read and Write back kernels are in charge of loading and saving data from and to global memory, while the Compute kernel is solely responsible for computation.

We recommend using the OpenCL Pipe based Channel build for data transmission and synchronization across kernels. We use the Channel implementation of the Pipe build to keep data communicating through the kernels on-chip. Memory accesses are made in parallel with the computation kernel, and data is exchanged through

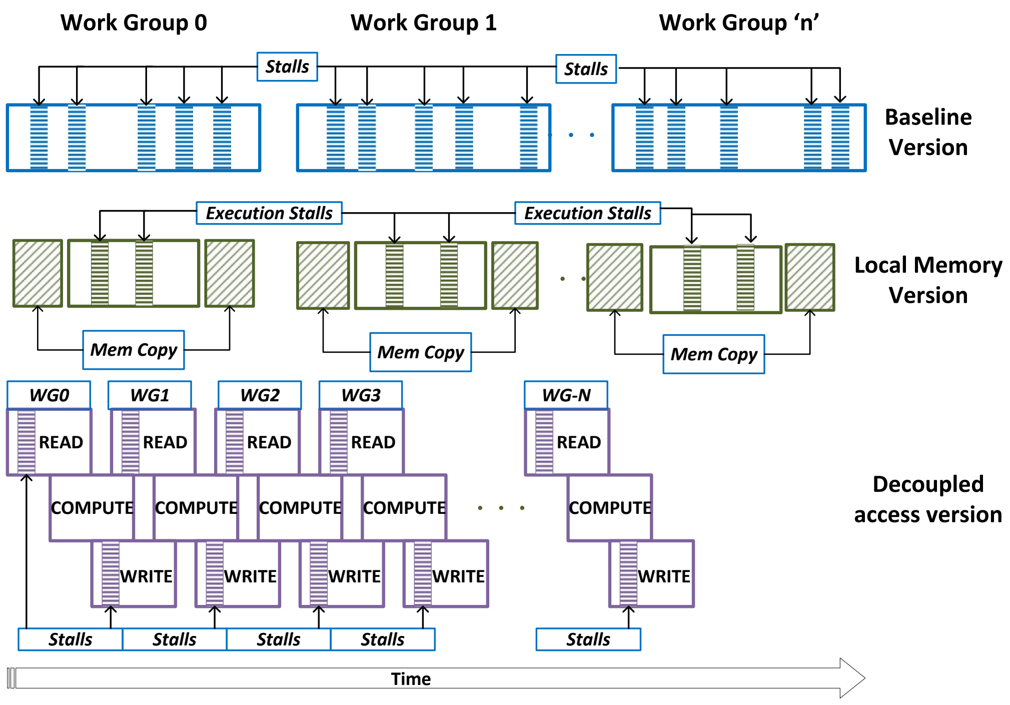


Figure 5.5: Execution pattern comparison

channels. The kernels run through several threads as long as the channels are not empty.

Figure 5.5 shows how our split-kernel solution, which is based on kernel parallelism, can be used to hide memory latency. It shows an abstract execution model of one OpenCL work-group in three different scenarios: (1) baseline (generated by OpenCL-HLS), (2) baseline with local memory, and (3) decoupled access. Both memory stalls are immediately exposed to execution in the baseline model (1). The number of real memory stalls exposed to the execution path can be reduced noticeably in the local memory model (2).

However, the execution is immediately subjected to the latency of memory transfer operations, which copy the next work-group data from global space (off-chip) to local space (on-chip). Through decoupling and operating all processes concurrently, the decoupled access version model (3) allows for the overlap of memory access latency and computation. Decoupled access will eliminate most memory stalls from execution when there are a large number of threads and a large enough Channel depth range.

5.3.2 LLVM-Based Memory Access Analysis

Memory decoupling has the ability to reduce execution stalls in the computing pipeline significantly, but handcrafted implementation of this technique becomes exceedingly time-consuming as an application's memory access complexity grows. This method can ideally be streamlined and built into OpenCL modeling software for FPGAs. We take a step toward this end goal by using the LLVM abstraction to formalize the identification of memory accesses that are appropriate for decoupling. The most fundamental criterion for decoupling a variable access is that the decoupling operation does not cause a data consistency issue.

OpenCL, as a parallel programming language, already necessitates memory accesses. We enforce the additional restriction that the variable's access must not be managed to any degree by an inner loop index variable due to hardware and power consumption limitations. This means that the instruction for determining the address for a specific access is either directly dependent on the inner-loop index variable or indirectly dependent on another variable that is itself dependent on an inner-loop index variable. Decoupling without this restriction could necessitate wide channel buffers and a lot of re-fetching of inner loop variables through threads. Instead, if on-chip memory is open, these vector accesses should be held in the main computation kernel and optimized with the local memory flag.

High-level languages like OpenCL use abstractions to mask the details of memory address intricacies from the end user. We switch to the lower level abstraction of LLVM to gain further insight into address calculations. LLVM is a target-independent programming abstraction that offers a micro-ops-level view of an application with no restriction on the number of registers available. This makes memory accesses easy to spot, as well as providing insight into how array indexes shift during execution. To make this investigation easier, we created a C++-based method that parses LLVM

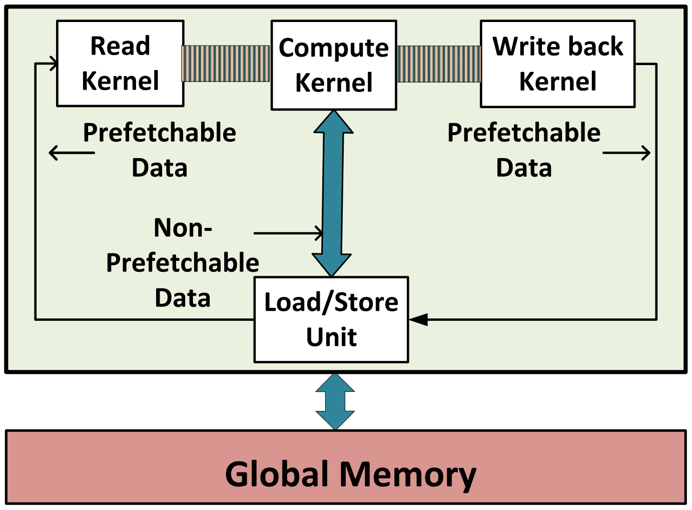


Figure 5.6: Memory decoupling with pipes

and automatically identifies variables appropriate for decoupling.

5.3.3 Algorithmic Implementation

Our suggested static memory access analysis is presented in the Algorithm 1. Using Clang, we get the LLVM IR from our computation kernel. The global point list (line 2) and the control and data flow graph (CDFG) are then extracted from this file (line 3). We then go through our list of global pointers one by one, evaluating each one for decoupling (lines 4 to 6).

We check our CDFG for any address offset calculations referring each pointer in our pointer list (lines 9 and 10). LLVM exposes this address offset approximation as a *getelementptr* or GEP command. We call the *ISSEPARABLE* function if we see a GEP instruction that corresponds to our target pointer to see if it can be decoupled.

Lines 15 to 31 of the *ISSEPARABLE* function recursively maps the CDFG for all dependencies of the beginning GEP instruction. If it encounters an inner-loop index variable, known in LLVM as an induction variable or *INDVAR*, the pointer access is automatically marked as non-separable (lines 21 and 22). During regular compiler passes, these induction variables are automatically detected and named. If the dependence is on one of our global pointers, we verify its separability by calling

Algorithm 1 LLVM Analyzer

```
 1: function MAIN()
 2:     $ptrList \leftarrow PARSE\_POINTERS(INPUT\_LLVM\_FILE)$
 3:     $cdfg \leftarrow PARSE\_CDFG(INPUT\_LLVM\_FILE)$
 4:     for each $ptr \in ptrList$ do
 5:         $PTR\_EVAL(ptr, ptrList, cdfg)$
 6:     end for
 7: end function

 8: function PTR_EVAL($PTR, PTR\_LIST, CDFG$)
 9:     for each $node \in cdfg$ do
10:         if $node.type = GEP$ and $node.ptr = ptr$ then
11:             $ptr.separable \leftarrow IS\_SEPARABLE(node, CDFG, PTR\_LIST)$
12:         end if
13:     end for
14: end function

15: function IS_SEPARABLE($NODE, CDFG, PTR\_LIST$)
16:     bool $separable$
17:     if NODE.op = THREAD_ID_CALL then
18:         return TRUE
19:     else
20:         for each $dep \in NODE.deps$ do
21:             if $dep.isINDVAR()$ then
22:                 $separable \leftarrow$ FALSE
23:             else if $dep \in PTR\_LIST$ then
24:                 $separable \leftarrow PTR\_EVAL(dep, PTR\_LIST, CDFG)$
25:             else
26:                 $separable \leftarrow IS\_SEPARABLE(dep, CDFG, PTR\_LIST)$
27:             end if
28:         end for
29:     end if
30:     return $separable$
31: end function
```

$PTREVAL$ on that pointer. We assume targeted pointer access to be separable if we approach the beginning of our CDFG without encountering a $INDVAR$ or non-separable global pointer. We don't have to think about tracing variables through functions and processes because of HLS' feature inlining restrictions.

A conceptual realization of the proposed design approach is seen in Figure 5.8. The OpenCL-HLS tool-chain blackbox can automate the entire process. The Figure 5.7 reveals a possible design flow for automation. A CDFG is produced for the kernel during synthesis, which is used for data-path mapping and synthesis. Our LLVM parser, as defined in Algorithm 1, is used to define the dependency trees for each variable access within the CDFG, as seen in Figure 5.7, prior to the final mapping.

The outlined fragments that correspond to the separable variables can then be separated into different kernels. A *channelwrite* function call is added to the end of read kernels to facilitate inter-kernel communication, and a *channelread* function call is added to write kernels to support inter-kernel communication.

5.4 Experiments

The experimental findings for testing the utility of our decoupling-based temporal parallelism approach are presented in this section. For our tests, we use the same eight OpenCL applications from the Rodinia benchmarks suite [31], which include a combination of hierarchical grid, graph traversal, linear algebra, and other application

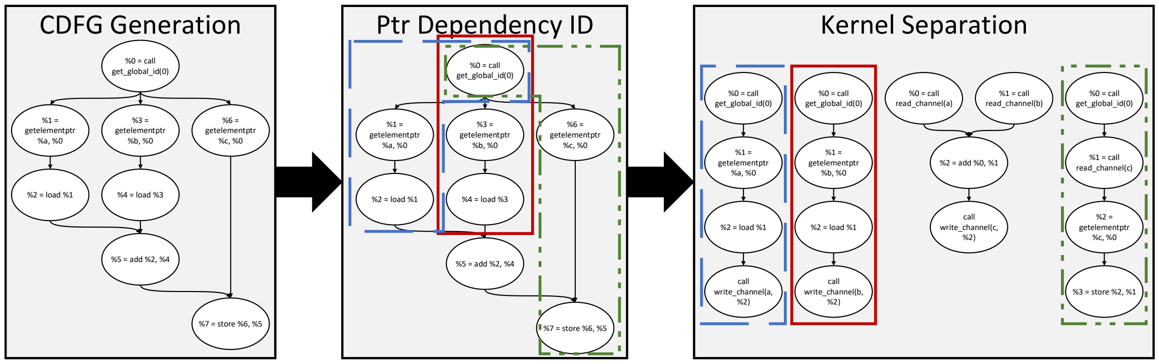

CDFG Generation
%0 = call get_global_id(0)
%1 = getelementptr %a, %0
%3 = getelementptr %b, %0
%6 = getelementptr %c, %0
%2 = load %1
%4 = load %3
%5 = add %2, %4
%7 = store %6, %5
Ptr Dependency ID
%0 = call get_global_id(0)
%1 = getelementptr %a, %0
%3 = getelementptr %b, %0
%6 = getelementptr %c, %0
%2 = load %1
%4 = load %3
%5 = add %2, %4
%7 = store %6, %5
Kernel Separation
%0 = call get_global_id(0)
%1 = getelementptr %a, %0
%2 = load %1
call write_channel(a, %2)
%0 = call get_global_id(0)
%1 = getelementptr %b, %0
%2 = load %1
call write_channel(b, %2)
%0 = call read_channel(a)
%1 = call read_channel(b)
%2 = add %0, %1
call write_channel(c, %2)
%0 = call get_global_id(0)
%1 = call read_channel(c)
%2 = getelementptr %c, %0
%3 = store %2, %1

Figure 5.7: Automated memory access decomposition

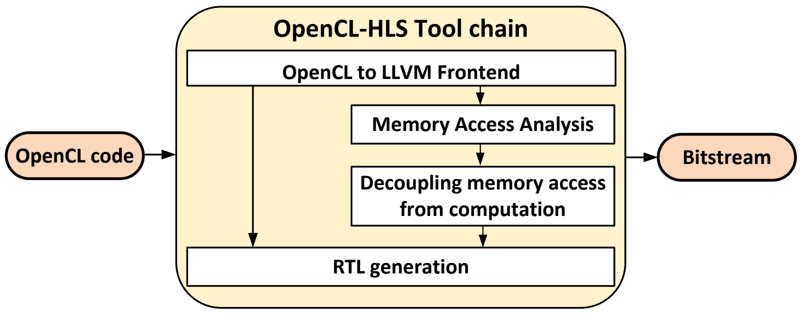


Figure 5.8: Conceptual realization of decoupled access incorporated into OpenCL HLS

Table 5.1: System characteristics used for study.

Host	Intel(R) Core(TM) i7-7700K
Host clock	4.2 GHz
FPGA Family	Stratix V
FPGA Device	5SGXMA7H2FE35C2
CLBs	234,720
Registers	939K
Block Memory bits	52,428,800
DSP Blocks	256

domains. B+Tree, BFS, Gaussian, Hotspot, Srad, LUD, and Nearest Neighbour are the applications employed as the test set. The parameters of our FPGA platform are mentioned in the Table 6.7. For compiling and synthesizing OpenCL code, we use the Intel OpenCL SDK with the Quartus back-end [8]. The Intel SDK-OpenCL(AOCL) profiler is also used to extract accurate execution performance. The profiler collects kernel output statistics, global memory bandwidth efficiency, and stalls (Percentage time that memory access caused the stall in kernel execution).

We recorded Logic utilization, Memory blocks, and Register numbers to evaluate resource overhead since these three parameters are the most influenced by the OpenCL pipe overhead. Also, even though we have some floating point applications in our benchmark, we don't see any significant difference in DSP use because the calculation of all applications in the decoupled access version stays the same.

Table 5.2: Kernel global variable data information per application

Benchmarks	Number of Threads	Decouplable variables		Non-Decouplable variables		Decouplable(%)
		Size per Thread (Bytes)	Total Size (Bytes)	Size per Thread (Bytes)	Total Size (Bytes)	
B+ Tree FindK	65536	12	786432	8	524288	60
B+ Tree RangeK	65536	20	1310720	0	0	100
Gaussian	65536	12	786432	0	0	100
HotSpot	16384	12	196608	0	0	100
BFS	1048576	8	8388608	12	12582912	40
NN	42764	8	342112	0	0	100
Srad Extract	65536	4	262144	0	0	100
LUD Diagonal	4096	4	16384	0	0	100

Table 5.3: Baseline profiling information for each application

Benchmarks	Resource utilization(%)			Execution Time(ms)	Bandwidth (MBps)	Stalls(%)	Clock Frequency (MHz)	Power (Watt)
	Logic utilization	Memory blocks	Registers					
B+ Tree FindK	23	24	11	25.79	3266	19.7	223.56	2.7
B+ Tree RangeK	25	24	12	17.39	5247	3.01	218.6	2.2
Gaussian	21	20	9	6.56	3265.5	8.6	228.2	2.1
HotSpot	23	16	4	0.52	1181.1	76.8	233.5	1.86
BFS	22	17	6	2.34	957.4	40.7	288.4	2.3
NN	20	19	8	0.28	3002.7	0.0	245.2	2.17
Srad Extract	20	16	8	1.23	1862	0.0	250	2.15
LUD Diagonal	23	20	10	0.11	140.4	0.23	234.5	2.27

5.5 Discussion

In this section, we talk about the experimental evaluations that were performed. First, we used our proposed LLVM analyzer method (presented in Algorithm 1 to define decouplable variables per kernel, as well as the number of decouplable and non-decouplable variables for each kernel, in order to conduct the experiments. The results of our static LLVM analysis for each OpenCL kernel are mentioned in Table 5.2. It displays the parallelism size (number of threads per kernel) as well as the size of decouplable and non-decouplable variables per thread for each kernel. It also shows the total number of decouplable and non-decouplable data per kernel in bytes.

The majority of the variables are decouplable in general (suitable for decoupled memory access). Non-decouplable variables are only found in two kernels: B+ Tree FindK and BFS. Please keep in mind that the table only displays the global memory access demand per thread. OpenCL-HLS tools reserve on-chip memory for local variables automatically (private memory in OpenCL semantic).

5.5.0.1 Performance Analysis

The baseline profile information is given in Table 5.3, which serves as a basis for interpreting the comprehensive performance analysis. The naive implementation of these programs, with no optimization, is the baseline version. The relative performance gain of benchmarks (with split-kernels temporal parallelism) over the baseline implementation as seen in Figure 6.12. It also shows how the local memory solution has improved from the baseline implementation.

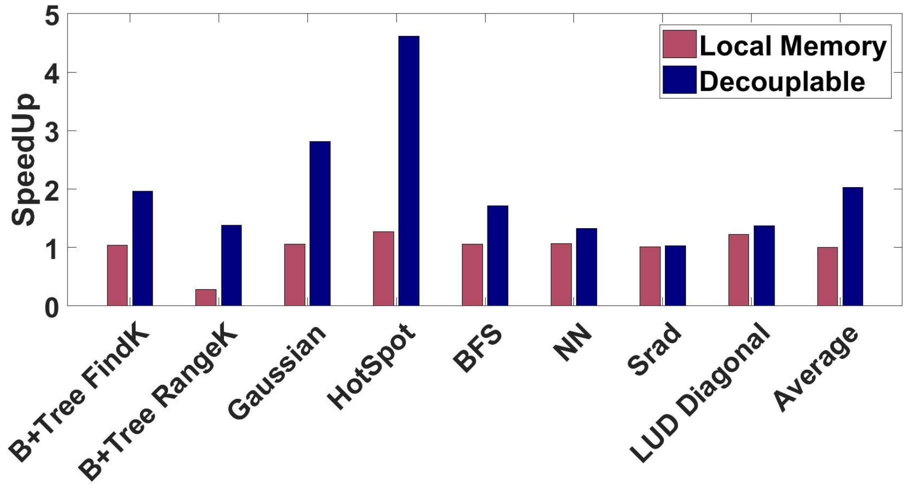


Figure 5.9: Performance improvement over the baseline

We calculate average speedup as a times change over absolute output numbers observed from each program [87]. Overall, our LLVM-based parallelism solution improves performance by up to 2x as compared to a baseline implementation. For the *hotspot* benchmark, it also reaches a maximum output gain of 4.6x. The local memory solution, on the other hand, can only reach a 1.03x speedup over the baseline implementation.

In order to provide further insight regarding the source of speedup, Figure 5.10 and Figure 5.11 reveal the percentage of memory stalls reduction and memory bandwidth improvement over baseline implementation, respectively [1]. On average, we observe

[1]In Figure 5.10, no improvement in memory stalls are shown by asterisk

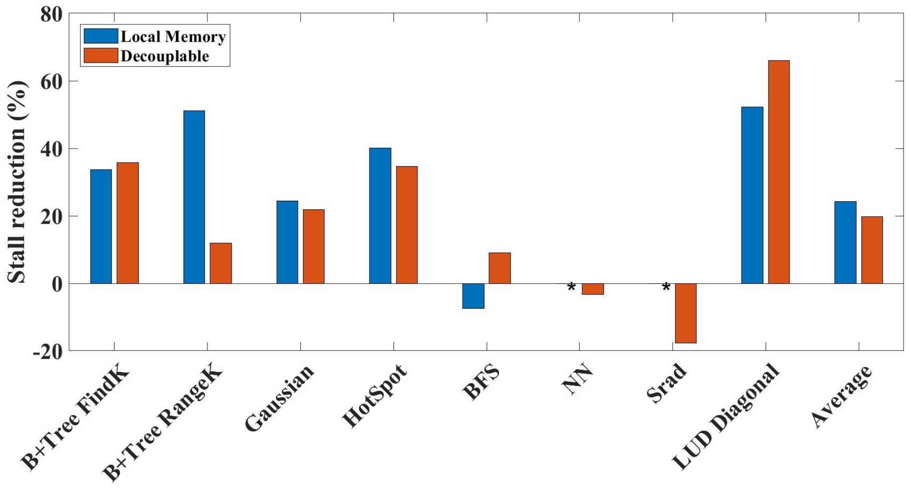


Figure 5.10: Memory stalls reduction over the baseline

26% of stalls reduction over baseline. We also observe 1.6x improvement in bandwidth utilization on average.

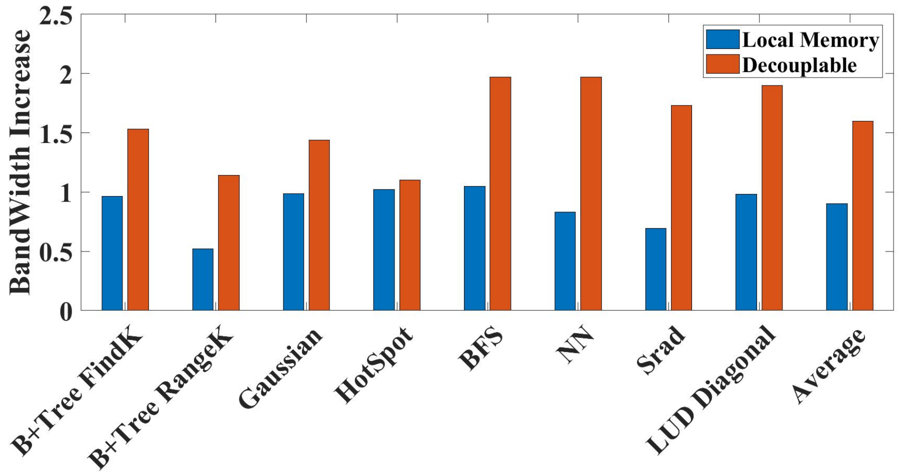


Figure 5.11: Memory bandwidth improvement over the baseline

Figures 5.10 and 5.11 report the memory stalls reduction in percentage and memory bandwidth gain over baseline implementation, respectively. This gives us more insight into the origins of speedup.[2]. We see a 26% drop in stalls on average as compared to the baseline. On average, bandwidth usage has increased by 1.6 times.

The performance of an application is observed to be a trade-off between percentage of decouplable variables, memory stalls and total job size. Furthermore applications that are highly regular and have very low stalls in their baseline versions do not

[2]In Figure 5.10, no improvement in memory stalls are shown by asterisk

benefit from our approach. As an example, *Srad Extract* benchmark has no stalls initially as shown in Table 5.3 and conversely adds up more stalls(18%) even with a 1.97x increase in bandwidth utilization. This leads to a very low performance improvement, only 1.03x. This is similar to the behaviour of the *NN* application leading to a performance improvement of a mere 1.3x.

Here we can note, that the percentage of decouplable variables, memory stalls, and overall job size are all parts of trade-offs in the application output. Additionally, our solution has little advantage for applications that are extremely routine and have very few stalls in their baseline models.

5.5.0.2 Memory Bandwidth

In contrast, the *hotspot* benchmark has higher stalls reduction of 38% and more decouplable data (as reflected in Table 5.2). This gives a higher performance improvement of 4.6X times over baseline. Two other benchmarks namely *B+ Tree Find K* and *BFS* do not achieve comparable speedup since they have more number of non-decouplable variables which affects their percentage of stall reduction.

The *hotspot* application, on the other hand, has a 38% lower stall rate and more decouplable data (as seen in Table 5.2). This results in a 4.6X increase in consistency over baseline. Two other benchmarks, *B+ Tree Find K* and *BFS*, do not reach similar speedup because they have a greater amount of non-decouplable variables that impair their stall reduction percentage.

An important consideration here is that the local memory solution will also accomplish a similar decrease in memory stalls (26% percent over baseline). However, since memory copies are part of the execution path, they add to the data-path and contribute to the overall execution time, resulting in just a minor performance boost.

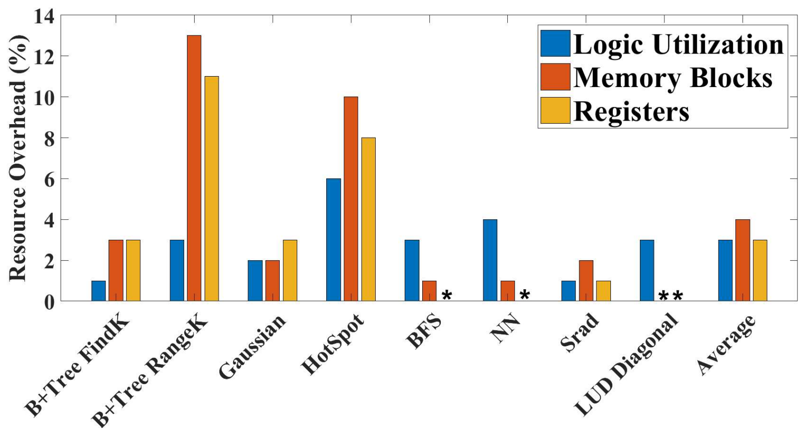


Figure 5.12: Resource utilization overhead over the baseline

5.5.0.3 Resource Overhead

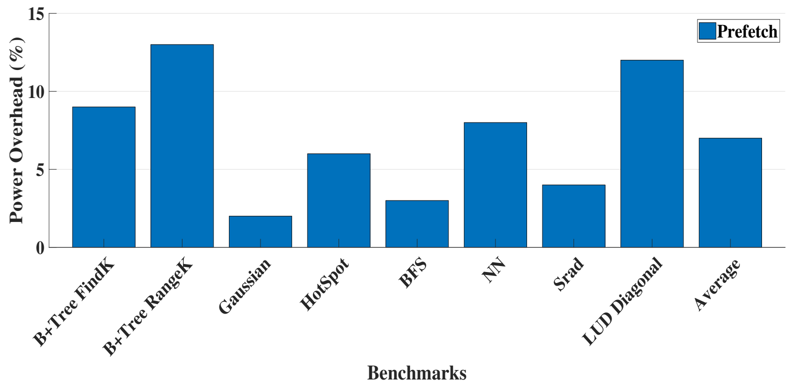
The impact of our proposed LLVM-based parallelism on power, energy and resource utilization is briefly discussed in this section. The percentage of resource overhead over the baseline is seen in Figure 5.12 . Zero resource overhead is shown by an asterisk in Figure 5.12.

Additional register blocks, memory blocks, and combinational logic are all necessary to create the pipe (channel) semantic, which adds to the resource overhead. We see a 3% rise in register blocks, a 4% increase in memory blocks, and a 3% increase in combinatorial logic blocks on average.

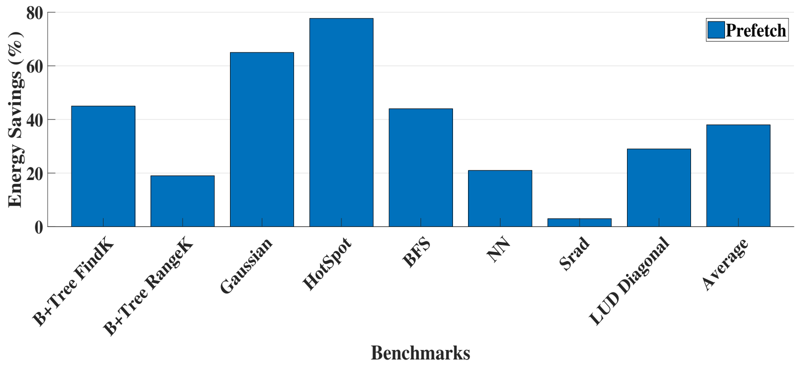
5.5.0.4 Power Overhead and Energy Saving

The power overhead induced by increased FPGA resource usage is presented in this section. We used the Stratix V PowerPlay Early Power Estimator method to measure the power overhead. The amount of Static Power (P_{STATIC}), Dynamic Power (P_{DYNAMIC}), and I/O Power [78] is used to measure total thermal power. Internal nodes shifting logic levels inside the system in the form of equal lumped capacitance's are used to quantify dynamic capacity.

The relative power overhead and energy savings over the baseline implementation are seen in Figure 5.13. On average, we see a 7% rise in power usage as compared



(a) Power



(b) Energy

Figure 5.13: Power overhead and energy saving over the baseline

to the reference implementation. In comparison to the reference implementation, we see a 40% reduction in total energy consumption. The energy savings was due to a substantial increase in execution speed and a decrease in average execution time. The *b+tree rangek* benchmark has a maximum power overhead of 13% (due to its comparatively visible resource overhead, as seen in Figure 5.12.

5.5.0.5 Performance per Watt

We use Performance per watt as an assessment criterion to measure the combined output and power efficiency of our proposed solution, as we did in previous work. We need to catch real computation demand (arithmetic operations) per each kernel in order to quantify Performance per watt.

We used PIN PYtracer [79] to extract the total number of instructions, memory, and branch accesses across each kernel to determine computation demand. Using the

Equation 5.1, we calculated the total computation demand per kernel.

$$\begin{aligned}\#ComputeOps = \#TotalInstructions- \\ (\#MemoryInstructions + \#BranchInstructions)\end{aligned} \tag{5.1}$$

$$Perf/Watt = \frac{\#ComputeInstructions/sec}{Powerconsumption(Watts)} \tag{5.2}$$

The results of Perf/Watt (Figure 5.14) show a comparison between normalized values obtained with respect to our proposed LLVM based parallelism approach. Our approach clearly wins over baseline and local memory versions for all kernels, except *LUD*. Again, we can see that improvement arises due to the cumulative effect of decouplable variables(Table 5.2), reduction of stalls(Figure 5.10) and improvement in performance numbers (Figure 6.12), against miniscule power overhead (Figure 6.14) due to increase in resource utilization (Figure 5.12).

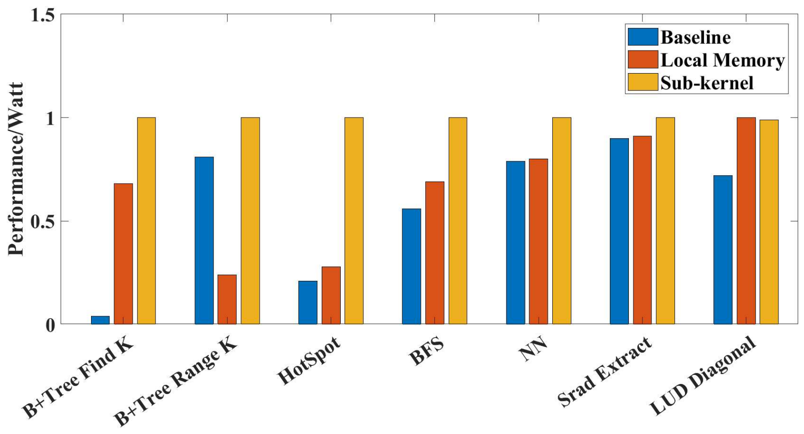


Figure 5.14: Normalized Performance/Watts

5.5.0.6 Channel Depth- A Performance Metric

The width of the channel or pipe is another important factor for the efficiency of our decoupling process. For example, we use the *b+tree rangek* to create different channels for each statically decouplable global variable. The data-path includes the private variables that are internal to each kernel. We increase the pipe depths for each of the global variables from *4* to *256*, thus expanding the channel buffer size per element.

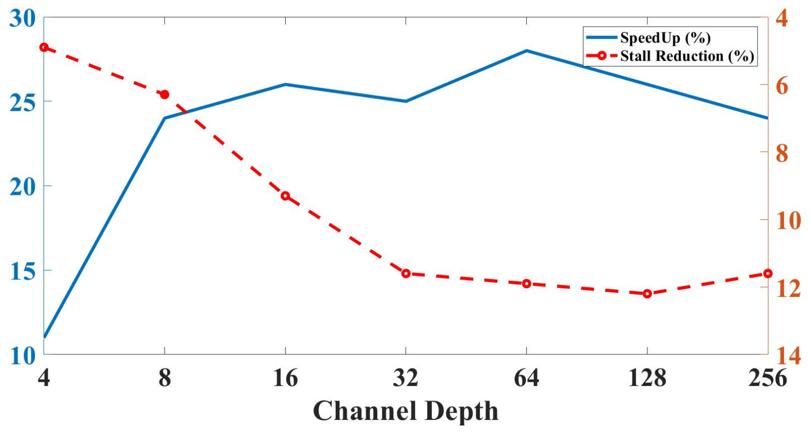


Figure 5.15: B+ Tree RangeK performance improvement and memory stalls over increasing channel depth

From Figure 5.15 we see a steady change in results (with peak improvement at the channel depth of 64). The 64-channel depth therefore ensures that memory stalls are kept to a minimum (2.65 percent of total memory stalls). We see a decrease in performance gain after channel depth 64, which is attributed to the greater channel depth's propagation latency.

5.5.0.7 Additional Factors Affecting Performance

Finally, we add a metric that affects efficiency, namely kernel load-balancing, to our comprehensive optimizations. Load balancing is the method of evenly distributing data among read, compute, and write kernels such that more data is fetched and processed in the same amount of time, resulting in a balanced work allocation and delivery among the channels. Load balancing can significantly increase efficiency by reducing certain inherent memory stalls in the pipeline. We show an example of the BFS application and provide a detailed overview to investigate this effect.

We look at the BFS application for this test. BFS contains 2 internal kernels viz

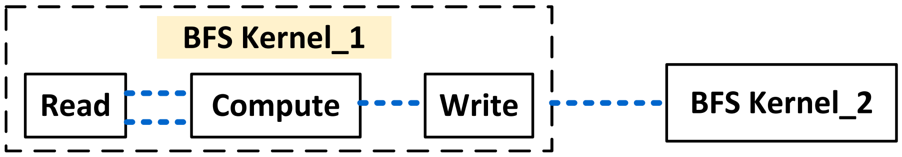


Figure 5.16: BFS split-kernel baseline channel version

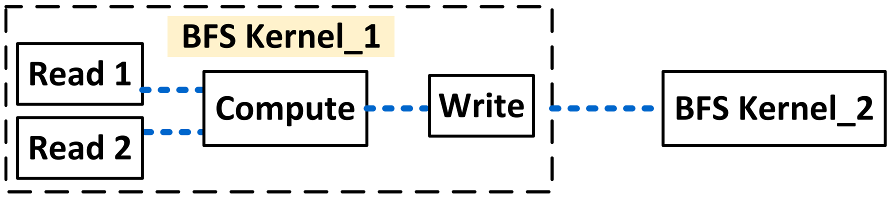


Figure 5.17: BFS load balanced version 1

BFS Kernel 1 and BFS Kernel 2. Figure 5.16 shows the baseline application. We've kept the channel diameter to a constant size of 4 as a test case for this experiment, and the blue dotted lines reflect channels. We construct two channels in the Read kernel and link them to the Compute kernel in the first example. In this case, performance is a function of memory stalls caused by a load mismatch between the Read and Compute kernels, since the Read kernel would feed variable data 1 first and then variable data 2, causing stalls in the Compute kernel.

We next solve this problem by 2 different approaches.

1. As shown in Figure 5.17 making 2 sub-kernels that have concurrently occurring independent reads.

2. Figure 5.18 that has perfectly balanced read and write kernel paths leading to efficient data transfer.

We validate the results from Figure 5.19. More details on the veracity of all the approaches have been detailed in our published work [88].

5.6 Memory Decoupling for Cloud FPGAs

An important and essential part of our work is exploring the validity and scalability of our proposed approach. In this section we move from the 'edge' to the 'cloud'

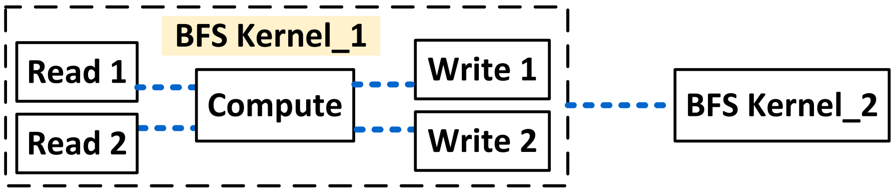


Figure 5.18: BFS load balanced version 2

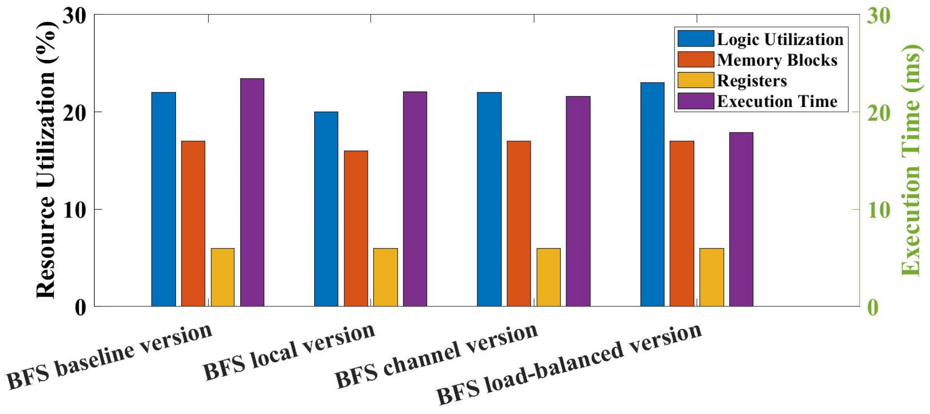


Figure 5.19: Execution time vs resource utilization results for various configurations of BFS

platform across different HLS tool-chains viz. Intel HLS to Xilinx SDAccel HLS. The Xilinx based AWS cloud platform is a bigger and better FPGA as opposed to our local FPGA. Here we deal with new platforms, new synthesis tools and encounter unique programmability challenges also.

Memory decoupling to reduce memory access latency has been explored in our previous works. This work is a continuation of our LLVM based automation tool for memory decoupling approach on the Xilinx based AWS cloud platform. We learn to use Xilinx based synthesis tools(SDAccel), identify the programming challenges in porting the OpenCL codes and suggest a simple, generic approach for decoupling and mitigate the memory wall bottleneck problem.

This research employs an updated variant of our LLVM-based automatic method, which statically analyzes an application's dynamic access behaviour and data dependencies before identifying global variables that can be decoupled from computation.

Next, we combine this data with the current definition of OpenCL pipes [35, 69, 89] (introduced in OpenCL version 2.0) to decompose OpenCL application kernels into distinct 'memory-read/write' and 'computation' sub-sections. Then we run each of those kernels in parallel. The pipe build is in charge of coordination between kernels and thread synchronization in OpenCL.

This is the very first piece of work to provide a common solution by mixing LLVM approach and OpenCL pipes for decoupling memory and compute, thus hiding memory latency and enhancing efficiency when executing on FPGA with massively parallel workloads suited for high-performance applications. We've open sourced all our source codes[3].

We can list the contributions of this work as follows:-

1. Introduce a new way of modifying the OpenCL pipe principle for FPGA-based applications applied on the AWS cloud.

2. A comprehensive and generic step-by-step method for implementation of split-kernel piping with OpenCL.

3. Port 7 of the new Rodinia benchmark suite version applications (version 3.1) [11] originally written to run on FPGAs in Xilinx cloud based FPGA systems.

5.6.1 OpenCL Semantic on Xilinx Cloud Based FPGAs

FPGA Amazon Machine Interface, contains the pre-installed Xilinx SDAccel HLS and Vivado Toolchains that is used by Xilinx AWS cloud FPGAs. With its performance profiler, the SDAccel provides all the traditional features for production by finding bottlenecks.

The OpenCL Execution Semantics of the default OpenCL Context created on Xilinx FPGA is seen in Figure 5.20. At a higher granularity stage we can see that the dispatcher work-group and memory interface unit distribute and synchronize work loads between computer units. A dedicated memory, thread dispatcher, load-store

[3]https://github.com/TeCSAR-UNCC/OpenCL-Pipes

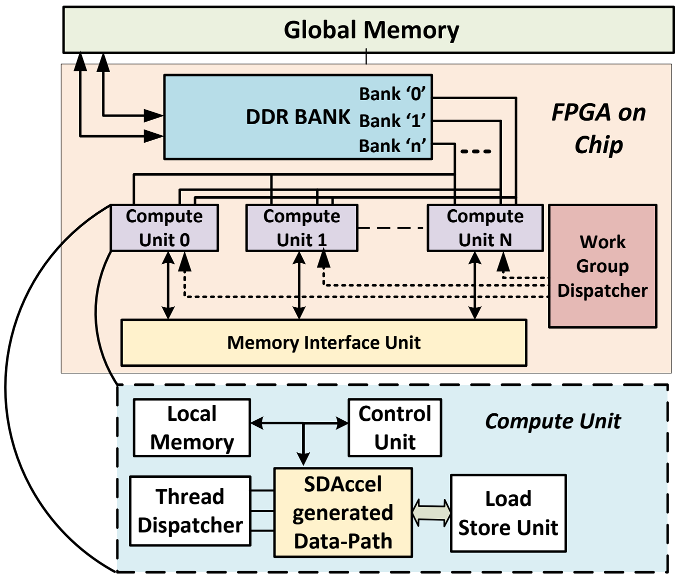


Figure 5.20: OpenCL semantic on Xilinx based FPGAs

and control unit is included in the synthesized compute unit (as seen for calculation unit 0). OpenCL threads share the same data path and run in a stand-alone manner over several SDAccel HLS tool pipeline levels.

5.6.2 Decoupling Memory Access from Computation

The design of splitting kernels for memory access decoupling is dependent on synchronous communication between each kernel. We do this by simply using the Xilinx OpenCL pipe semantic's producer-consumer model, which ensures that threads execute in the correct order. When the address of the variables is not statically identifiable, or in other words, is run-time based, the big stumbling block for kernel splitting occurs once again as discussed in previous section.

An updated version of our [90] LLVM-based tool is used to statically scan the application's run-time behaviour and automatically classify the complete set of global variables as 'decouplable' or not. We use the OpenCL 'Queue' in Xilinx SDAccel to overlap memory access and execute thread during run-time to allow all the read, compute, and write-back kernels work simultaneously.Figure 5.21 displays our solution for the Xilinx toolchain.

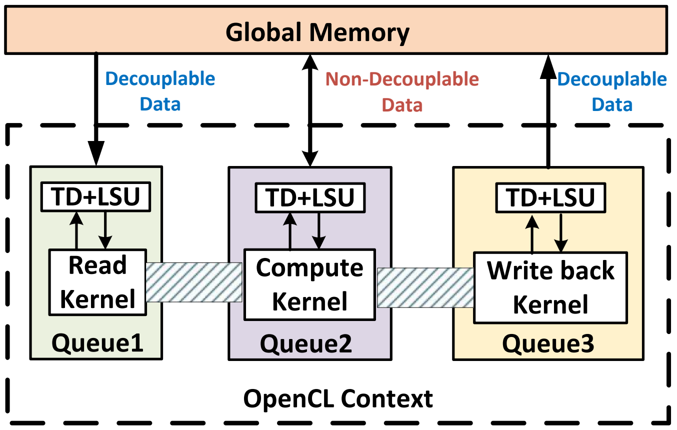


Figure 5.21: Kernel communication through OpenCL pipes

5.6.3 Generic Source-Code Template

In this section, we present a step-by-step guide that can be used as a standardized framework for decoupling memory access and computation in this section. For demonstration purposes, we use the Nearest Neighbor(NN) application from the Rodinia benchmark suite.

We generate our LLVM codes with CLang (version 3), which are then fed into the static LLVM analysis method, which helps distinguish between decouplable (predictable) global variables and run-time dependent global variables (Non-decouplable). Next, we optimize our kernel by going over the default pipe models.

Listing 5.1: Nearest Neighbor kernel baseline

```
__kernel void NN(__global LatLong *d_locations,
        __global float *d_distances...)
{...
if (globalId < numRecords) {
 __global LatLong *latLong = d_locations+globalId;
 __global float *dist=d_distances+globalId;
 *dist = (float)sqrt((lat-latLong->lat)*(lat-latLong->lat)+(lng-
    latLong->lng)*(lng-latLong->lng));
...}
```

Listing 5.2: Nearest Neighbor kernel decouplable version

```
//Declaring Pipe buffer memory with Depth 'DEPTH'
pipe float p0 __attribute__((xcl_reqd_pipe_depth(DEPTH)));
pipe float p1 __attribute__((xcl_reqd_pipe_depth(DEPTH)));
```

Listing 5.3: Nearest Neighbor kernel decouplable version(continued)

```
__kernel void NN_read(__global const float *x,
             __global const float *y)
{...
write_pipe_block(p0, &d_locations[globalId]);
...}

__kernel void NN_compute(...)
{...
read_pipe_block(p0, &loc_lat);
float d_distances = (...);
write_pipe_block(p1,&d_distances);
...}

__kernel void NN_write(__global float *d_distances,
                       ...)
{...
read_pipe_block(p1, (d_distances+globalId));
}
```

The Nearest Neighbor application's baseline kernel is seen in Listing 5.1. Both local variables and extra computations not related to global memory access have been omitted from the port list. We will see that all of the global variables in this case are decouplable using the LLVM static analysis function [90].

The template for splitting the kernels is seen in Listings 5.2 and 5.3. The 5.2 listing is only used to create OpenCL pipes with a user-defined special Pipe depth. Pipe depths of powers of two up to 32768 are possible with the Xilinx SDAccel toolchain.

The template for splitting the kernels is seen in Listings 5.2 and 5.3. The 5.2 listing is only used to create OpenCL pipes with a user-defined special Pipe depth. Pipe depths of powers of two up to 32768 are possible with the Xilinx SDAccel toolchain.

Table 5.4: Baseline profiling information for each application

Benchmarks	Resource utilization(%)				Execution Time(ms)	Avg bandwidth utilization(%)
	LUTs	LUTMem	REG	BRAM		
Nearest Neigbor	0.37	0.22	0.25	0.09	5.68	26.49
SRAD Extract	0.33	0.19	0.16	0.05	319.57	56.75
Gaussian	0.37	0.18	0.22	0.05	3681.99	24.95
Hotspot	0.69	0.21	0.37	8.76	47.63	4.58
BFS	0.6	0.36	0.35	0.14	2.498	9.664
LUD Diag	0.34	0.19	0.2	0.23	7.47	16.99
LUD Internal	0.36	0.19	0.22	0.09	0.2	19.76

The exact splitting of kernels into read, compute, and write is seen in Listing 5.3. The transformed kernel code is renamed in the host source code, and is encapsulated in the same OpenCL sense as the transformed kernel code, but within three queues figreffig:pipetransfer. Each queue is divided into separate compute units, each with its own Thread Dispatcher (TD) and Load/Store unit (LSU), which operate in parallel. The OpenCL pipe's property allows for inter-kernel communication as well as thread synchronization.

5.7 Experiments

We use eight standard applications from the Rodinia benchmark suite [11] for our experimental evaluation purposes. They namely are Nearest Neighbors, Srad_base(Srad extract application), Gaussian, B+Tree, LUD Diagonal, LUD internal and Hotspot. We have used the Intel SDK for OpenCL [8] based on OpenCL version 1.0 for compiling the OpenCL code. Our FPGA implementations are synthesized on the Virtex Ultrascale FPGA while we have used the AMD FirePro W7100 device for our GPU implementation. systemdetails shows the system parameters of our FPGA and GPU platform in more detail.

Table 5.5: System characteristics used for study.

Host	Intel(R) Core(TM) i7-7700K
Host clock	4.2 GHz
FPGA Family	Virtex Ultrascale
FPGA Device	VU9P
LUTs	1,157,112
LUTMem	584,988
REG	2,330,479
Block RAM blocks	2,134
GPU	AMD FirePro W7100
GPU Max Compute Units	28

5.8 Discussion

Table 6.1 displays the profiling data for each application's baseline implementation. Each OpenCL kernel's resource consumption, execution time, and average bandwidth are mentioned. It should be remembered that the Hotspot program consumes the most energy, while the SRAD Extract consumes the least. The Gaussian application kernel takes the longest to run, followed by the SRAD, and finally LUD Internal. Each kernel's calculation, data collection, and data dependence all relate to the execution time. The same pattern can be seen in average bandwidth usage, with SRAD Extract having the highest, NN and Gaussian having equal average bandwidth utilization and following SRAD, and hotspot having the lowest.

5.8.0.1 Performance Analysis

The relative performance gain of the Rodinia benchmarks with the pipeline implementation over the reference performance as seen in Figure 5.22. Overall, the pipeline architecture achieves 6x speedup as compared to the baseline implementation. The LUD Diagonal implementation achieves the highest efficiency, with a 16x increase in performance. The LUD Internal program, on the other hand, shows the least change in efficiency. Since there are more local variables whose values are dependent on the global variables, the LUD Internal application has more stalls than the LUD diagonal application.

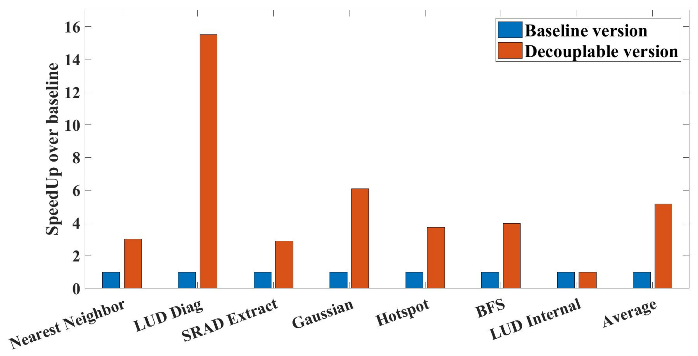


Figure 5.22: Performance improvement over baseline

5.8.0.2 Resource Overhead

The amount of LUTs, Registers, BRAM, and LUT memory used to create the pipe semantics is factored into the resource usage. As a result, when opposed to the baseline implementation, they display an increase in resource utilization.

Next we look at the resource overhead numbers (Figure 6.8) over the baseline. LUT and LUT memory have an average utilization of 1.75%, while Registers and BRAM have an average utilization of 1.5% and 0.5%, respectively. The pipe seems to act as a single producer-consumer system. As a consequence, multiple access to the same global variable necessitates the use of separate channels, increasing the register stack, memory block, and logic gate count. Gaussian's pipe version consumes 7 pipes, increasing resource consumption and causing it to consume the highest amount of resources; similarly, LUD Internal needs 3 pipes, which is comparable to Gaussian.

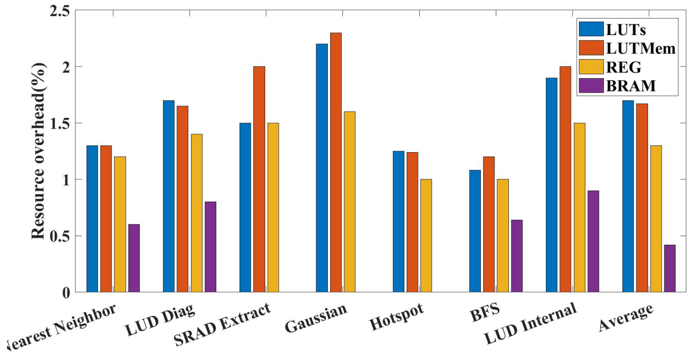


Figure 5.23: Resource overhead over baseline

5.8.0.3 Memory Bandwidth Results

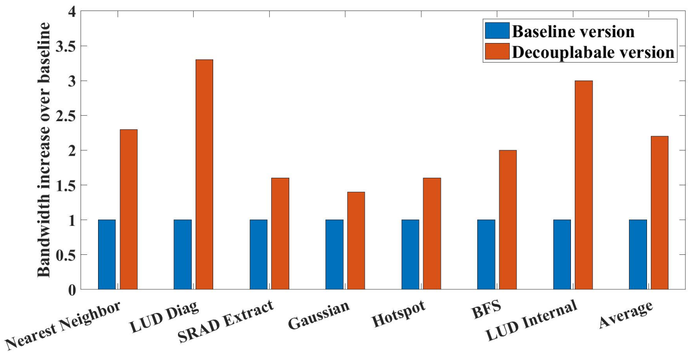


Figure 5.24: Bandwidth improvement over baseline

The average bandwidth increase of the pipeline version over the baseline version is reported in Figure 6.8. Overall, we see a substantial improvement in bandwidth usage as a result of the reduced number of memory stalls. The Pipeline version consumes 2x the amount of bandwidth as the baseline version. Of the seven programs, LUD Diagnol and LUD Internal use the most bandwidth, while Gaussian uses the least.

5.8.0.4 FPGA vs GPU Performance Comparison

In this part, we compare GPU and CPU performance for all of our applications. We ported the Xilinx FPGA OpenCL codes(vs. 2017.4) and made them work for the GPU version suited for running on our local AMD FirePro W7100 GPU. We specifically rewrote the entire host code written for Xilinx FPGAs while keeping the kernel code the same for individual applications. For the implementation part, we used the generic OpenCL APIs and used AMD C++ bindings. We obtained the CPU numbers from the Xilinx SDAccel tool.

We compare GPU and CPU output for all of our applications in this section. We adapted the Xilinx FPGA OpenCL codes (vs.2017.4) to run on the AMD FirePro W7100 GPU we had on hand. We completely rewrote the host code for Xilinx FPGAs while leaving the kernel code for individual applications the same. We used the common OpenCL APIs and AMD C++ bindings for the implementation.

Table 5.6: GPU vs FPGA performance comparison

Application	**Timing results(ms)**		
	FPGA Baseline	**FPGA best**	**GPU**
NN	5.68	1.88	0.3
SRAD Extract	319.57	109	70
Gaussian	3681.57	603	406
Hotspot	47.63	12.7	2.0
BFS	2.498	0.62	0.9
LUD Diag	7.47	0.49	2.06
LUD Internal	0.2	0.2	0.01

Finally, we compare the performance numbers obtained across FPGA and GPU (Table 6.3). Except for a few applications where GPU outperforms FPGA by a significant margin, GPU efficiency is comparable to FPGA figures. This shows that, considering their bandwidth limitations, FPGAs behave similarly to GPUs.

5.9 Conclusion

Memory wall is a major hindrance affecting FPGA performance especially when running massively parallel applications with large workload. The crux of this research revolves around mitigating this crucial issue by hiding the memory latency. The approach we present is twofold in nature. First, we introduce a novel LLVM based automation tool to successfully identify and segregate between statically and run-time decouplable data. We work on the gained insights to present an efficient architecture that involves the OpenCL pipe construct to decouple memory access and computation. Thereby we implement a spatial parallelism approach on top of the existing pipelined parallelism of FPGAs and maximize performance and improve power efficiency.

For successful validation of our approach, we test our algorithm on two separate FPGA candidates (edge and cloud) across completely different synthesis tools, applications and vendors. We further attack the programming challenges while porting our design and suggest a complete generic framework that can be easily adapted across the board. The overall performance benefits verifies our claim and we successfully mitigate the problem of memory bottleneck.

CHAPTER 6: Throughput-Oriented Design for Cloud FPGAs

6.1 Introduction

The final contribution of this dissertation is laid out in the following pages of this chapter. The focus in this part is to improve the throughput of reconfigurable devices of AWS cloud FPGAs.We explore a much larger, higher bandwidth capable Xilinx based FPGA and the SDAccel HLS toolchain. Our approach is similar to the previous contributions, in it we look for pressing issues effecting FPGA performance and propose novel solutions to counter the same. This work can be categorized in two main parts,

1. Identify the challenges in the existing Compute Unit (CU) replication optimization technique. We systematically solve the problems and deliver a generic framework and an automation tool to exploit CU replication to the fullest.

2. Explore a couple of software solutions (Double Data Rate (DDR) and Burst transfer) that result in improving the memory access parallelism by addressing efficient traffic between FPGAs global and local memory.

This project has been funded and supported by Xilinx Inc., as a part of research grant for Xilinx University Program (XUP).

6.2 Design Automation for Compute Parallelism

The goal of this project is to investigate the scalability of OpenCL coarse-grain parallelism on cloud FPGAs using *Compute Unite (CU)*. We show that for each application, there is an optimal number of CUs to achieve the best output in terms of memory bandwidth, memory conflicts caused by CU duplication, and usable FPGA resources. At the same time, this work includes a source-code template and a front-end architecture exploration tool that programmers can use to determine the best CU number for a given application while hiding the programming and exploration difficulties.

6.2.1 Compute Unit Replication on Xilinx Platform

Compute unit(CU) replication for the Xilinx FPGAs is a task based parallelism approach [23, 74, 75] implemented on top of temporal parallelism inherent to the OpenCL FPGA semantic [91, 92]. Multiple CUs can be instantiated utilizing different DDR banks for chip-global memory communication. Fig 6.1 shows the architecture semantic of multiple CUs for Xilinx FPGAs. We have dealt with CU replication in great detail in Chapter 4. This section outlines 3 fundamental problems in CU replication and aims to mitigate them in the coming sections.

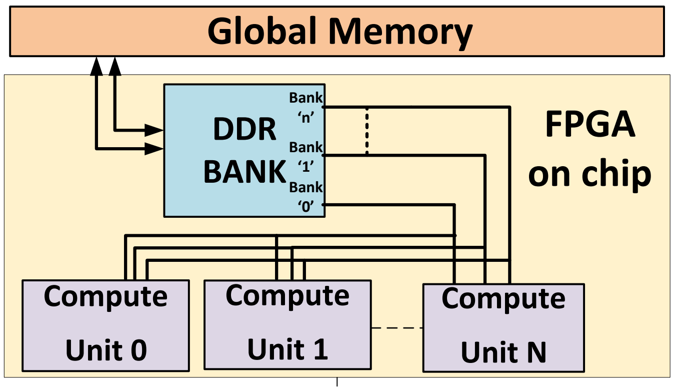


Figure 6.1: Multiple Compute Unit on Xilinx platform

6.2.2 Hard problem of CU replication

CU Replicating comes with its own set of challenges of difficulties. To begin with, CU replication is an NP Hard problem due to the numerous variables involved (such as FPGA tools, work-group sizing, and additional costs associated with CU setup). Unintelligent application of several CUs at the same time does not boost efficiency because over splitting the job size results in redundant CU configuration costs, which can either maintain or increase the kernel execution time. As a result, programmers must take responsibility, as it is impossible to even make an informed guess in such situations.

Furthermore, Xilinx employs the AXI interconnect, limiting the number of master/slave interfaces for kernels and the interconnect connected to the memory controller to ten. As a result, there's still a risk of exhausting FPGA capital (Resource utilization and Bandwidth). Last but not least, the replication of OpenCL CUs on Xilinx FPGAs presents computing difficulties to the average programmer who employ a 'trial and error' strategy to determine the optimum number of compute units for each application.

Listing 6.1: Generic template of kernel

```
__kernel void template(__global const float *var_1,
                        __global const float *var_2,
                         __global float *var_n){
int global_id = get_global_id(0);
int size = Y_SIZE/get_global_size(0);
    for(y=global_id*size; y<(global_id+1)*size; y++)
    {
    }
}
```

6.3 Generic template for CU replication

Listing 6.2: Generic template of host

```
#include <iostream.h>
................
int main(int argc, char* argv[])
{
................
cl::NDRange global_size = WORK_GROUP;
cl::NDRange local_size = 1;
q.enqueueNDRangeKernel(krnl, 0, global_size, local_size, NULL, NULL)
    ;
................
}
```

This section introduces our generic template for CU replication. The template goes over (listing 6.1 for kernel changes) and (listing 6.2 for host file changes) to be made on any application.

Each work-group is mapped to each CU [22] with every given program mapped into Xilinx FPGAs. As a result, we divide the kernel's outer loop, Y_SIZE, by the total global job size, which equals the number of CUs (Listing 6.1).

Since the work items are pipelined within the kernel using the xcl_ pipeline loop pragma given by Xilinx SDAccel optimization, the local work size in the host code is kept at 1. As a result, throughput and consistency improve. The NDRange OpenCL API call specifies both the local and global job scale. As a result, every OpenCL kernel (two-dimensional or three-dimensional) can be conveniently broken up using the prototype.

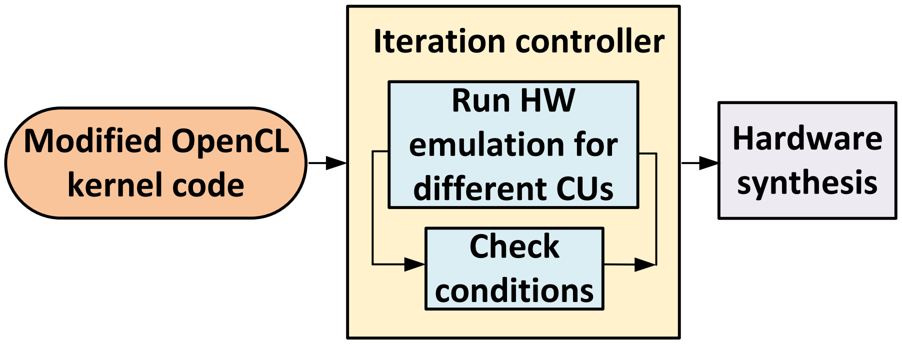

Modified OpenCL kernel code
Iteration controller
Run HW emulation for different CUs
Check conditions
Hardware synthesis

Figure 6.2: Tool flow

6.4 Automation tool for CU replication

Our proposed automation tool for CU replication is presented in Algorithm 2. The updated OpenCL kernel code (based on the generic template) is the tool's primary input, which is initialized through the command line by specifying the number of CUs(WG) to be used (line 2). The tool flow diagram in Figure 6.2 outlines our proposed algorithm.

At the heart of our tool is the iteration controller(Figure 6.2). We choose Hardware (HW) emulation for performing different check conditions. We set a multiplication factor of *0.75* to extract the maximum parallelism without maxing out the resources. This number is purely a programmer's choice. The next component of the controller is *Delta* (Δ)=t (line 4) which is an adaptive value that accounts for difference between the current and previous CU execution times. Delta (Δ) allows for 'automatic jump through iterations' i.e, depending on present Delta (Δ) value iterations across number of CUs can be unevenly incremented. This allows for a more complex, quicker exploration process with much lesser iterations.

In the algorithm, the iteration controller concurrently tests for the following two conditions:-

1. If Delta (Δ) is <t, begin the iteration jump; otherwise, pause.

2. Begin iterating if x(t) is <0.75*x(t-1) (where x(t) is the present value of CU execution time and x(t-1) is the previous), otherwise end.

Based on resource consumption and bandwidth, the maximum amount of CU per program can vary. Having a large number of CUs increases the likelihood of resource exhaustion and lengthens the synthesis cycle. Until the optimum CU number is created, the TIMING (line 15-20) and RESOURCE (line 21-25) checks are run in parallel.

6.5 Experiments

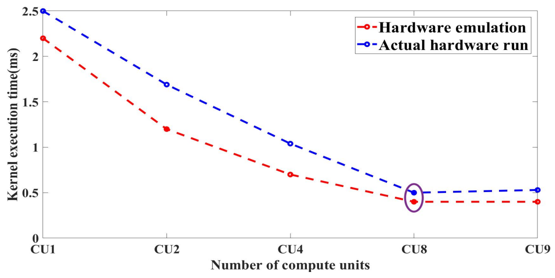
For compiling and synthesizing OpenCL code, we use Xilinx SDAccel HLS for OpenCL [93] software. The SDAccel profiler records kernel performance, global memory bandwidth efficiency, resource utilization, and power consumption. We also report our GPU results for kernel performance and power details using the AMD FirePro

Algorithm 2 Compute Unit Replication Tool

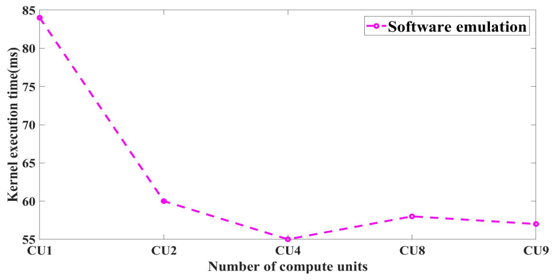
```
1: function MAIN()
2:     Initialize Compute Unit(WG), Execution Argument(nk)
3:     Validate project directory path
4:     Set Delta(Δ) = Variable t>0
5:     while true do
6:         WG ← parse input host file
7:         Run Hardware Emulation
8:         if TIMING or RESOURCES then
9:             Terminate loop
10:        end if
11:    end while
12:    Execute system synthesis
13:    Store results
14: end function

15: function TIMING()
16:     $y = f(x) \leftarrow$ parse profile summary file
17:     if $(y = f(x) = x(t) < 0.75 * x(t-1)$ then
18:         $y = f(x) = x(t)$
19:     end if
20: end function

21: function RESOURCES()
22:     $y = f(x) \leftarrow$ parse profile summary file
23:     if (LUT or LUTMem or REG or BRAM or BW)<100 then
24:         $y = f(x) = x(t)$
25:     end if
26: end function
```



(a) Hardware emulation results



(b) Software emulation results

Figure 6.3: Histogram application results for tool validation

Table 6.1: Baseline profiling information for each application

Benchmarks	Class of application	Baseline optimization	Memory access pattern	Resource utilization(%)				Execution Time (ms)	Average Bandwidth (%)	FPGA Power (W)	GPU Power (W)
				LUTs	LUTMem	REG	BRAM				
Affine	Image processing	A, B	Irregular	1.18	0.25	0.79	0.75	8.04	39.09	35.1	18.4
AES	Security	C, D	Regular	0.66	0.26	0.36	2.06	2.21	100	34.81	18.2
Median Filter	Digital filtering	A, B, E	Irregular	0.74	0.26	0.5	3.05	1.14	6.79	35.01	19.2
Histogram	Image processing	A, C	Regular	4	0.38	2.27	8.2	2.52	96	36.68	19.7
Tiny Encryption	Security	A, B, C	Regular	4.86	0.18	2.24	3.66	7.33	59.1	36.9	18
Watermarking	Image processing	A, E, F	Regular	0.45	0.25	0.31	0.91	0.29	81	37	19.6
Systolic Array	Array architecture	A, C, G	Regular	2.65	0.73	1.1	0.09	0.4	0.51	35.14	18.8
Large Loop OCL	Convolution layer	A, G	Regular	0.56	0.22	0.48	21.72	2391	100	39.08	20.6
Nearest Neighbor	Data mining	A	Regular	0.37	0.22	0.25	0.09	5.68	9.6	34.9	17
LUD Diag	Linear Algebra	A, G	Irregular	0.34	0.19	0.2	0.23	7.74	16.9	35	18
SRAD Extract	Image processing	A	Regular	0.33	0.19	0.16	0.05	316.5	56.7	35	19.2
Gaussian Fan1	Linear Algebra	A	Regular	0.84	0.44	0.46	0.09	1.52	24.9	34.7	18
Gaussian Fan2	Linear Algebra	A	Regular	0.87	0.36	0.56	0.19	0.2	19.7	35	18.3
Hotspot	Physics Simulation	A, C, E	Regular	0.69	0.21	0.37	8.06	47.63	4.58	34	18.2
BFS	Graph Algorithms	A, E	Irregular	0.73	0.36	0.45	0.19	2.49	9.6	35	20.6

W7100 GPU and the AMD CodeXL Power profiler.

For our studies, we used 15 massively parallel programs (Table 6.1 from the Xilinx SDAccel repository [94] and the Rodinia Benchmark suite [31]. This allows for a reasonable comparison of output between single work item kernels optimized for FPGAs and multiple work item kernels optimized for GPUs. In Table 6.1, we can see the baseline profiling statistics.

6.6 Discussion

6.6.0.1 Tool Validation

In this segment, we choose a *histogram* program at random and analyze it using hardware (HW) + software (SW) emulation and real hardware on the FPGA. Due to the time complexity of synthesis, we restrict our number of experiments. Following that, we'll go through the three facets of tool validation that we'll be talking about.

1. *Provide reasoning for Hardware over Software emulation*- The execution time of HW emulation and machine runs is shown in Figure 6.3a, while the execution time of SW emulation is shown in Figure 6.3b. Both graphs demonstrate the discrepancy in timing details. While the hardware emulation is accurate, the machine emulation is quick but inconsistent. For larger kernels and data, however, HW emulation has a longer compile period. However, when considering the accuracy criteria, this is a trade-off that we feel.

Another disadvantage of HW emulation is that logic optimizations performed by Synthesis and Place & Route will minimize necessary capital, allowing us to put in more CUs than emulation indicates. However, since we almost never exceed the resource quota, other considerations such as exceeding maximum bandwidth are the primary efficiency bottleneck.

2. *Show one complete design space exploration to verify our tool results*- HW emulation (Fig 6.3a) shows the optimal speed up with 8 CUs. We validate this by running an entire design space from CU1 (baseline) through CU9[1] and observe that the actual system run also shows the maximum speed up at CU8 after which the performance degrades. CU8 is therefore the most optimal solution.

3. *Timing information*-Finally, we report the synthesis time for each iteration of CUs (Table 6.2) for the same application. We discuss the implications of these results later in detail.

Table 6.2: Design space exploration of Histogram application

Number of Compute Units(CUs)	1	2	4	8	9
Synthesis time(mins)	87	127	191	330	360
Execution time(milli secs)	2.52	1.69	1.04	0.5	0.53

6.6.0.2 Performance Evaluation

We analyze the performance of our applications in two parts:-

First, we see the relative performance improvement of benchmarks over the baseline implementation for the CU(CU[N]) that gives the maximum speedup in Figure 6.12. We get a maximum performance improvement of 53.1X over baseline implementation for the LUD Diag application and a 6.4X on average. However, we either get a constant or very little speed up for few of the applications like AES, Watermarking, Gaussian Fan1 etc., This can be attributed to 3 factors-baseline execution time, optimizations, bandwidth usage.

As an illustration, the Gaussian Fan2 application's baseline performance numbers

[1]CU10 not synthesized due to resources maxing out

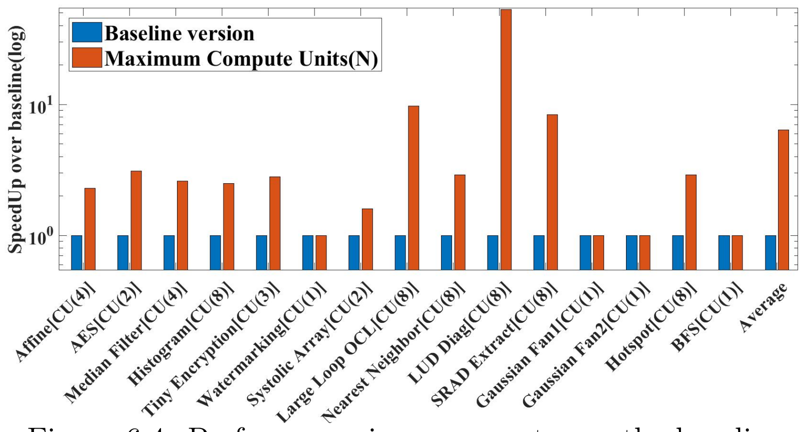


Figure 6.4: Performance improvement over the baseline

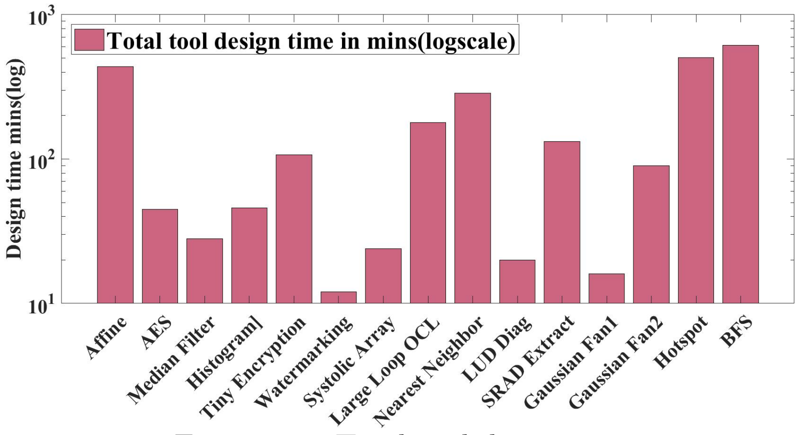


Figure 6.5: Total tool design time

are the lowest. The scheduler will take the least number of CU during the run time since using a CU will create an overhead of calling the host which is counter productive to its performance.

LUD Diag on one hand has low bandwidth utilization (Table 6.1) and uses burst transfer (Table 6.1) where the data transfer happens in larger chunks and therefore benefits the most (21X improvement) from this approach. AES on the other hand has a very high baseline bandwidth utilization Table 6.1. This makes bandwidth a limiting factor hindering its ability to improve speedup.

Next, we observe the total design time that our tool takes to reach the optimal CU number for maximum speedup in Figure 6.5. The total design time for the tool to reach the optimal number of CUs is given as in Eq. (6.1):-

$$Total D_{\mathrm{T}} = \sum_{i=1}^{N} D_{\mathrm{T}}(CU) + D_{\mathrm{T}}(CU_{\mathrm{N+1}}) \tag{6.1}$$

where,

$\mathrm{D_T}$(CU) shows the tool design time for each iteration of compute unit.

We correlate this with the design time of the Histogram application (Figure 6.5) that takes 46m30s and additional synthesis time (for max CU[8]) of 300m totalling 346m30s. In contrast, the total design space exploration time (Table 6.2) is 1095m. Our tool achieves over **31%** design time improvement for the Histogram application.

The tool design time ($\mathrm{D_T}$) is a factor of size of the application(dataset), computation demands and memory access patterns. Overall, we observe that applications with regular memory accesses (Table 6.1) run faster since irregular memory accesses cause divergence affecting the tool design time. As an example, the Watermark application takes a mere 12m35s vs the BFS that takes 612m25s of tool design time ($\mathrm{D_T}$).

6.6.0.3 Resource and Power overhead

Resource overhead is mainly introduced due to additional register blocks, memory blocks, combinational logic and block RAMs which are required for replicating CUs that significantly increase every time a new CU is added. Figure 6.13 shows the average percentage resource utilization overhead. On average we measured a 6% increase in LUTs, 4% increase in LUTMem, 8% increase in registers and 8% increase in Block RAMs.

Power results (Figure 6.14) show similar trends like the resource utilization and we observe the maximum power usage for the Large Loop OCL application that shows maximum power usage of 3.6x over baseline owing to its larger BRAM utilization(Table 6.1 and Figure 6.13). However, for most of the applications with very

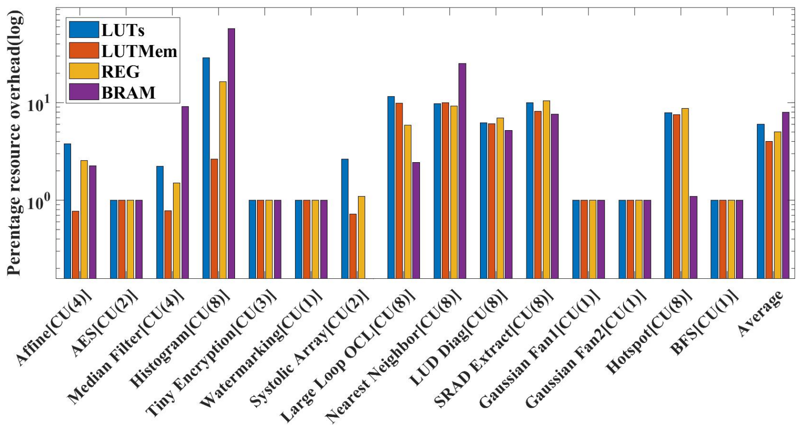


Figure 6.6: Percentage resource utilization overhead over baseline

miniscule baseline resource utilization (Table 6.1), adding more CUs does not affect power as much. The average power overhead was thus reported a mere 1.33X over baseline.

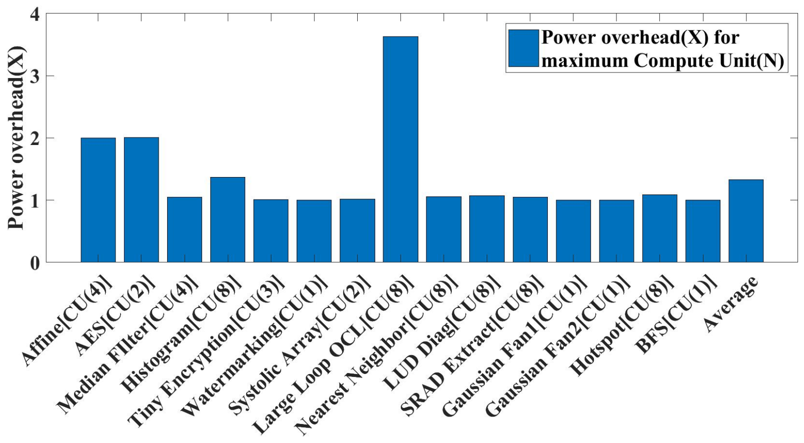


Figure 6.7: Power overhead over baseline

6.6.0.4 Bandwidth Improvement

Bandwidth utilization increase (Figure 6.8) represents the maximum read and write bandwidth improvement that the application can use, thus more the number of CUs more is the bandwidth. This however is different for applications like Large Loop OCL that have a large baseline bandwidth number (Table **??**). Also, with increasing bandwidth we do see a increased speed up- pointing to the fact that more CUs can extract more performance. This is evident from the Hotspot application that saw a 21X rise in bandwidth leading to 2.8X speed-up. On average we observe a 3.8X improvement in bandwidth utilization numbers.

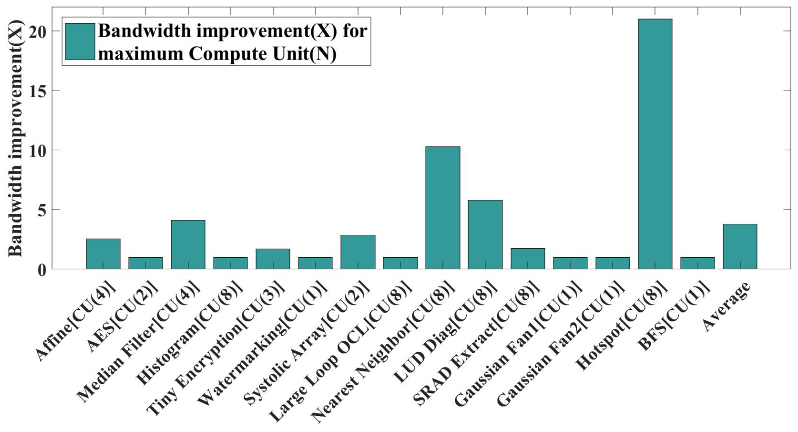


Figure 6.8: Bandwidth improvement over baseline

6.6.0.5 FPGA vs GPU Performance Comparison

In this section (gpufpga) we give a performance perspective for all of our applications. While Rodinia applications [11] were written for GPUs, we ported the rest of applications to GPU version using the generic OpenCL APIs and AMD C++ bindings devoid of any optimizations. We chose to run our applications on the AMD GPU since it has comparable bandwidth [2] to the FPGA.

While our FPGAs best performance beats baseline across most of the applications, GPU performance is comparable to the FPGA numbers except for a few of the applications where GPU beats FPGA by a huge margin. Dynamic power (W) numbers of the applications on both the platforms listed in Table 6.1 are comparable in nature.

This concludes that despite FPGAs being bandwidth limited they perform comparably well alongside GPUs. With an increase in bandwidth utilization capacity FPGAs can surely outperform GPUs in many of the massively parallel applications.

6.7 Memory Access Parallelism on Cloud FPGAs

Software based optimizations enables efficient transfer of OpenCL data between the global and local memory [23]. The host code and the kernel code needs to be there-

Table 6.3: GPU vs FPGA performance comparison

Application	**Timing results(ms)**		
	FPGA Baseline	**FPGA best**	**GPU**
Affine	8.04	2.71	1.05
AES Decrypt	2.21	1.93	0.024
Median Filter	1.14	0.21	0.902
Histogram Equalization	2.52	0.55	87.61
Tiny Encryption	7.33	2.72	1.26
Watermark	0.29	0.29	0.016
Systolic Array	0.40	0.1	0.89
Large loop OCL	2391	244.26	0.0029
Nearest Neighbor	5.68	1.96	0.3
LUD Diag	7.74	0.145	2.06
SRAD Extract	316.5	37.7	70
Gaussian Fan1	1.52	1.52	0.3
Gaussian Fan2	0.20	0.20	0.13
Hotspot	47.63	16.40	2.0
BFS	2.49	2.49	0.9

fore changed for a better data transfer. There are multiple optimizations available for better data transfer that are vendor specific. Double Data Rate and Burst Transfer are some of the data transfer optimizations that are currently introduced in the XIlinx SDAccel HLS toolchain. However, these techniques being recently introduced [22] suffer from programmability challenges. We look at both of these optimization techniques and propose generic framework(s) so as to aid the naive FPGA programmer to apply these on a variety of applications.

Double Data Rate (DDR) is used for the application which needs larger data to be transferred. A large application with compute intensive workload such as Larg[22] is an example of such a data hungry neural net benchmark. The data-transfer efficiency of such an application can be invariably improved by splitting up the global memory inside the kernel to separate DDRs.

6.8 Double Data Rate [DDR] optimization

This section explains how the Xilinx HLS toolchain implements Double Data Rate optimization. To extend DDR optimization to a number of applications, we first formalize different approaches. Next, we'll take a closer look at a sample program

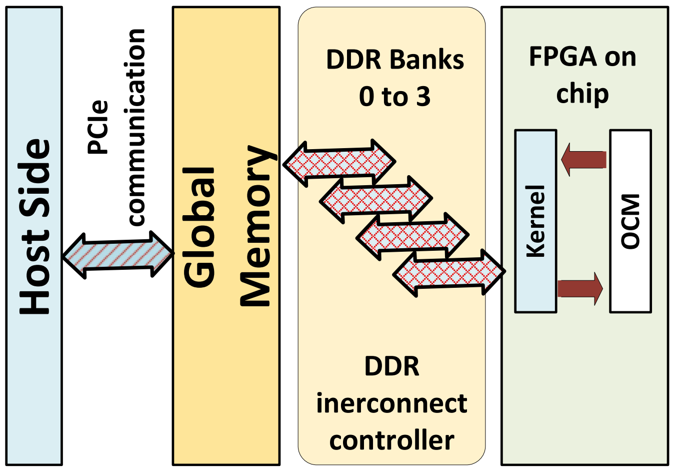


Figure 6.9: DDR banks architecture

called "Watermarking."

Between the global and local memory, DDR synchronization is used to transmit and receive signals twice per clock cycle. However, only one DDR bank is used in the default created data-path (Refer Figure 4.2, Chapter 4). As seen in Figure 6.9, a device with several DDR banks can be targeted so that kernels can access all available memory banks at the same time. By dividing the global memory within the kernel into different DDRs, the data transfer performance can be increased. On the Virtex VU9P FPGA, Xilinx HLS has 4 slots/banks (Bank 0 to Bank 3) that can provide up to 80GB/s raw DDR bandwidth.

It's important to remember that the effectiveness of DDR success is determined by a number of factors, including:

- Memory access pattern- nature of global memory accesses, regular or irregular.
- Memory address pattern- nature of memory transaction, contiguous or non-contiguos.
- Random access pattern- at any given time, the inflow and outflow of data interacting through various banks.

6.8.1 Generic Template

This segment incorporates a standardized DDR optimization template. To use several DDR banks, the user must allocate CL memory buffers to separate banks in the host language, as well as fit the XCL binary file to the branch. These have simply been divided into two generic groups. The template is spread into parts, Listing 6.3 for host side changes and Listing 6.4 for make file changes that can be made for any application.

On the Makefile side, Listing 6.4 provides the default prototype of Make. For each global pointer in the kernel, the max memory ports flag is needed to create an AXI MM interface. The âsp switch is then used to map AXI interface names to each corresponding bank. The âsp switch helps the designer to map kernel ports to individual DDR memory banks using the System Port mapping option.

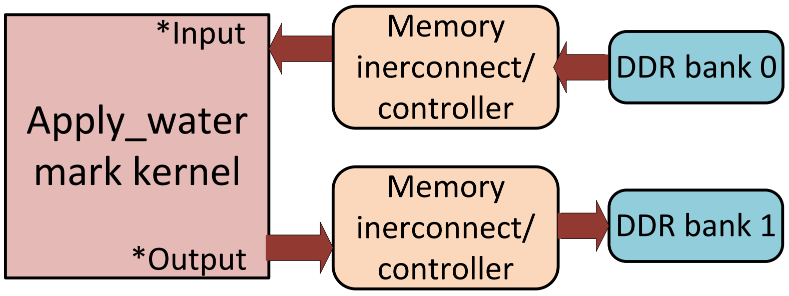

*Input
Apply_water mark kernel
*Output
Memory inerconnect/ controller
Memory inerconnect/ controller
DDR bank 0
DDR bank 1

Figure 6.10: Understanding generic template using the apply watermark kernel

Listing 6.3: Generic template of Hostfile

```
#include <iostream.h>
................
int main(int argc, char* argv[])
{
................
cl::NDRange global_size = WORK_GROUP;
cl::NDRange local_size = 1;
................
cl_mem_ext_ptr_t inExt, outExt;
inExt.flags  = XCL_MEM_DDR_BANK0;
outExt.flags = XCL_MEM_DDR_BANK'N';
inExt.obj = 0   ; outExt.obj = 0;
inExt.param = 0 ; outExt.param = 0;

q.enqueueNDRangeKernel(krnl, 0, global_size, local_size, NULL, NULL)
    ;
................
}
```

Listing 6.4: Generic template of Makefile

```
{
...............
kernelname_CLFLAGS= -I./src --max_memory_ports kernelname
kernelname_LDCLFLAGS=--sp
kernelname_1.m_axi_gmem0:bank0 --sp kernelname_1.m_axi_gmemN:bankN
...............
}
```

6.8.2 Watermark Application

We'll now take a look at the "Watermarking" program as an example and add DDR optimization to it. The watermarking algorithm's main purpose is to overlay a logo at a given position in a 1080p HD video stream. The logo may be active, which is represented by a brief, repeated video clip, or passive, which is usually represented by a still image. The most common technique used by broadcasting companies is passive watermarking, used in this example. The Apply Watermark kernel, which is an efficient implementation of a watermarking algorithm, is at the heart of the program.

The application takes image data and applies watermarking to it. The kernel char-

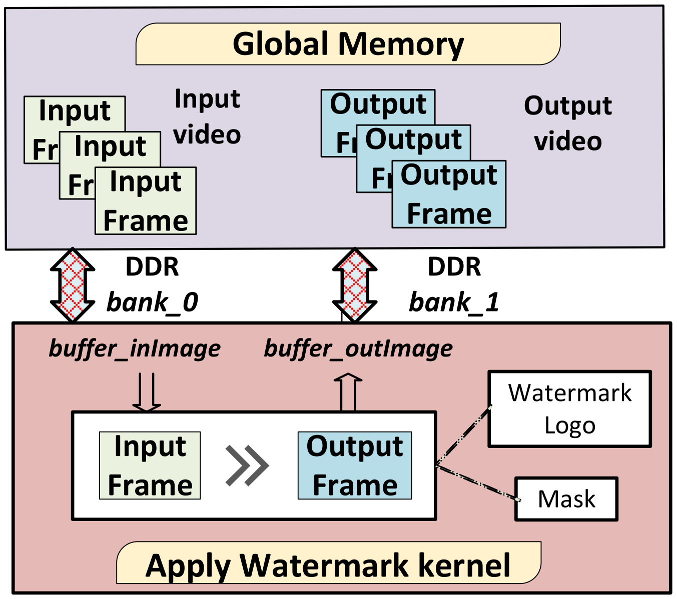


Figure 6.11: DDR 2 banks apply watermark kernel

acteristics are described in Table 6.4. The *buffer_ inImage* and *buffer_ outImage* frames, respectively, are two-dimensional arrays of pixels represented in a three-dimensional color space (YCbCr). Every pixel is expressed by three components: Y stands for luma, Cb for chroma blue-difference, and Cr for chroma red-difference. Each portion is an 8-bit value, giving the pixel a total of 24 bits. The host processor retrieves the input video stream from disk and serially transfers it to the FPGA global memory via a single DDR port in the default FPGA storage line.

The memory access is thus divided between 'N' DDR ports (2 here) by applying the generic DDR optimisation system, As a result, the transmission rate is doubled and the access to local and global memories is increased efficiently.

Table 6.5 shows the output characteristics of the water marking program. The baseline speeds up to 1.76X, while the bandwidth and transmission rate figures rise to 1.5X. This shows that applying DDR optimization in the watermarking program has increased its overall efficiency.

Table 6.4: Watermark Kernel properties.

Kernel name	apply_benchmark
Kernel global arguments	input, output
Read Buffer	buffer_inImage
Write Buffer	buffer_outImage

Table 6.5: Watermark kernel performance numbers.

Combination	**Execution time (ms)**	**Bandwidth utilization (%)**	**Transfer rate (GB/s)**
Baseline	0.53	40.8	4.7
2 DDR banks	0.3	62.87	7.24

6.9 Experiments

This section presents our experimental setup and results. We used a total of 9 massively parallel applications comprising of different classes and access patterns (Table 6.6) from the Xilinx SDAccel repository [22] for our experiments.The baseline profiling information is shown in Table 6.1. We have used the Xilinx SDAccel HLS tool for OpenCL [8] based on OpenCL version 1.0 for compiling the OpenCL code.

All our applications were synthesized on Virtex VU9P FPGA deployed on Amazon AWS cloud. Table 6.7 lists the parameters of our FPGA platform. We use Xililx SDAccel HLS for OpenCL [93] for compiling and synthesizing OpenCL code. The SDAccel profiler collects kernel performance data, bandwidth efficiency of global memory and resource utilization. Finally, we compare naive CPU and GPU implementation with baseline and best case FPGA performance.

For the CPU version of each application we use the Intel(R) Core(TM) i7-7700K as our host system. We also choose the AMD FirePro W7100 GPU since it provides comparable bandwidth as that of our cloud FPGA. To report our GPU implementation results for kernel performance execution time and average dynamic power consumption, we use the AMD CodeXL Power Profiler version 2.5 [80].

Table 6.6: List of Applications.

Application	**Class of application**
Affine	Image processing
Convolve	Convolutional image filtering
Edge detection	Image processing
Histogram	Image processing
Large Loop OCL	Convolution layer of CNN
Median Filter	Non-linear digital filtering
Systolic Array	Array architecture
Tiny encryption	Security
Watermarking	Image processing

Table 6.7: System characteristics used for the study

Host System	Intel(R) Core(TM) i7-7700K			
Host specs	4.2 GHz, 16GB DDR4 Memory			
FPGA Family	Virtex Ultrascale VU9P			
FPGA Device Specs	LUTs	LUT Mem	REG	BRAM
	1157 K	585 K	2330 K	2 K
GPU Device Specs	AMD FirePro W7100, 8GB DDR5 Memory			
	Compute=28, Memory Bandwidth= 160GB/s			

6.10 Discussion

6.10.0.1 Performance Analysis

Figure 6.12 indicates the efficiency of 2 DDR banks that provide maximum speedup relative over baseline implementation. For Systolic Array we achieve an overall performance gain of near 2X over baseline and an average of 1,4X. We are, however, attributable to the three factors mentioned in the Section 6.8, as well as the high initial bandwidth utilization as seen in the Tab;e 6.1.

As an illustration, Systolic array has the least average baseline bandwidth of 0.51% and results in the most (1.96X) improvement.Histogram on the other hand has a very high baseline bandwidth utilization (Table 6.1) and gets the least improvement (1.03X). This makes bandwidth a limiting factor hindering its ability to improve speedup.

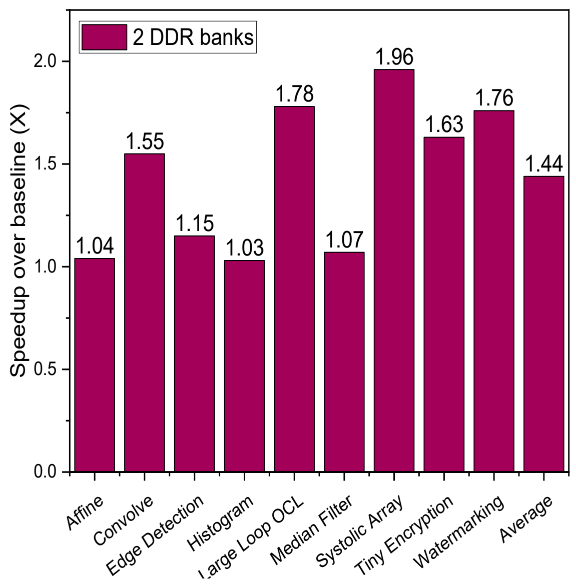


Figure 6.12: Speedup over baseline

6.10.0.2 Resource Overhead

This section briefly presents the resource utilization overhead on account of addition of 2 DDR banks per application. Figure 6.13 shows the average percentage resource utilization. The resource overhead is mainly introduced due to additional register blocks, memory blocks, combinational logic and block RAMs which are required for replicating DDR banks that significantly increase every time a new bank/port is added.The average increase in resource utilization was observed to be a 7.8% increase in LUTs, 5.3% increase in LUTMem, 3.5% increase in registers and 4.2% increase in Block RAMs.

Resource utilization overhead is briefly presented in this part (Figure 6.13). The resource overhead is mainly due to additional register blocks, memory blocks and RAMs that are used to replicate DDR banks which increase when a new bank is added. The overall rise in use of resources has been found to increase the LUT's by 7.8%, LUTME's by 5.3%, the register's by 3.5% and Block RAMs by 4.2%.

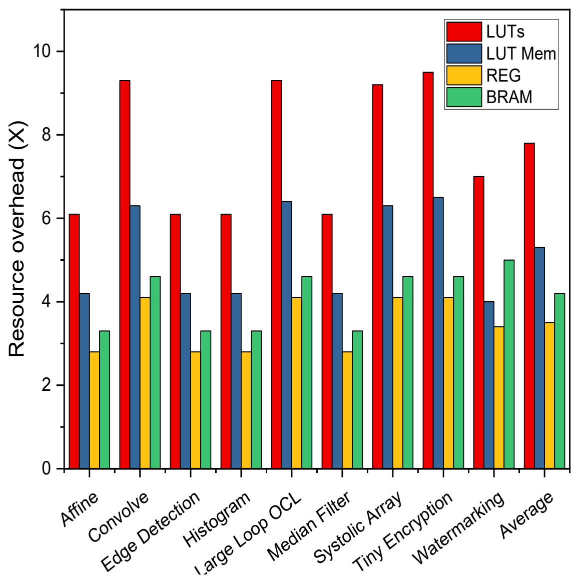


Figure 6.13: Resource utilization overhead over baseline

6.10.0.3 Total Dynamic Power Consumption

Figure 6.14 presents the power consumption of all the applications for 2 DDR banks. The Dynamic power is found from SDAccel power profiler tool that calculates dynamic power dissipated across each of Adaptive Logic Modules(ALMs), RAM Blocks(RAM), DSP blocks(DSP), Phase Lock loops(PLLs), Clock, High Speed Differential I/Os(HSIO) and associated routing modules. The power results show similar trends like the resource utilization, however, the power numbers do not vary a lot. This owing to the fact that low resource overhead was observed for most of the applications.

The figure 6.14 shows all application's power consumption. Dynamic power can be found from individual power component dispensed through each of Adaptive Logic Modul(ALM), RAM (RAM), DSP(DSP), phase loop(PLLs), clock and related routing modules. The effects of power indicate related patterns like that of resource utilization, but the power rates are not so different. This is because the overhead on most applications are limited in terms of resources.

For reference we also report the average baseline power of FPGAs to be around

36W (Table 6.1) while the average power overhead is reported to be a mere 1.06X over baseline. We also obtain an average GPU on-chip power consumption of *18.8W* using the CodeXL Power Profiler version 2.5 [80].

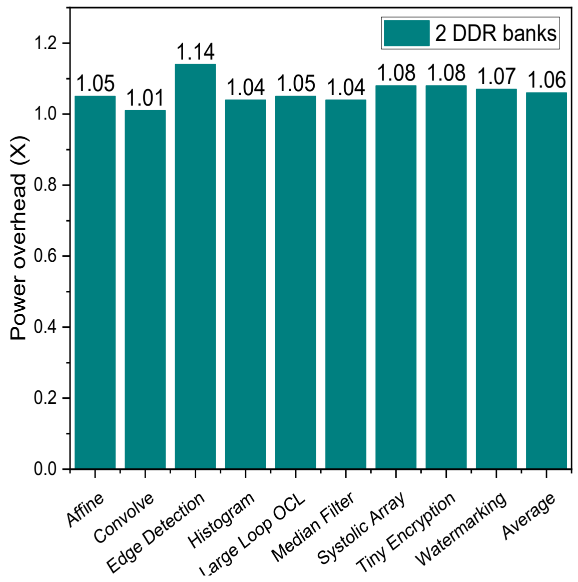


Figure 6.14: Power overhead over baseline

6.10.0.4 Timing Comparison Across Various Platforms

We compare the performance of our applications with GPU and CPU in this section (Table 6.8). We've ported the Xilinx FPGA OpenCL (vs. 2017.4) codes to function on our local AMD FirePro W7100 GPU. As our FPGA has a similar bandwidth, we have decided to run our applications on the AMD GPU [2]. The entire host code of Xilinx FPGAs has been specifically rewritten while the kernel code for different applications is the same. We used generic OpenCL APIs and used AMD C++ bindings for the implementation component. The host CPU has been provided with the CPU numbers. Of course, implementations of the CPU and the GPU are naive device versions without any optimizations whatsoever. The results clearly indicate that FPGA *best* performance comfortably beats *baseline* and CPU numbers across most of the applications while showing comparable performance with the GPU numbers.

Table 6.8: FPGA vs GPU vs CPU comparison

Application	**Timing results(ms)**			
	FPGA Baseline	**FPGA best**	**CPU**	**GPU**
Affine	8.04	7.66	34.35	1.05
Convolve	28.511	18.35	137.09	72.25
Edge detection	0.82	0.74	42.12	73.4
Histogram	2.52	2.51	128.15	0.0078
Large Loop OCL	3385.11	1894.77	766.78	0.0029
Median Filter	1.23	1.14	1.36	0.902
Systolic Array	0.16	0.08	0.24	0.89
Tiny encryption	0.23	0.14	1.69	1.26
Watermarking	0.53	0.303	1.14	0.017

6.11 Burst Transfer [BT] Optimization

In this section we talk about the final optimization technique called, 'Burst transfer[BT]'. Burst transfer works on the principle of transferring chunks of data in single bursts rather than one after another. This helps reduce memory latency while increasing the memory controller's bandwidth consumption and performance. Ideally, burst transfers should be inferred from consecutive data requests from consecutive address sites, if possible. This maximizes memory controller performance and keeps the FPGA device's CU occupied at all times.

The data object's memory structure is an important thing to remember when trying to improve data transfer performance. Consider a 3x3 matrix layout, which is a two-dimensional array in concept, as seen in the matrix logical structure. The burst transition cannot be achieved if the data is been read column-by-column as shown in Figure 6.15 since every time a discrete location is encountered. Arrays are physically stored in row-major order in C/C++ programming, which means that all the data in a row is stored in the same order (Figure 6.16). This is the required memory layout for Burst transfer to happen.

Row/Col	0	1	2
0	a[0][0]	a[0][1]	a[0][2]
1	a[1][0]	a[1][1]	a[1][2]
2	a[2][0]	a[2][1]	a[2][2]

Figure 6.15: Inefficient logical layout, BT not applicable

Address	Location
0	a[0][0]
1	a[0][1]
2	a[0][2]
3	a[1][0]
4	a[1][1]
5	a[1][2]
6	a[2][0]
7	a[2][1]
8	a[2][2]

Figure 6.16: Efficient physical layout, BT applicable

6.11.1 Generic Template

We continue with our tradition to add a generic template that can be standardized for any given application. Note the simple procedure here is two fold:

- Ensure data layout is in contiguos memory locations.
- Determine chunk size of data that is to be transferred to local memory.

Let's consider an example 'kernelread' as shown in Listing 6.5 to put this into perspective. The kernel contains 'N'-bit input 'vect1' which is to be copied to the device memory for computation. We assume that we have made sure the global variable vect1 is consecutively arranged and that will be transferred back-to-back to the local memory buffer variable 'loc'. We use Xilinx pipeline_loop attribute to ensure that the data path is kept occupied at all times. Consecutive memory transfer in burst mode happens in the inner loop. The data is accessed from global memory and transferred to local memory via loops. Each CU is equipped with its own local memory. As a result, each CU can copy the data it needs to process to its local memory. Also an important consideration here is that local memory is limited and depends on vendor specific value.

Listing 6.5: Generic template of adding Burst transfer

```
#define numrows X //Outer loop size of the 2D array matrix
#define rowsize Y //Inner loop size of the 2D array matrix

kernel __attribute__ ((reqd_work_group_size(1, 1, 1)))
void kernelread(
 __global uintN *vect1, // 'vect1' from global memory
          uintN *loc, // 'loc' local memory buffer
 ..
 )
{
 for (int i=0; i < numrows; ++i) {
        __attribute__((xcl_pipeline_loop))
        for (j=0; j < rowsize; ++j) {     //Burst mode transfer
            loc [rowsize*i + j] = vect1[rowsize*i + j];
            }
        )
}
```

6.12 Experiments

Burst transfer is used for hiding memory latency during the data transfer. This is done by loading all the data from global memory to the local memory. The latency for accessing the local memory is lower than accessing global memory.

For Burst Transfer optimization we tested the Affine application for demonstrative purposes using our proposed generic template. The Affine application is a kind of image processing application that linearly maps data points, straight lines and planes[94]. After an affine transformation is complete, sets of parallel lines remain parallel. Affine transformations are often used to compensate for dimensional distortions or deformations caused by non-ideal camera angles. We used the same evaluation setup as in Sec 6.8 to measure Burst Transfer effect on Affine.

6.13 Discussion

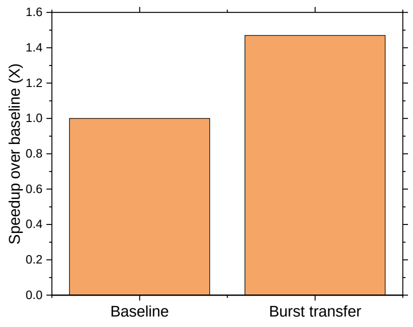


Figure 6.17: Speedup with Burst transfer on Affine application

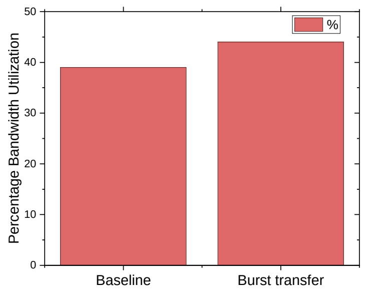


Figure 6.18: Bandwidth improvement with Burst transfer on Affine application

Figure(s) 6.17 and 6.18 shows an improvement of 1.4X and increase in 5% bandwidth utilization over baseline for the Affine application. The increase is again dependent on the memory access nature of the application. However, a combination of BT with DDR (6.8) can highly effect the performance numbers of the application with minimum resource and power overhead.

6.14 Conclusions

Double Data Rate (DDR) and Burst Transfer (BT) are a few optimization technique(s) that can be applied to improve thread-level utilization, performance, and occupancy on FPGAs. The major contribution of this work primarily lies in proposing generic framework(s) for both these techniques based on OpenCL parallelism abstraction and execution semantic of FPGAs. The aim is to provide a formalized template for OpenCL written codes in order to enhance their memory access parallelism efficiency guiding FPGA programmers to better utilize High-Level Synthesis (HLS) tools.

Overall, our results on nine applications of SDAccel benchmark applications indicate that DDR based FPGA-aware OpenCL codes can achieve up to 1.4X maximum throughput while consuming minimum dynamic power and resource utilization overhead compared to baseline implementation. While Burst transfer for one demonstrative application achieved a 1.4X speedup and 5% increase in bandwidth utilization.

CHAPTER 7: Conclusions

This dissertation presents new architectural tools and techniques to enhance the thread-level utilization, performance and occupancy of OpenCL based massively parallel applications when mapped on to FPGAs. The focus is to identify, optimize and make algorithmic decisions that result in a more efficient application specific datapath. We present three main insights from this research. The primary work focuses on proposing a new taxonomy based on the correlation between OpenCL parallelism abstraction and execution semantic of FPGAs. The aim is to leverage the granularity of OpenCL threads that can exploit spatial parallelism on top of temporal parallelism. Further, we formalize OpenCL written codes on FPGAs to enhance their efficiency guiding both OpenCL programmers and OpenCL synthesis tools. Overall, we achieve a maximum speedup of 6.7X over baseline implementation with minimal resource and power overhead.

Next, our proposed LLVM based automation tool for memory decoupling provides the opportunity to prefetch the data of future threads along with the execution of current threads by creating a concurrency model between the computation and memory accesses among the OpenCL threads. This proposed approach uses OpenCL channel semantic to realize splitting of kernels for temporal parallelism, and LLVM static analysis to identify the decouplable data. Our experimental results over eight Rodinia kernels running on Intel Stratix-V FPGA demonstrates an average of 2X speedup with 40% energy reduction and minimum resource utilization overhead compared to baseline implementation. To validate our approach we ran an entirely different set of applications on Xilinx's cloud FPGA. Here, we were able to achieve an average of 5.2X speedup with a 2.2X increase in bandwidth utilization and just under 2.5X resource utilization overhead.

Finally, the dissertation expanded on the throughput oriented design principles on cloud FPGAs and carried out 2 separate projects. Our fully automatic rapid design space exploration tool for identifying the optimum number of Compute Units(CUs) resulted in a a maximum of 53X and 6.4X average speedup over 15 massively parallel OpenCL applications. Further, FPGA aware OpenCL code for increasing the throughput of memory access parallelism using Double Data Rate and Burst Transfer optimization resulted in a performance improvement of 1.2X on average over naive baseline implementation and 1.47X increase in speed up for the Affine application respectively.

Overall, this dissertation opens up a lot of research possibilities on OpenCL for FPGAs while resolving many critical issues that hinder parallelism capabilities on FPGAs.

7.1 Future Directions

Reconfigurable platforms have played a key role in the high-performance computing community. The novel techniques and optimization explorations on the OpenCL programming framework have given us deep insights and opened up many new opportunities for establishing an efficient data-path, better utilization of the resources and maximizing the parallelism potential on deeply pipelined FPGA architectures. We believe this dissertation can result in the following exciting future research directions.

1. With the help of a generalized taxonomy for leveraging spatial parallelism (Chapter 4), OpenCL programmers and tool developers now have a better idea of various source level decisions that can be taken to increase thread level parallelism on FPGAs. One key insight that we obtain from the results is that the parallelism approaches do not affect the applications in the same manner. A much needed trade-off between availability of FPGA resources, timing constraints etc., is imperative to efficient data path design considerations. One

attractive research area that could thus stem from this work is a way to consider these trade-offs and develop a much-sophisticated tool to further improve performance leading to more power-efficient solutions.

2. The validation of our novel memory decoupling approach presented in Chapter 5 to overcome memory stall bottleneck has many new possibilities. The general idea is that applications with most decouplable variables benefit from this approach. Thus the proposed tool and framework can be applied across applications that are streaming in nature with less amount of inter-thread dependencies. Moreover, our LLVM based static analysis tool can be further developed to give more insight on the nature of the application. Such an understanding can truly benefit to categorize applications that can truly benefit from memory decoupling as opposed to other available methods like local memory or double buffering. This also presents an opportunity to finally answer the raging question of what works the best on FPGA vs the GPU.

3. In addition to the generic template for Compute Units (CUs) and a design space exploration tool presented in this research (Chapter 6), there are a lot of profiling and optimization techniques supported by the leading FPGA vendors that we have left unexplored due to the vastness in scope to cover in this work. One among them is the I-I (Initiation Interval), which is of particular interest. I-I values are used as a profiling tool to make sure that the pipelined data path is kept busy at all times. However, this is also a much unexplored territory and significant improvement can be made by incorporating a optimization 'knob' for I-I values.

7.2 Publications

This dissertation has resulted in outcome of the following publications :

1. A. A. Purkayastha, S. A. Shiddhibhavi and H. Tabkhi, "Taxonomy of Spatial

Parallelism on FPGAs for Massively Parallel Applications," 2018 31st IEEE International System-on-Chip Conference (SOCC), Arlington, VA, USA, 2018, pp. 55-60, doi: 10.1109/SOCC.2018.8618501.[PUBLISHED [23]]

2. A. A. Purkayastha, S. Raghavendran, J. Thiagarajan and H. Tabkhi, "Exploring the Efficiency of OpenCL Pipe for Hiding Memory Latency on Cloud FPGAs," 2019 IEEE High Performance Extreme Computing Conference (HPEC), Waltham, MA, USA, 2019, pp. 1-7, doi: 10.1109/HPEC.2019.8916236. [PUBLISHED [95]]

3. Arnab A. Purkayastha, Samuel Rogers, Suhas A. Shiddibhavi, Hamed Tabkhi, LLVM-based automation of memory decoupling for OpenCL applications on FPGAs, Microprocessors and Microsystems, Volume 72, [PUBLISHED [88]] 2020,102909,ISSN 0141-9331,https://doi.org/10.1016/j.micpro.2019.102909.

4. J. Thiagarajan, A. A. Purkayastha, A. Patil and H. Tabkhi, "Exploring the Scalability of OpenCL Coarse Grained Parallelism on Cloud FPGAs," 33rd IEEE International System-on-Chip Conference (SOCC), [ACCEPTED]

5. A. A. Purkayastha and H. Tabkhi, "Design Study on Impact of Memory Access Parallelism for Cloud FPGAs", Submitted to 2021 IEEE 32nd International Conference on Application-specific Systems, Architectures and Processors (ASAP), USA. [Under Review]

REFERENCES

[1] "Implementing fpga design with the opencl standard @ONLINE." `https://www.intel.com/content/dam/www/programmable/us/en/pdfs/literature/wp/wp-01173-opencl.pdf`, Nov. 2011.

[2] "Amd firepro w7100 specifications @ONLINE." `https://www.amd.com/en/products/professional-graphics/firepro-w7100`, 2019.

[3] W. Ma, Y. Ao, C. Yang, and S. Williams, "Solving a trillion unknowns per second with hpgmg on sunway taihulight," *Cluster Computing*, vol. 23, pp. 493–507, 2020.

[4] R. Dennard, F. Gaensslen, H.-N. Yu, V. Rideout, E. Bassous, and A. Leblanc, "Design of ion-implanted mosfet's with very small physical dimensions," *Proceedings of the IEEE*, vol. 87, no. 4, pp. 668–678, 1999.

[5] G. E. Moore, "Cramming more components onto integrated circuits, reprinted from electronics, volume 38, number 8, april 19, 1965, pp.114 ff.," *IEEE Solid-State Circuits Society Newsletter*, vol. 11, no. 3, pp. 33–35, 2006.

[6] "Meet fpga: The tiny, powerful, hackable bit of ... @ONLINE." `https://www.darkreading.com/edge/theedge/meet-fpga-the-tiny-powerful-hackable-bit-of-silicon-at-the-heart-of-iot/b/d-id/1335730`, Oct.

[7] Wise Guy Reports, "Global SRAM FPGA Market Report 2019 - Market Size, Share, Price, Trend and Forecast- WiseGuyReports."

[8] "Intel sdk for opencl applications @ONLINE." `https://www.altera.com/en_US/pdfs/literature/hb/opencl-sdk/aocl_programming_guide.pdf`, 2017.

[9] A. Yilmazer and D. Kaeli, "HQL: A Scalable Synchronization Mechanism for GPUs," in *2013 IEEE 27th International Symposium on Parallel and Distributed Processing*, pp. 475–486, 2013.

[10] Y. Ukidave, C. Kalra, D. Kaeli, P. Mistry, and D. Schaa, "Runtime support for adaptive spatial partitioning and inter-kernel communication on gpus," in *Computer Architecture and High Performance Computing (SBAC-PAD)*, pp. 168–175, 2014.

[11] "Rodinia:accelerating compute-intensive applications with accelerators." `https://www.cs.virginia.edu/~skadron/wiki/rodinia/index.php/Rodinia:Accelerating_Compute-Intensive_Applications_with_Accelerators`.

[12] D. Chen and D. P. Singh, "Fractal video compression in opencl: An evaluation of cpus, gpus, and fpgas as acceleration platforms," in *18th Asia and South Pacific Design Automation Conference*, 2013.

[13] J. Andrade, G. FalcÃ£o, V. Silva, and K. Kasai, "Flexible non-binary ldpc decoding on fpgas," in *IEEE International Conf. on Acoustics, Speech, and Signal Processing - ICASSP*, vol. 1, pp. 1–5, 2014.

[14] H. R. Zohouri, N. Maruyama, A. Smith, M. Matsuda, and S. Matsuoka, "Evaluating and optimizing opencl kernels for high performance computing with fpgas," in *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis, SC 2016, Salt Lake City, UT, USA, November 13-18, 2016*, 2016.

[15] V. Adhinarayanan, T. Koehn, K. Kepa, W.-c. Feng, and P. Athanas, "On the performance and energy efficiency of fpgas and gpus for polyphase channelization," in *2014 International Conference on ReConFigurable Computing and FPGAs (ReConFig14)*, pp. 1–7, IEEE, 2014.

[16] D. Yang, J. Sun, J. Lee, G. Liang, D. D. Jenkins, G. D. Peterson, and H. Li, "Performance comparison of cholesky decomposition on gpus and fpgas," in *Symposium on Application Accelerators in High Performance Computing*, 2010.

[17] O. Segal, N. Nasiri, M. Margala, and W. Vanderbauwhede, "High level programming of fpgas for HPC and data centric applications," in *IEEE High Performance Extreme Computing Conference, HPEC 2014, Waltham, MA, USA, September 9-11, 2014*, 2014.

[18] S. Lee, J. Kim, and J. S. Vetter, "Openacc to fpga: A framework for directive-based high-performance reconfigurable computing," in *2016 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, pp. 544–554, May 2016.

[19] M. J. Flynn, "Some computer organizations and their effectiveness," *IEEE Trans. Comput.*, pp. 948–960, Sept. 1972.

[20] D. B. ASkillicorn, "A taxonomy for computer architectures," (Los Alamitos, CA, USA), IEEE, IEEE Computer Society Press, 1988.

[21] C. D. C. Ralph Duncan, "A survev of parallel computer architectures," *IEEE Computer Society*, Feb. 1990.

[22] "Xilinx opencl." `http://www.xilinx.com/products/design-tools/software-zone/sdaccel.html`, 2015.

[23] A. A. Purkayastha, S. A. Shiddhibhavi, and H. Tabkhi, "Taxonomy of spatial parallelism on fpgas for massively parallel applications," (Washington DC), IEEE, International System on Chip Conference (SoCC), 2018.

[24] P. Bose, *Power Wall*, pp. 1593–1608. Boston, MA: Springer US, 2011.

[25] W. A. Wulf and S. A. McKee, "Hitting the memory wall: Implications of the obvious," *SIGARCH Comput. Archit. News*, vol. 23, p. 20â24, Mar. 1995.

[26] "The opencl specification @ONLINE." `https://www.khronos.org/registry/OpenCL/specs/opencl-1.2.pdf`, 2008.

[27] C. Grozea, Z. Bankovic, and P. Laskov, "Fpga vs. multi-core cpus vs. gpus: hands-on experience with a sorting application," in *Facing the multicore-challenge*, pp. 105–117, Springer, 2010.

[28] J. He, M. Lu, and B. He, "Revisiting Co-Processing for Hash Joins on the Coupled CPU-GPU Architecture," *ArXiv e-prints*, July 2013.

[29] J. He, S. Zhang, and B. He, "In-cache query co-processing on coupled CPU-GPU architectures," *PVLDB*, vol. 8, no. 4, pp. 329–340, 2014.

[30] Q. Gautier, A. Shearer, J. Matai, D. Richmond, P. Meng, and R. Kastner, "Real-time 3d reconstruction for fpgas: A case study for evaluating the performance, area, and programmability trade-offs of the altera opencl," in *International Conference on Field-Programmable Technology (FPT)*, 2014.

[31] 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 *Proceedings of the 2009 IEEE International Symposium on Workload Characterization (IISWC)*, IISWC '09, 2009.

[32] C. E. Laforest, Z. Li, T. O'rourke, M. G. Liu, and J. G. Steffan, "Composing multi-ported memories on fpgas," *ACM Trans. Reconfigurable Technol. Syst.*, vol. 7, Sept. 2014.

[33] D. Weller, F. Oboril, D. Lukarski, J. Becker, and M. B. Tahoori, "Energy efficient scientific computing on fpgas using opencl," in *Proceedings of the 2017 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays, FPGA 2017, Monterey, CA, USA, February 22-24, 2017*, 2017.

[34] Y. Ukidave, C. Kalra, D. R. Kaeli, P. Mistry, and D. Schaa, "Runtime support for adaptive spatial partitioning and inter-kernel communication on gpus," in *26th IEEE International Symposium on Computer Architecture and High Performance Computing, SBAC-PAD 2014, Paris, France, October 22-24, 2014*, 2014.

[35] A. Momeni, H. Tabkhi, Y. Ukidave, G. Schirner, and D. R. Kaeli, "Exploring the efficiency of the opencl pipe semantic on an FPGA," *SIGARCH Computer Architecture News*, 2015.

[36] U. Aydonat, S. O'Connell, D. Capalija, A. C. Ling, and G. R. Chiu, "An opencl(tm) deep learning accelerator on arria 10," *CoRR*, 2017.

[37] A. Putnam, A. M. Caulfield, E. S. Chung, D. Chiou, K. Constantinides, J. Demme, H. Esmaeilzadeh, J. Fowers, G. P. Gopal, J. Gray, M. Haselman, S. Hauck, S. Heil, A. Hormati, J. Kim, S. Lanka, J. R. Larus, E. Peterson,

S. Pope, A. Smith, J. Thong, P. Y. Xiao, and D. Burger, "A reconfigurable fabric for accelerating large-scale datacenter services," *Commun. ACM*, 2016.

[38] Z. Wang, B. He, W. Zhang, and S. Jiang, "A performance analysis framework for optimizing opencl applications on fpgas," in *2016 IEEE International Symposium on High Performance Computer Architecture, HPCA 2016, Barcelona, Spain, March 12-16, 2016*, 2016.

[39] J. Zhang and J. Li, "Improving the performance of opencl-based FPGA accelerator for convolutional neural network," in *Proceedings of the 2017 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays, FPGA 2017, Monterey, CA, USA, February 22-24, 2017*, 2017.

[40] X. Wei, C. H. Yu, P. Zhang, Y. Chen, Y. Wang, H. Hu, Y. Liang, and J. Cong, "Automated systolic array architecture synthesis for high throughput CNN inference on fpgas," in *Proceedings of the 54th Annual Design Automation Conference, DAC 2017, Austin, TX, USA, June 18-22, 2017*, 2017.

[41] M. Z. Hasan and S. G. Sotirios, "Customized kernel execution on reconfigurable hardware for embedded applications," *Microprocessors and Microsystems*, 2009.

[42] M. Tan, B. Liu, S. Dai, and Z. Zhang, "Multithreaded pipeline synthesis for data-parallel kernels," in *Proceedings of the 2014 IEEE/ACM International Conference on Computer-Aided Design*, ICCAD '14, 2014.

[43] R. J. Halstead, J. Villarreal, and W. Najjar, "Exploring irregular memory accesses on fpgas," in *Proceedings of the 1st Workshop on Irregular Applications: Architectures and Algorithms*, IA3 '11, ACM, 2011.

[44] R. J. Halstead and W. Najjar, "Compiled multithreaded data paths on fpgas for dynamic workloads," in *Proceedings of the 2013 International Conference on Compilers, Architectures and Synthesis for Embedded Systems*, CASES '13, 2013.

[45] M. Tan, G. Liu, R. Zhao, S. Dai, and Z. Zhang, "Elasticflow: A complexity-effective approach for pipelining irregular loop nests," in *Proceedings of the IEEE/ACM International Conference on Computer-Aided Design*, ICCAD '15, 2015.

[46] M. Z. Hasan and S. G. Ziavras, "Customized kernel execution on reconfigurable hardware for embedded applications," *Microprocessors and Microsystems - Embedded Hardware Design*, 2009.

[47] E. Nurvitadhi, J. C. Hoe, S. Lu, and T. Kam, "Automatic multithreaded pipeline synthesis from transactional datapath specifications," in *Proceedings of the 47th Design Automation Conference, DAC 2010, Anaheim, California, USA, July 13-18, 2010*, 2010.

[48] R. J. Halstead and W. A. Najjar, “Compiled multithreaded data paths on fpgas for dynamic workloads,” in *International Conference on Compilers, Architecture and Synthesis for Embedded Systems, CASES 2013, Montreal, QC, Canada, September 29 - October 4, 2013*, 2013.

[49] S. P. Vanderwiel and D. J. Lilja, “Data prefetch mechanisms,” *ACM Comput. Surv.*, vol. 32, p. 174â199, June 2000.

[50] M. Shevgoor, S. Koladiya, R. Balasubramonian, C. Wilkerson, S. H. Pugsley, and Z. Chishti, “Efficiently prefetching complex address patterns,” in *Proceedings of the 48th International Symposium on Microarchitecture*, MICRO-48, 2015.

[51] B. Panda and S. Balachandran, “Hardware prefetchers for emerging parallel applications,” in *Proceedings of the 21st International Conference on Parallel Architectures and Compilation Techniques*, PACT ’12, 2012.

[52] T. Kim, D. Zhao, and A. V. Veidenbaum, “Multiple stream tracker: A new hardware stride prefetcher,” in *Proceedings of the 11th ACM Conference on Computing Frontiers*, CF ’14, 2014.

[53] S. Iacobovici, L. Spracklen, S. Kadambi, Y. Chou, and S. G. Abraham, “Effective stream-based and execution-based data prefetching,” in *Proceedings of the 18th Annual International Conference on Supercomputing*, ICS ’04, 2004.

[54] D. Bernstein, D. Cohen, and A. Freund, “Compiler techniques for data prefetching on the powerpc,” in *Proceedings of the IFIP WG10.3 Working Conference on Parallel Architectures and Compilation Techniques*, PACT ’95, 1995.

[55] G. Marin, C. McCurdy, and J. S. Vetter, “Diagnosis and optimization of application prefetching performance,” in *Proceedings of the 27th International ACM Conference on International Conference on Supercomputing*, ICS ’13, 2013.

[56] A. C. Klaiber and H. M. Levy, “An architecture for software-controlled data prefetching,” *SIGARCH Comput. Archit. News*, 1991.

[57] D. F. Zucker, R. B. Lee, and M. J. Flynn, “Hardware and software cache prefetching techniques for mpeg benchmarks,” *IEEE Transactions on Circuits and Systems for Video Technology*, 2000.

[58] N. P. Jouppi, “Improving direct-mapped cache performance by the addition of a small fully-associative cache and prefetch buffers,” in *[1990] Proceedings. The 17th Annual International Symposium on Computer Architecture*, 1990.

[59] J. Lu, H. Chen, R. Fu, W.-C. Hsu, B. Othmer, P.-C. Yew, and D.-Y. Chen, “The performance of runtime data cache prefetching in a dynamic optimization system,” in *Proceedings of the 36th Annual IEEE/ACM International Symposium on Microarchitecture*, 2003.

[60] E. Nurvitadhi, J. C. Hoe, S.-L. L. Lu, and T. Kam, “Automatic multithreaded pipeline synthesis from transactional datapath specifications,” in *Proceedings of the 47th Design Automation Conference*, DAC ’10, ACM, 2010.

[61] K. Turkington, G. A. Constantinides, K. Masselos, and P. Y. K. Cheung, “Outer loop pipelining for application specific datapaths in fpgas,” *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, 2008.

[62] H. Rong, Z. Tang, R. Govindarajan, A. Douillet, and G. R. Gao, “Single-dimension software pipelining for multi-dimensional loops,” in *Code Generation and Optimization, 2004. CGO 2004. International Symposium on*, 2004.

[63] J. E. Smith, “Decoupled access/execute computer architectures,” in *25 Years ISCA: Retrospectives and Reprints*, pp. 231–238, ACM, 1998.

[64] T. Chen and G. E. Suh, “Efficient data supply for hardware accelerators with prefetching and access/execute decoupling,” in *The 49th Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO-49, (Piscataway, NJ, USA), pp. 46:1–46:12, IEEE Press, 2016.

[65] S. Cheng and J. Wawrzynek, “Architectural synthesis of computational pipelines with decoupled memory access,” in *FPT*, pp. 83–90, IEEE, 2014.

[66] B. Betkaoui, Y. Wang, D. B. Thomas, and W. Luk, “A reconfigurable computing approach for efficient and scalable parallel graph exploration,” in *23rd IEEE International Conference on Application-Specific Systems, Architectures and Processors, ASAP 2012, Delft, The Netherlands, July 9-11, 2012*, 2012.

[67] J. Zhang, H. Tabkhi, and G. Schirner, “Ds-dse: Domain-specific design space exploration for streaming applications,” in *2018 Design, Automation Test in Europe Conference Exhibition (DATE)*, pp. 165–170, March 2018.

[68] H. Tabkhi, R. Bushey, and G. Schirner, “Function-level processor (flp): Raising efficiency by operating at function granularity for market-oriented mpsoc,” in *2014 IEEE 25th International Conference on Application-Specific Systems, Architectures and Processors*, pp. 121–130, June 2014.

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

[70] “Fpga developer ami @ONLINE.” `https://aws.amazon.com/marketplace/pp/B06VVYBLZZ`, 2015.

[71] “Project brainwave for real-time ai @ONLINE.” `https://www.microsoft.com/en-us/research/blog/microsoft-unveils-project-brainwave/`, 2017.

[72] J. MÃŒller, J. MÃŒller, and R. Tetzlaff, “A new high-speed real-time video processing platform,” in *2014 14th International Workshop on Cellular Nanoscale Networks and their Applications (CNNA)*, 2014.

[73] T. Kryjak, M. Komorkiewicz, and M. Gorgon, "Real-time hardware–software embedded vision system for its smart camera implemented in zynq soc," *Journal of Real-Time Image Processing*, 2016.

[74] M. W. Hassan, A. E. Helal, P. M. Athanas, W. Feng, and Y. Y. Hanafy, "Exploring fpga-specific optimizations for irregular opencl applications," in *2018 International Conference on ReConFigurable Computing and FPGAs (ReConFig)*, pp. 1–8, Dec 2018.

[75] J. Cong, Z. Fang, Y. Hao, P. Wei, C. H. Yu, C. Zhang, and P. Zhou, "Best-effort FPGA programming: A few steps can go a long way," *CoRR*, vol. abs/1807.01340, 2018.

[76] M. L. H. W. J. X. S. Z. J. Cong, Z. Fang, "Understanding performance differences of fpgas and gpus," in *International Symposium on Field-Programmable Gate Arrays*, 2018.

[77] K. Wang and J. Nurmi, "Using opencl to rapidly prototype fpga designs," in *2016 IEEE Nordic Circuits and Systems Conference (NORCAS)*, pp. 1–6, Nov 2016.

[78] "Intel powerplay early power estimator user guide @ONLINE." `https://www.altera.com/en_US/pdfs/literature/ug/ug_epe.pdf`, year = 2017.

[79] V. J. Reddi, A. Settle, D. A. Connors, and R. S. Cohn, "Pin: A binary instrumentation tool for computer architecture research and education," in *Proceedings of the 2004 Workshop on Computer Architecture Education: Held in Conjunction with the 31st International Symposium on Computer Architecture*, 2004.

[80] "Amd. codexl 3.1 edition @ONLINE." `https://gpuopen.com/compute-product/codexl/`, year = 2017.

[81] J. Cong, Z. Fang, M. Lo, H. Wang, J. Xu, and S. Zhang, "Understanding performance differences of fpgas and gpus: (abtract only)," in *Proceedings of the 2018 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, FPGA '18, (New York, NY, USA), pp. 288–288, ACM, 2018.

[82] J. Cong, P. Wei, C. H. Yu, and P. Zhou, "Bandwidth optimization through on-chip memory restructuring for hls," in *Proceedings of the 54th Annual Design Automation Conference 2017*, DAC '17, (New York, NY, USA), pp. 43:1–43:6, ACM, 2017.

[83] "Intel best practices guide @ONLINE." `https://www.intel.com/content/dam/www/programmable/us/en/pdfs/literature/hb/opencl-sdk/aocl-best-practices-guide.pdf`, year = 2017.

[84] C. Lattner and V. Adve, "LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation," in *Proceedings of the 2004 International Symposium on Code Generation and Optimization (CGO'04)*, (Palo Alto, California), Mar 2004.

[85] A. Canis, J. Choi, M. Aldham, V. Zhang, A. Kammoona, T. Czajkowski, S. D. Brown, and J. H. Anderson, "Legup: An open-source high-level synthesis tool for fpga-based processor/accelerator systems," 2013.

[86] S. Rogers and H. Tabkhi, "Locality aware memory assignment and tiling," in *Proceedings of the 55th Annual Design Automation Conference*, DAC '18, (New York, NY, USA), pp. 130:1–130:6, ACM, 2018.

[87] T. Hoefler and R. Belli, "Scientific benchmarking of parallel computing systems: Twelve ways to tell the masses when reporting performance results," in *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, SC '15, (New York, NY, USA), pp. 73:1–73:12, ACM, 2015.

[88] A. A. Purkayastha, S. Rogers, S. A. Shiddibhavi, and H. Tabkhi, "Llvm-based automation of memory decoupling for opencl applications on fpgas," *Microprocessors and Microsystems*, vol. 72, p. 102909, 2020.

[89] J. Cong, P. Wei, C. H. Yu, and P. Zhang, "Automated accelerator generation and optimization with composable, parallel and pipeline architecture," in *Proceedings of the 55th Annual Design Automation Conference*, DAC '18, (New York, NY, USA), pp. 154:1–154:6, ACM, 2018.

[90] S. Rogers and H. Tabkhi, "Locality aware memory assignment and tiling," in *2018 55th ACM/ESDA/IEEE Design Automation Conference (DAC)*, pp. 1–6, June 2018.

[91] "Altera sdk for opencl." `http://www.altera.com/literature/lit-opencl-sdk.jsp`, 2015.

[92] "Sdaccel environment optimization guide @ONLINE." `https://www.xilinx.com/support/documentation/sw_manuals/xilinx2016_4/ug1207-sdaccel-optimization-guide.pdf`, year = 2017.

[93] "Sdaccel2017.4 @ONLINE." `https://www.xilinx.com/products/design-tools/software-zone/sdaccel.html`, year = 2017.

[94] "Xilinx sdaccel hls examples @ONLINE." `https://github.com/Xilinx/SDAccel_Examples`, year = 2017.

[95] A. A. Purkayastha, S. Raghavendran, J. Thiagarajan, and H. Tabkhi, "Exploring the efficiency of opencl pipe for hiding memory latency on cloud fpgas," in *2019 IEEE High Performance Extreme Computing Conference (HPEC)*, pp. 1–7, 2019.