EXPLORING THE SCALABILITY OF OPENCL FOR MASSIVELY PARALLEL APPLICATIONS ON XILINX CLOUD PLATFORM

by

Jhanani Thiagarajan

A thesis submitted to the faculty of
The University of North Carolina at Charlotte
in partial fulfillment of the requirements
for the degree of Master of Science in
Electrical Engineering

Charlotte

2020

Approved by:

_____
Dr. Hamed Tabkhi

_____
Dr. Fareena Saqib

_____
Dr. Ronald Sass

ABSTRACT

JHANANI THIAGARAJAN. Exploring the scalability of OpenCL for massively parallel applications on Xilinx cloud platform. (Under the direction of DR. HAMED TABKHI)

Field Programmable Gate Array (FPGA) is becoming a preferred platform for the high-performance computing community because of the flexibility to adapt to new computing challenges. FPGAs also provide a greater power-efficient alternative for GPUs with its customizable data path and deep pipelining capability. Using high-level synthesis and optimization tools, we can achieve better and comparable performance to that of a CPU and GPU. OpenCL is the standard programming language for general-purpose parallel programming of a heterogeneous system. The availability of OpenCL has empowered high-performance execution of the massively parallel application. OpenCL-HLS for FPGA enables programmers to explore various software optimization with enhanced hardware capability.

We introduce a novel approach to study the scalability of OpenCL coarse-grain parallelism, Compute Unit (CU) replication on cloud FPGAs. This work demonstrates that for every application there is an optimum number of CU to achieve the maximum performance benefits with higher memory bandwidth utilization and optimum FPGA resources. We also provide a generic source-code template and a front-end design exploration tool to explore and identify the optimum CU number for a given application.

We have used the Xilinx SDAccel 2017.4 synthesis toolchain, which is an integrated development environment for FPGA for evaluation purposes. On the hardware side, the AWS cloud based Xilinx VU9P FPGA was employed. This project was funded by the Xilinx University Program (XUP). Our experimental results on a mix of 15 applications taken from the Xilinx benchmark suite vs2017.4 and the Rodinia Benchmark Suite vs3.1 show an average speedup of 6.4× and average bandwidth utilization

improved by 3.4× over baseline. Further to this, a mere 8% average resource utilization and 1.33× power overhead was reported. Our tool results in a 31% improvement in the total design synthesis time for an illustrative Histogram application.

Xilinx SDAccel based 'DDR' and 'burst transfer' optimizations were also explored to improve bandwidth and performance. These optimization are helpful for data hungry applications which have bandwidth as the major bottleneck. Combining CU along with DDR, we could achieve a 7.5× speedup for the Largeloop OCL (from SDAccel benchmark suite) application. In addition to this, we address the 'memory wall' and hide the memory latency problem by using OpenCL pipes. This approach involves splitting an application into 'read', 'compute', and 'write back' sub kernels which work concurrently. Results on seven massively parallel applications gave an average speedup of 5.2× with 2.2× bandwidth improvement on cloud FPGAs.

DEDICATION

I would like to dedicate this to my father Dr. Thiagarajan Venugopal for his encouragement and support. I would also like to thank my mother Mrs.U.S. Ramani Lakshmi and my brother Karthik Raja Kumar Thiagarajan for their love, care, and support.

ACKNOWLEDGEMENTS

I would like to thank Dr. Hamed Tabkhi for his supervision and for many great brainstorming sessions, and Mr. Arnab Purkayastha for patiently guiding me throughout my thesis. We would also like to thank Xilinx for allowing me to work on AWS cloud FPGA and provided me technical support for my work. Also, I would like to thank my family and friends for their support and love.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

CHAPTER 1: INTRODUCTION

The internet boom from the mid-90s which is also called the dot-com bubble has made a huge impact on us[1]. It drastically changed how we live by making a major impact on business, education, medicine and also helps us to make a better lifestyle choice. The wider use of the internet has led to an abundant data creation that can be processed and used for predicting the next trend. The processing of such huge data will not be an easy task. The architecture of the processing devices needs to be faster and efficient in producing the result. Sometimes it needs a bigger processing server which can process billions of data in a fraction of second. To process such a huge amount of data we need a computer with enormous processing capability.

With the impending death of Moore's law and the huge need for high volume processing[2], the technology started moving towards the search for better architecture which could process data with better throughput. This escalated the growth of High-Performance Computers and made them widely used in all industries. Field Programming Gate Arrays and Graphics Processing Units (GPUs) have increased the breadth of research in all the scientific fields and are used in solving huge mathematical equations in a short period. The modern GPU architecture contains thousands of core which can process terabytes of data. This ability to handle multiple tasks at the same time makes it highly efficient. For Instance, searching for a word in the document is a GPU friendly task since it can process multiple sections of the document in parallel whereas calculating the sum of a Fibonacci sequence is not so GPU friendly because of it's serial nature. GPUs have their builtin scheduler to manage concurrent thread execution at a massive scale through which it can hide the memory stalls. GPUs as the name says its primarily used for graphics rendering but its architecture

can be made use of in data-intensive industries which demand a powerful tool to evaluate massive machine learning problems. However, programming should be done very carefully to tap the full potential of GPU. FPGA also provides a huge scope be-

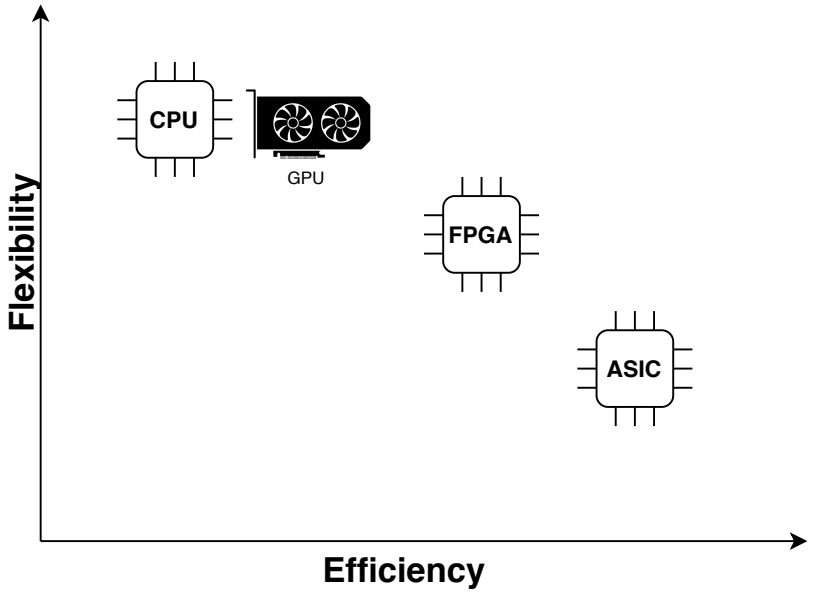


Figure 1.1: Flexibility vs Efficiencies of different platforms

cause of its programmability and its better energy efficiency as compared to GPUs as shown in Figure 1.1. FPGA consists of three main parts: Configurable Logic Blocks, Programmable Interconnects, Programmable I/O blocks. The logic block implements the logical functionality required by the design. These logical blocks are configurable i.e., its internal states can be controlled. This gives the FPGA it's cost-efficient parallel computing power which makes it suitable for rapid prototyping. It is also used for processing real-time Computer vision applications because of their structure to use both spatial and temporal parallelism. FPGA has a custom data path if there are any stalls in the thread it has to wait for the data. This affects the bandwidth of the device which leads to poor performance on FPGA.

Open Compute Language (OpenCL) in Figure 1.2 is a programming framework that can execute across heterogeneous platforms consisting of central processing units, graphics processing units, digital signal processors, field programming gate arrays and

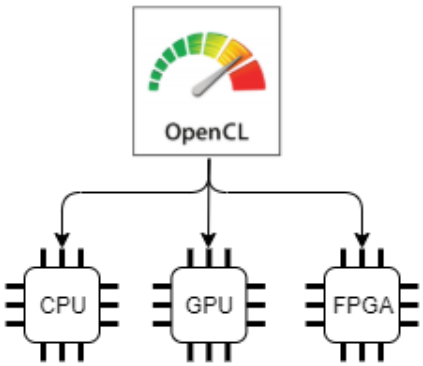


Figure 1.2: OpenCL platforms

other hardware accelerators. OpenCL has it's design challenges despite its potential. The challenges are mainly because of the significant difference in the architecture of CPU, GPU, and FPGA[3]. CPUs and GPUs are based on ISA whereas FPGA is mainly based on re-programmability. FPGA converts the algorithm into a customized data path and operational level parallelism. Also, it can exploit deep pipelining and temporal parallelism along with the advantage of spatial parallelism. GPUs, on the other hand, exploits spatial parallelism with its large number of cores. So, the programmers should understand the heterogeneous architecture before writing in OpenCL to have a greater speed up.

The hardware architecture should be in pace with the advancement of technology. Even though there is a lot of research done on optimizing code on GPUs and CPUs the growth has become stagnant. So, the traditional CPU and GPUs may not be suitable for all kinds of workloads. FPGA has a wider scope for research in all the fields. It has gained an interest in both industry and academia. Many companies like Xilinx and Intel have released lots of documentation and software which could optimize for the FPGA architecture. Programmers can understand how OpenCL source code will be mapped to FPGA architecture with the help of High-Level Synthesis (HLS). So, the programmer can optimize at the software level to have an impact at the hardware level.

GPUs work efficiently with the data level parallelism because of multiple CUs in its

architecture. Compute Unit (henceforth referred to as CU) can also be replicated on FPGAs. So, CU replication is a promising approach to increase OpenCL coarse-grain parallelism on FPGAs.

1.1 Problem statement

Explore possibility to improve high performance computing using spatial parallelism. CU replication is a good approach for exploring the spatial parallelism. The programmer follows the iterative method for replicating CU to achieve maximum performance which could take more than half a day for synthesizing and testing. There is a vital need for a better method to find an optimum number of CU with lower time spent for design space exploration.

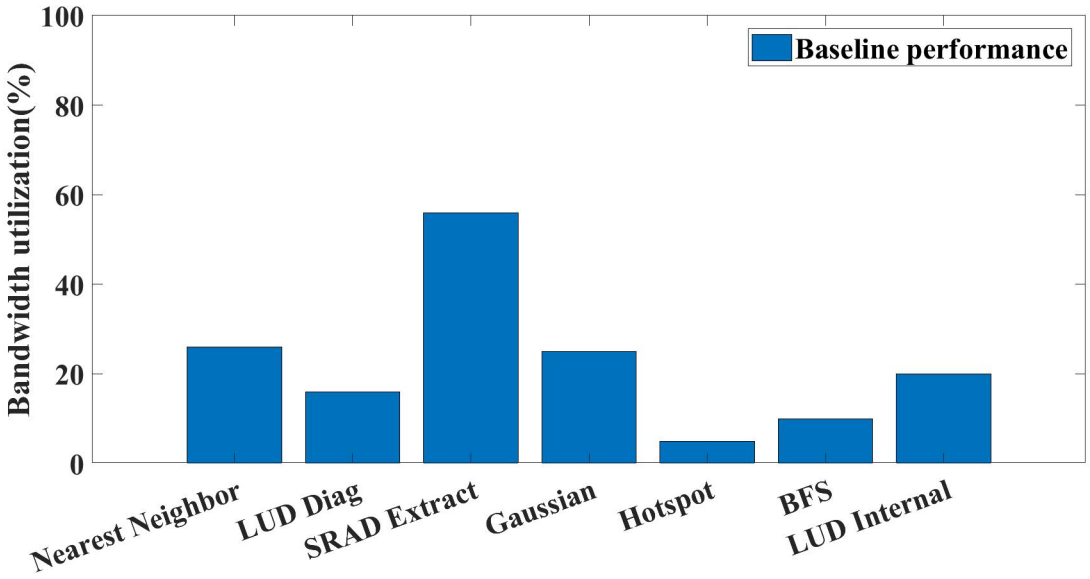


Figure 1.3: Baseline bandwidth utilization(%)

Figure 1.3, the bandwidth utilization is lower than 50% for most of the application. This allows us to do design space exploration. Often, the programmer should understand the architecture of FPGA to do design space exploration which makes it harder to use FPGA or OpenCL. There is a need for proper design space exploration which could cover the benefits and limitations of having CU replication. Another bottleneck for bandwidth is memory latency. Accessing the global memory posses

additional delay in the processing of data and significantly affect the performance.

1.2 Contributions

Our contribution is to propose a generic template to replicate CUs combined with a front-end tool to automatically identify the optimum number of CUs most suitable for a given application and synthesize that design.

This enables the programmer to remain oblivious of the design space and use our tool as a push-button solution. To improve the bandwidth further, we worked on optimizing the code by adding additional DDR and included Burst transfer. OpenCL pipe-based split-kernel temporal parallelism approach to hide the memory latency of massively parallel applications running on FPGA devices.

In summary, the contributions made are:-

1. Contribution 1: Automatic CU replication on cloud FPGA

Our tool automatically selects the sweet spot for extracting maximum spatial parallelism potential out of the FPGA

2. Contribution 2: Exploring DDR optimizations and burst transfer on cloud FPGA

Increase in DDR makes FPGA faster for power hungry applications. Optimization like DDR and burst transfer can be applied to increase the throughput

3. Contribution 3: Exploring OpenCL pipes for cloud FPGA

To hide the memory latency of massively parallel applications running on FPGA devices, OpenCL pipe-based split-kernel temporal parallelism approach can be used

1.3 Thesis Outline

The outline of this thesis is as follows. Chapter 2 will give you the background of the technology used. It reviews the basic OpenCL model on Xilinx FPGA using SDAccel 2017.4. Chapter 3, briefly overviews the related works in the field of OpenCL for FPGA devices. Chapter 4 proposes an open source tool to explore the scalability

of OpenCL coarse-grained parallelism on cloud FPGA. In chapter 5, optimizations for bandwidth and DDR have been explored. In chapter 6 , we present our first contribution to exploring the efficiency of OpenCL pipes for hiding Memory latency on cloud FPGA. Finally, Chapter 7 concludes the thesis and suggestions for future work.

CHAPTER 2: BACKGROUND

In this chapter we see the over all background of using FPGA and the impact of OpenCL on FPGAs. Then OpenCL sematics and the execution model is briefly explained. Additionally, execution steps of FPGA on AWS is explained in the this chapter.

2.1 OpenCL for FPGA

OpenCL is a parallel processing framework developed by Khronos group[4] to address the challenges of programming in multicore and heterogeneous platforms. OpenCL has made major changes in the computational capability for various types of applications. It also has the advantage of running on a variety of platforms by porting the algorithm to that architecture(CPUs, GPUs, and FPGA). Since OpenCL single programming model and a set of system abstraction that is supported by all heterogeneous hardware platform confirming to standard. And also has the major advantage of vendor portability with a single programming model. OpenCL kernel follows the programming method of C or C++. It provides lower-level hardware abstraction that allows OpenCL to expose to the underlying platform, memory and executions models of the device[5]. Multiple OpenCL kernels can be handled by a single FPGA. With such features, FPGA could gain significant performance gain compared to GPUs for certain applications. One of the limitations of using OpenCL is memory latency. Like GPUs, the data has to be moved from host memory to FPGA memory before any processing is done and back to host memory after the computation. OpenCL for FPGA needs significant improvement and fine-tuning to make the use of FPGA more powerful.

OpenCL platform model:

In High-Performance Computing, the knowledge about the hardware architecture is important to reap full performance benefit. An OpenCL platform model consists of one or more devices connected to the host. Each device can have one or more CUs. CUs are further divided into one or more processing elements as shown in Figure 2.1. Even within the single system, there could be many OpenCL platforms that could be targeted. The platform model's API gives the flexibility to choose and adapt to the desired platform and compute devices for executing its computation. This platform model is essential for application development as it enables the portability between OpenCL capable systems.

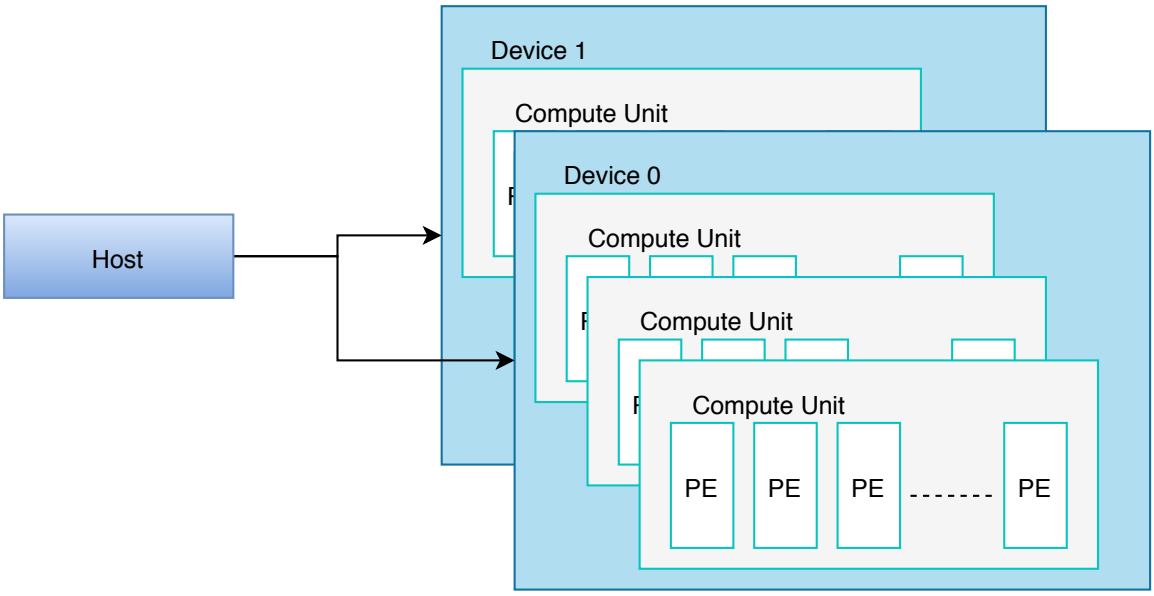


Figure 2.1: OpenCL Programming model

An OpenCL platform always starts with the host processor to communicate with the device using PCIe. The host processor has the following responsibility:

1. Manage the Operating system and enables the driver for all devices.

2. It sets up all global memory buffers and handles data transfer between the host and the device.

3. Execute the application's host program and also monitors the status of all CUs

in the system.

The platform model also presents an abstract device architecture that programmer's target when writing code. FPGA hardware vendors like Intel, Xilinx, and AMD will translate this abstract architecture to the physical hardware. All OpenCL platform executes a common set of OpenCL API. The implementation of OpenCL API functions is provided to each hardware vendor and have their own runtime libraries. The OpenCL runtime library is responsible for translating user commands as described by OpenCL API into the hardware-specific command for the given device.

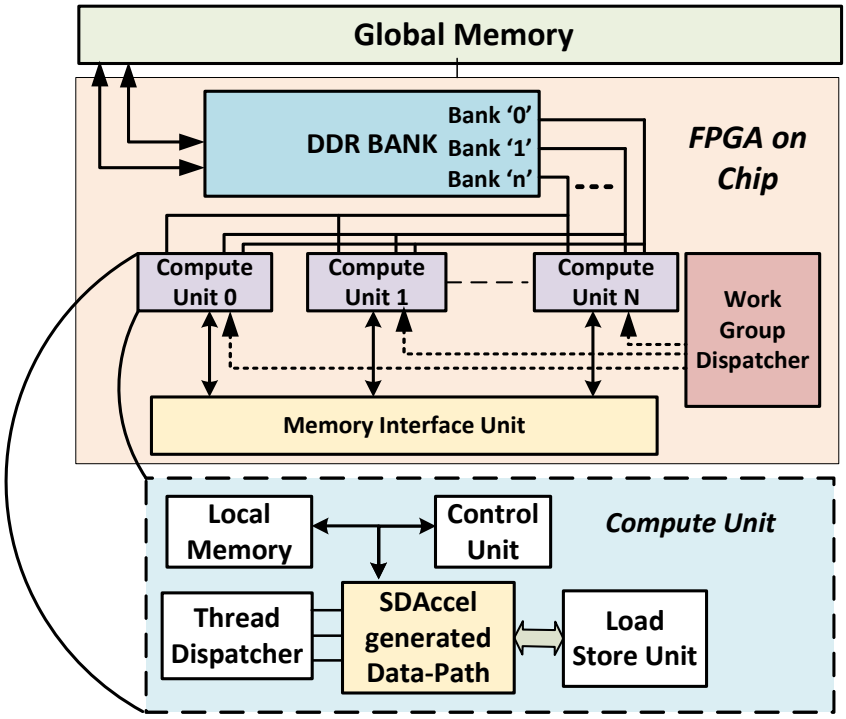


Figure 2.2: OpenCL Memory model

OpenCL context refers to the physical collection of hardware resources on which the application kernels are executed. The platform should at least have one heterogeneous device available for the execution of the kernels. The architecture of CPUs and GPUs cannot be changed and also has a fixed data path, memory system, and I/O architecture as shown in Figure 2.2. Also, it is not possible to attach high speed I/O to the compute kernel. This makes FPGA more useful because of the blank

computational canvas. The user can determine the level of customization that is needed to support the application kernel. In deciding the level of customization in a device, the programmer can take advantage of the fact the kernel CUs are not placed in isolation with the FPGA fabric. The hierarchical representation of the memory OS common across all OpenCL implementation but it is up to the individual vendor to define how the OpenCL memory model is mapped to specific hardware.

The OpenCL specification 3 major memory layers:

1. Host memory: The representation of system memory which is directly accessible from the host processor.

2. Global memory: This region of memory is accessible to both host and device.

3. Local memory: Local memory is local to a single CU. The host memory has no control over the Local memory.

4. Private Memory: It is the region of memory that is private to the individual work item that is executed by the processing element.

2.2 OpenCL execution Model

The openCL execution model defines how kernels execute. The most important concept is to understand NDRange execution on the OpenCL device. OpenCL kernel functions are executed one time for each point in the NDRange index space. This unit of work for each point in the NDRange is called work item. Each unit of work in for loop is allowed to execute in parallel and any order. It is this characteristic of OpenCL that allows the programmer to take advantage of parallel computing resources. Work items are organized into workgroups which are the unit of work scheduled onto the CU. Because of this workgroup also define the set of work items that may share the data using local memory. When the user submits the kernel for execution, they also provide the NDRange. This is called the Global size in OpenCL. The workgroup can also be set at runtime. This is called the local size in the OpenCL API. The user can also let the local workgroup size selected at the run time. Once the workgroup size

has been determined the NDRange has divided automatically into a workgroup and it is scheduled for execution on the device as shown in Figure 2.3.

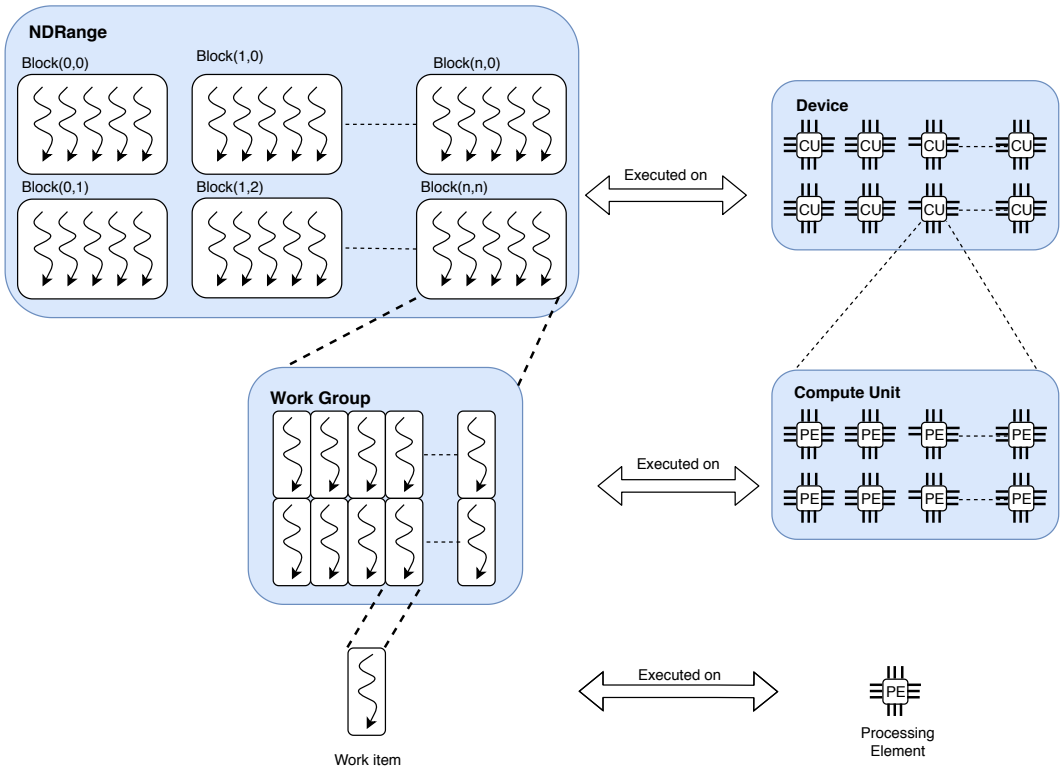


Figure 2.3: OpenCL Execution model

2.3 AWS Cloud FPGA

Amazon EC2 F1 instances use FPGA the deliver custom hardware acceleration. F1 instance offers high-performance computing with virtually unlimited capacity to scale out the infrastructure and flexibility to change resources easily as often as your workload demands. It also provides various development environments from low-level hardware developers to software developers with support for C/C++ and OpenCL and provides everything you need to develop, simulate, debug and compile and run hardware acceleration in EC2. The AWS market place includes multiple versions of the FPGA AMI with support SDAccel 2017.4 and other toolchain versions.

2.4 Xilinx SDAccel 2017.4 Tool chain

The SDAccel environment is a heterogeneous system architecture platform to accelerate compute-intensive tasks using Xilinx FPGA devices. It contains a host x86 machine that is connected to one or more Xilinx FPGA devices through a PCIe Bus. Also, offers all of the features of standard software development environments like optimized compiler for host applications, cross compilers for FPGA with a strong debugging environment to identify and resolve issues in the code as shown in figure 2.4. Within this environment, the SDAccel build process uses a standard compilation and linking process for software and hardware element. The host process is built through GCC and the FPGA is built through a separate process using the Xilinx XOCC compiler.

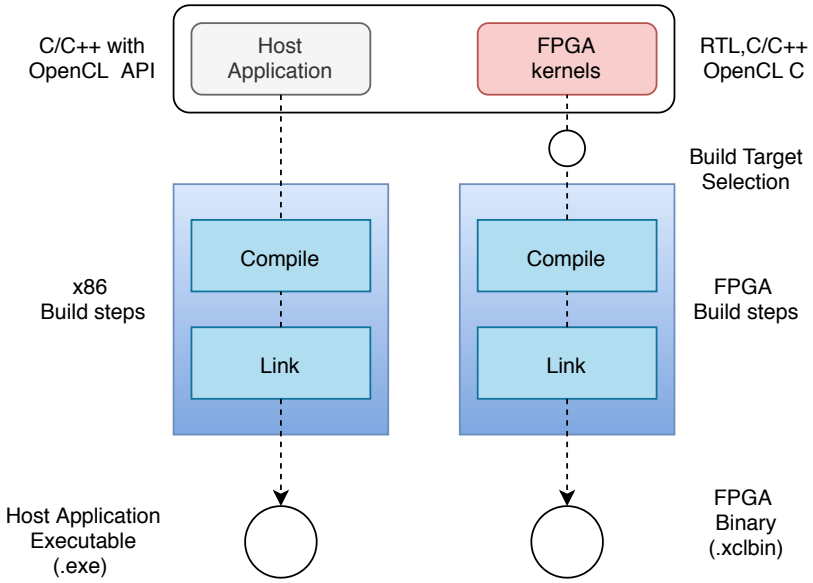


Figure 2.4: SDAccel Execution model

1. Host application source file is compiled to object file (.o) The object files are now linked with Xilinx SDAccel runtime shared library to create the executable (.exe)

2. FPGA build process using XOCC. Each kernel is independently compiled to Xilinx Object (.xo) file i.e, C/C++ and OpenCL C kernels are compiled for imple-

mentation in an FPGA using the XOCC compiler. This step leverages the Vivado HLS compiler. RTL kernels are compiled using package_xo utility. The RTL kernel wizard in the SDAccel environment can be used to simplify this process.

3. The kernel .xo files are linked with the hardware platform (.dsa) to create the FPGA binary (.xclbin)

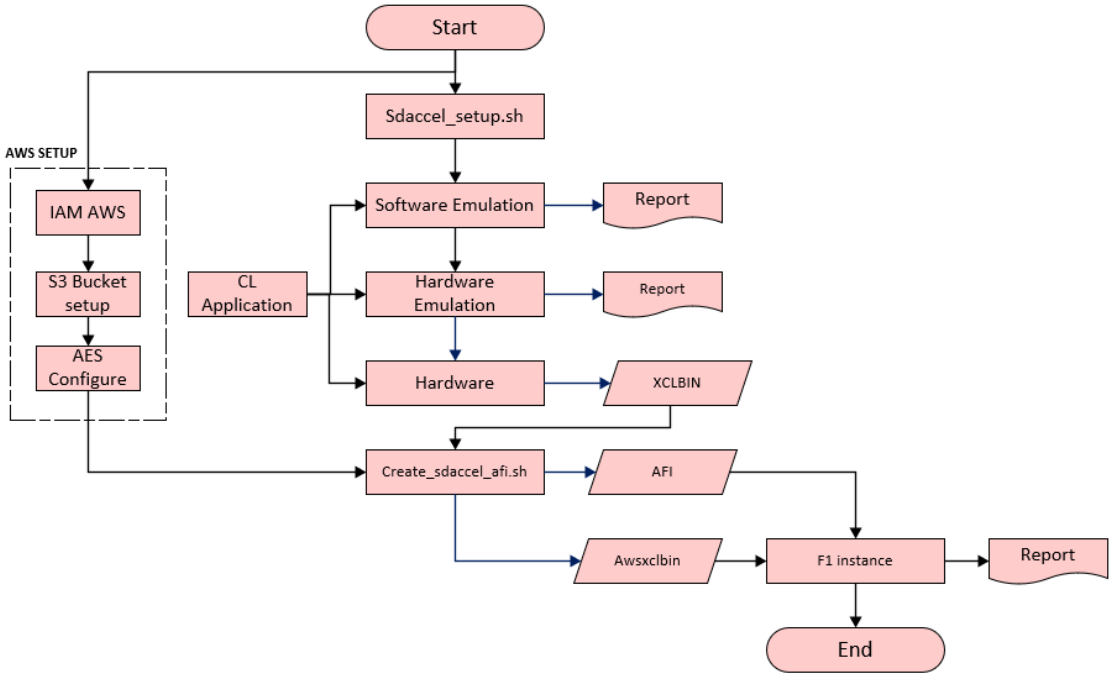


Figure 2.5: SDAccel Execution and setup flowchart

2.4.1 Execution Model

SDAccel 3 different built targets. in which 2 of them are used to debug the kernel code and to validate the purpose and default hardware target used to generate the actual FPGA binary as shown in Figure 2.5.

Software Emulation (sw_emu): Host and kernel are compiled to run on the x86 processor. This target is useful to identify syntax issues, perform source-level debugging and to confirm the functional correctness of the system of the kernel code running together with the application and verify the behavior of the system. It is

faster than the hardware emulation since it runs on x86.

Hardware Emulation (hw_emu): The kernel code is compiled into a hardware model(RTL) which is run in a dedicated simulator. It takes a longer time to build for the hardware emulation. This target is useful for testing the functionality of the logic that will go into FPGA and to get a performance estimate with the best-debugging capability and moderate compilation time.

System (hw): Kernel code is compiled into a hardware model(RTL) and this is implemented on the FPGA device. The final FPGA run provides accurate performance results with long build time.

Table 2.1: System characteristics used for study.

Host system	Intel Xeon Platinum 8000
Host specs	16-core, 32GB DDR4 Memory
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 CUs	28

2.5 Experimental setup

We have taken applications from SDAccel 2017.4 directory and Rodinia benchmark. SDAccel directory contains a diverse range of complex applications that allows the users to directly work on them without larger modifications. Rodinia [6] is an open source benchmark suite that allows the researcher to study architecture such as FPGA, GPU, and CPU. It has a diverse set of applications with various computation patterns with state of the art algorithm and also, provides input sets for testing different situations. We used Xilinx SDAccel 2017.4 to compile and synthesize OpenCL codes. SDAccel profiler collects kernel performance data such as execution time took by one CU, bandwidth efficiency of the global memory, resource

utilization, and power. The applications binaries are built for Xilinx Virtex VU9P FPGA using SDAccel 2017.4. Table 2.1 list the parameters of our FPGA platform. We also used AMD FirePro W7100 GPU to report our GPU kernel performance and AMD CodeXL power profiler for power information.

CHAPTER 3: RELATED WORKS

The advent of OpenCL for FPGAs combined with capabilities of HLS tools has generated tremendous amounts of interests and impacted a lot of research in this area. OpenCL execution efficiency on FPGAs devices has been explored in many works [7, 8, 9]. Graphic processing unit is relatively older than FPGA. GPU programming is relatively easier than FPGA [3]. To harness the full potential of FPGA, the programmers should understand the architecture of an FPGA. There is a lot of research in various different field exploring the possibilities of using FPGA and OpenCL in different domain such as image processing, deep learning and neural network[10, 11]. Also, many researches focused on the comparison of the efficiency of GPU and FPGA[12, 13, 14]. For some applications, FPGA's performance numbers are better than GPU. This is mainly due to availability of software optimization on FPGA[15]. Intel[16] and Xilinx[17] are providing various user friendly opportunity for the research community to explore FPGA. They also provide development environment with various possibilities of software optimization[18, 19].

Most of the previous research work primarily focus on application-specific performance optimization techniques[20, 12]. The past work also proposes a framework that performs a metric-guided design space exploration while [21, 22] introduce analytical models for FPGA based architectures and explore OpenCL directives like loop unrolling, pipelining, array partitioning, PE duplication(Work-Item replication) etc.

Efficient multi threading is done by constructing Sparse matrix vector using compiler and it can support irregular applications with dynamic workload. context switching is used to hide the memory latency[23]. Few Researches have also proposed memory optimization for convolution neural networks. Apart from that, to reduce the

data movement while maintaining the accuracy favorable architecture which is hardware friendly has been used[24].Algorithm which posses irregular memory access also affects performance on multi core architecture[25, 26].

The researches in the feld of OpenCL HLS has benefited many fields of high performance computing. Many recent articles have explored the possibilities of improving performance of applications on FPGA[8, 6, 7]. By using optimization methods such as a single pipeline model and by exploiting data parallelism by generating SIMD parallel pipes, up to 6 times speedup [15] can be achieved. Another optimization technique to extract the run time information to obtain the true dependencies between operations [27]. This assists the designer in evaluating different architectural options in the context of high-level synthesis and a better understanding of the performance impact of different accelerator design choices [28]. COMBA(Comprehensive model-based analysis) [29] framework analyzes the effect of functions, loops, and arrays in the design description which can be done by using pluggable analytical models.

There is also an interest in optimizing the interkernel communication using OpenCL pipe semantics. A mechanism in [30] can efficiently capture the behavior for 2 dimensional (2D) vision algorithms to benefit OpenCL pipe based execution.

Another example [31] is to use pipes for efficient DNA and RNA sequencing. A design space exploration is offered [21] by a parallel pipeline analytical model to capture the performance resource trade-off . All these ideas focus mainly on interkernel parallelism and data transfer across multiple kernels in producer-consumer fashion. Prefetching the data available before the time of execution [32, 33] is another approach that can hide memory stalls and lead to better speedup. There has been a few relevant works done for hardware prefetching [34], [35] software prefetching [36, 37] and merging them to achieve maximum performance. Moreover techniques like double buffering [38] and [39] have shown performance improvements by overlapping memory accesses with computation. Double buffering involves decreasing the granularity of

data transfer to smaller tiles and loading into on-chip memories. Data is accessed directly from the main memory, which reduces the setup costs when the bulk transfer of data within tiles takes place. A major disadvantage of this method is that the memory must have a predictable streaming access pattern. Another disadvantage would be the need to understand the buffer allocation of buffer tile sizes to perform double buffering optimization.

Overall, OpenCL for FPGAs is still at its early stages. There is a lack of in-depth analysis and generalized solutions to enhance the OpenCL execution efficiency on FPGA devices. The major focus has been on creating an efficient application specific data-path rather than removing the bottleneck of memory latency. In this paper, we propose a generalized design solution (borrowed from throughput-oriented design principles) to overlap memory access and computation at a massive scale to hide memory latency and avoid memory stalls.

Even though current accelerators show better performance over the generic processor, the design space of FPGA is still not well explored and systematized. This is especially true for Compute Unit replication which has enormous performance potential. It is therefore important to not only design scalable accelerators but also design tools that can significantly reduce the synthesis design-time that is economically viable across all infrastructure. In this work, we formalize these ideas to propose a generic template and a novel tool that allows faster FPGA synthesis with maximum parallelism potential and least resource and power overhead. To the best of our knowledge this is the very first work that attempts to do so for Xilinx FPGA platforms.

CHAPTER 4: EXPLORING THE SCALABILITY OF OPENCL COURSE GRAINED PARALLELISM ON CLOUD FPGA

This research provides an exhaustive study on the scalability of OpenCL coarse-grain parallelism, CU (referred to as CU) replication on cloud FPGAs. This work demonstrates that for many applications there is an optimum number of CUs to achieve the maximum performance benefits for memory bandwidth, memory conflicts introduced by CU replication, and available FPGA resources.

OpenCL primarily allows two types of parallelism- data level and task level. CU replication a promising approach to increase OpenCL coarse-grain parallelism on FPGAs. The single kernel CU is split into multiple kernels so that all the data can be processed in parallel. Not all applications will work perfectly with the splitting of kernels because of the bandwidth and resource restriction. Our work shows a detailed study on all range of applications and with the maximum CU achieved per each application. Our experimental results of 15 applications are taken from Xilinx SDAccel 2017.4 suite and Rodinia benchmark suite v3.1 and achieved a maximum speedup of 6.4 times with 3.4 times bandwidth improvement over the baseline with mere 8% of average resource utilization.

Figures 4.1 and 4.2 plot the baseline profiling information of 15 massively parallel applications on Xilinx cloud FPGAs. The figures show the average resource utilization of less than 5% and a bandwidth of less than 50%. This opens up opportunities to explore CU replication. However, implementing CU replication requires a detailed architecture knowledge of the device and programmers often opt for the 'trial and error' approach which is highly inefficient and cannot achieve full parallelism benefits. This is partly because the extensive design space exploration consumes a lot of time and partly due to the unavailability of a generic programming approach to carry out

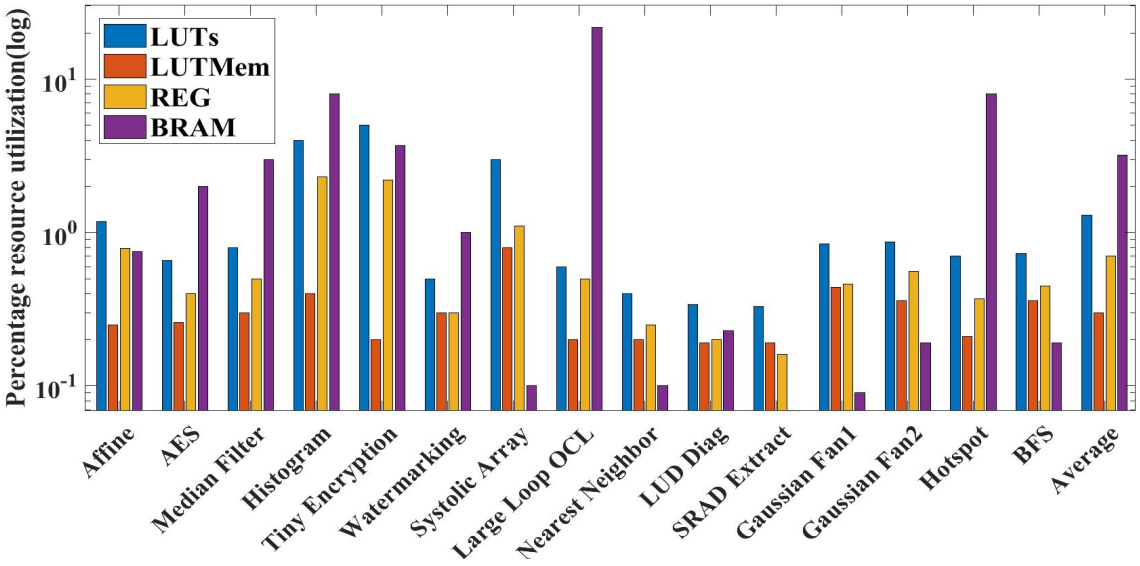


Figure 4.1: Baseline resource utilization

the coarse-grain CU parallelism.

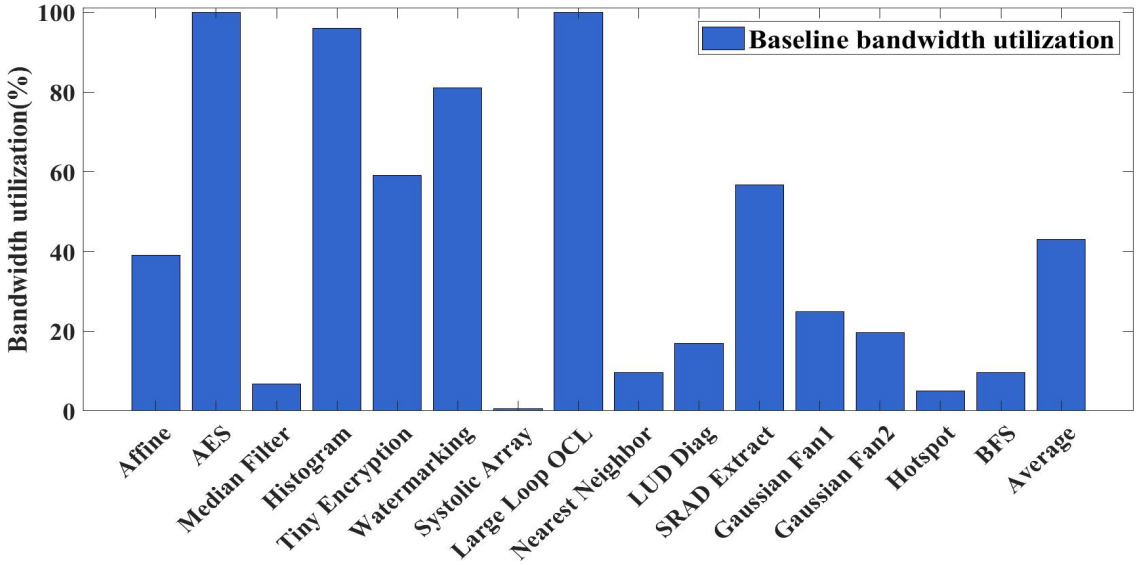


Figure 4.2: Baseline bandwidth utilization(%)

In summary, the contributions made are:-

1. Proposed a generic template to replicate CUs so that the user can modify the application especially suited for Xilinx platforms.

2. Present a novel tool to automatically identify an optimum number of CUs most

suitable for a given application hiding the hassle of manual design space exploration.

3. Port all the 15 OpenCL applications to run on GPUs and compare the performance.

This work explores the scalability of coarse-grain parallelism on Xilinx cloud FPGAs. It provides an in-depth study on limitation and benefits of CU (CU) replications across many FPGA devices. The work demonstrates that there is a unique optimum number of CUs per each given application for available memory bandwidth, memory access conflicts across CUs, and available FPGA resources. At the same time, the work proposes a generic template to replicate CUs combined with a front-end tool to automatically identify the optimum number of CUs most suitable for a given application and synthesize that design. This enables the programmer to remain oblivious of the design space and use our tool as a push-button solution.

4.1 CU Replication

CU replication is a task-based parallelism approach implemented on top of the existing pipeline or temporal parallelism. This offers courser level granularity for spatial thread along with the temporal thread-level granularity which can give maximum speed up.

CU replication is done for the single kernel program. Multiple copies of the single program as shown in Figure 4.3 are replicated along with datapath, thread dispatcher, control unit and, load/store. The workgroup dispatcher splits the work among multiple CUs and, all CUs run concurrently utilizing the full potential of FPGA while maintaining the synchronization between all the threads.

CU replications comes with its own set of problems.

1. Replicating CU is an NP-hard problem since multiple parameters like FPGA resources, workgroup sizing, the additional cost associated with the CU setup is involved.

2. CU will not give any performance improvement since the over splitting of jobs

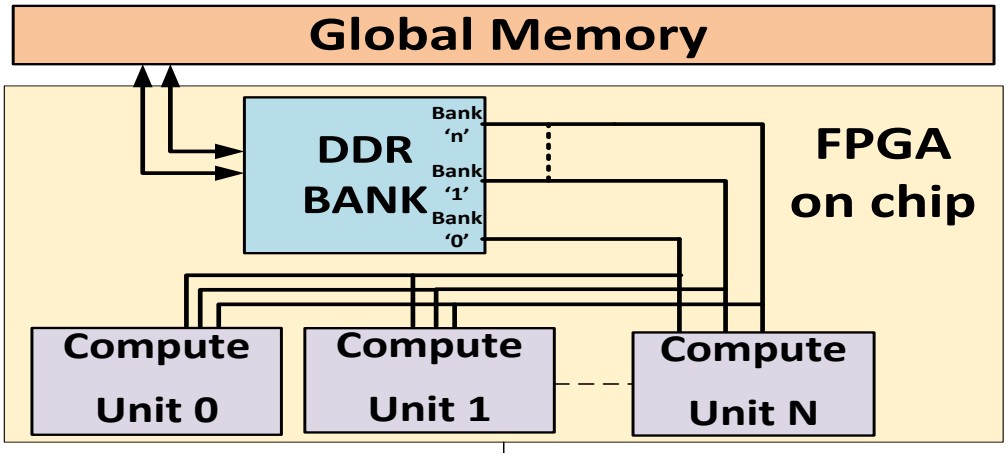


Figure 4.3: Multiple CU

leads to the extra time taken to access the global memory.

3. The bandwidth might reach its maximum potential. Xilinx used AXI interconnect and they limit to having 10 masters/slave interface the kernels and the interconnect connecting to the memory controller.

4. The OpenCL CU replication on Xilinx FPGAs imposes programming challenges to an average programmer.

4.2 Generic Tool for CU replication

This section introduces generic template for CU Replication.

Listing 4.1: 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++)
    {
        }
```

```
}
```

Listing 4.2: 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);
.................
}
```

For any given application mapped into Xilinx FPGAs, each workgroup is mapped to each CU [19]. We, therefore, divide the outer loop of the kernel viz. Y_SIZE by the total global work size which is equal to the number of CUs (Listing 4.1).

The local work size in the host code is kept as 1 as the work items are pipelined inside the kernel using xcl_pipelineloop pragma provided by Xilinx SDAccel optimization. In this way, throughput and performance increases. Both local and global work size is specified in NDRange OpenCL API call. Thus any OpenCL kernel(2D/3D) can be easily split up as per the template.

4.2.1 Algorithmic implementation

Algorithm 1 presents our proposed automation tool for CU replication. The modified OpenCL kernel code (based on the generic template) is the primary input to the tool that gets initialized through the command line by providing an argument for the number of CUs (Work Group)(line 2). Tool flow diagram in Figure 4.4 outlines the algorithm.

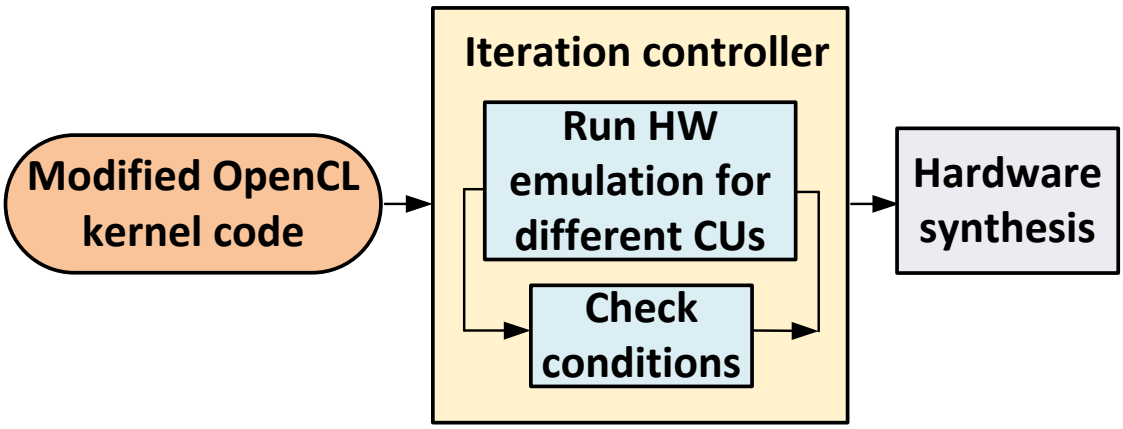


Figure 4.4: Tool flow

At the heart of our tool is the iteration controller(Figure 4.4). We choose Hardware (HW) emulation for performing different check conditions. We set a multiplication factor of *Delta* to extract the maximum parallelism without maxing out the resources. This number is purely a programmer's choice. Delta(Δ) allows for 'automatic jump through iterations' i.e, depending on present Execution time value iterations across number of CUs can be unevenly incremented. This helps in a dynamic, faster design exploration time resulting in very few iterations.

The iteration controller simultaneously calculates if x(t) $<(\Delta)$*x(t-1) (where x(t) is the current value of CU execution time and x(t-1) is the previous), continue iterating else stop.

The maximum number of CU per application will vary depending on the resource utilization and bandwidth. Having many CUs will result in a higher probability of maxing out resources and also increase the synthesis time. The TIMING (line 17-22) and RESOURCE (line 24-29) checks are concurrently made until the optimum CU number is generated.

4.2.2 Case Study

In this section, we first arbitrarily select the *histogram* application and run hardware (HW) + software (SW) emulation and an actual hardware run on the FPGA for

Algorithm 1 CU Replication Tool

```
function MAIN()
    Validate project directory path
    Initialize Execution Argument
    Set Delta(Δ) = Variable t>0
    Set CU ← 1
    while true do
        Set CU ← CU + 1
        Run Hardware Emulation with Execution Argument
        if TIMING or RESOURCES then
            Terminate loop
        end if
    end while
    Execute system synthesis
    Store results
end function

function TIMING()
    y = f(x) ← parse profile summary file
    if (y = f(x) = x(t) < Delta ∗ x(t − 1) then
        y = f(x) = x(t)
    end if
end function

function RESOURCES()
    y = f(x) ← parse profile summary file
    if (LUT or LUTMem or REG or BRAM or BW)<100 then
        y = f(x) = x(t)
    end if
end function
```

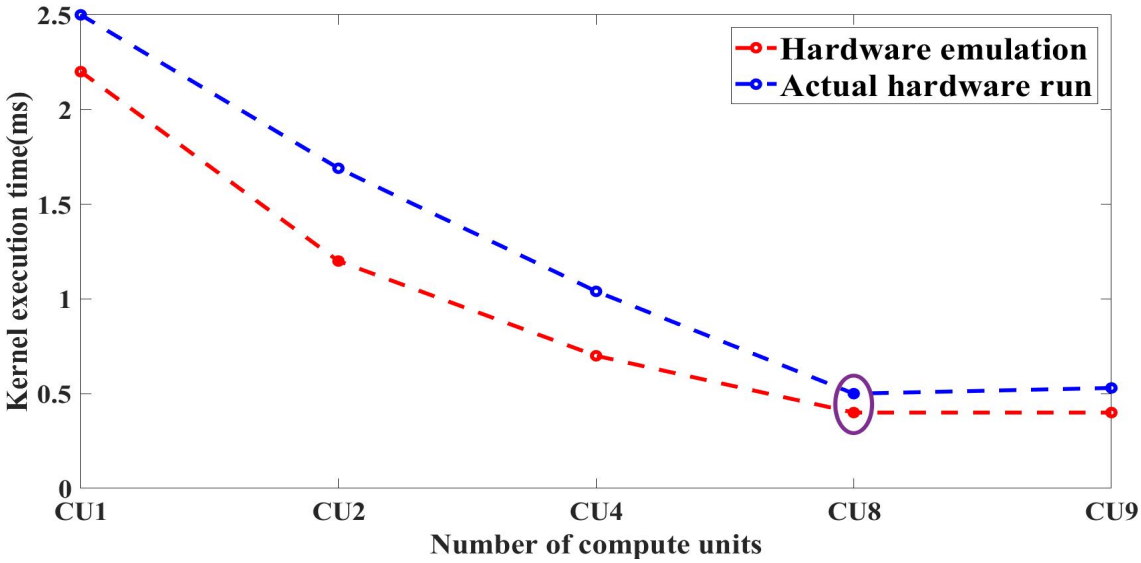


Figure 4.5: Histogram application for tool validation Hardware emulation

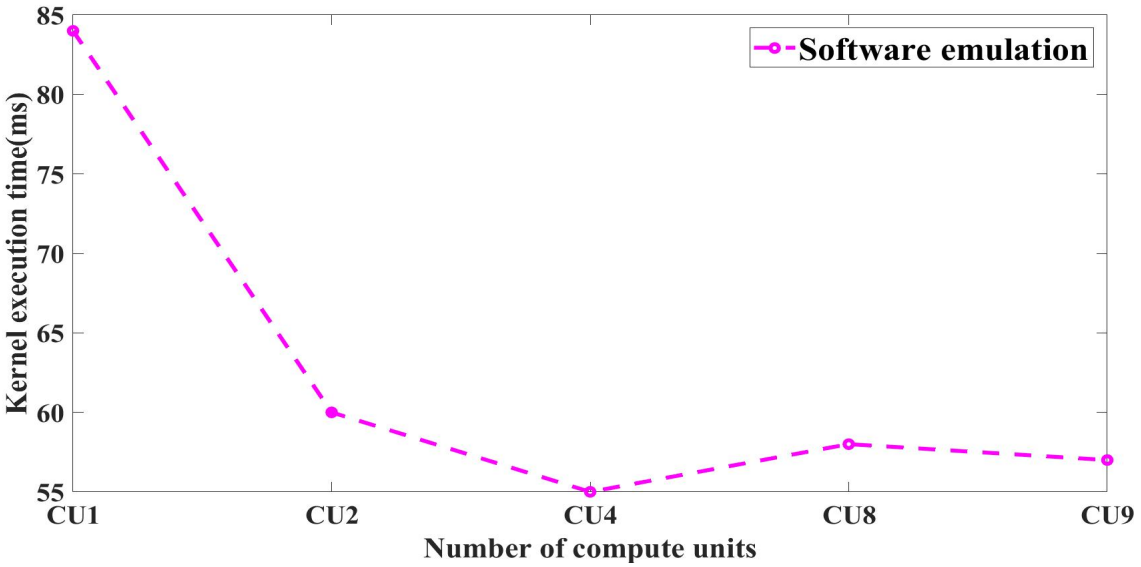


Figure 4.6: Histogram application results for tool validation Software emulation

analysis. We limit our set of experiments due to the time complexity of synthesis. Next, we discuss the following three aspects for our tool validation.

1. *Provide reasoning for choosing HW over SW emulation*- Figure 4.5 shows the HW emulation and system run's execution time and Figure 4.6 shows the SW emulation execution time. The disparity between timing information is evident from both the graphs. While the HW emulation resembles the actual hardware, software emulation time is quick but inconsistent. On the flip side, HW emulation suffers from comparatively longer compilation time for larger kernel and bigger data. However this is a trade-off that we take into account considering the accuracy requirements.

HW emulation suffers another drawback which arises since logic optimizations done by Synthesis and Place & Route can reduce required resources, meaning we could fit in more CUs than indicated by emulation. However, this is overshadowed by the fact that we almost never hit the resource limit, other factors like reaching max bandwidth is the prime performance bottleneck.

2. *Show one complete design space exploration to verify our tool results*- HW emulation (Fig 4.5) shows the optimal speed up with 8 CUs. We validate this by

running an entire design space from CU1 (baseline) through CU9 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. *Report timing*-Finally, we report the synthesis time for each iteration of CUs (Table 4.1) for the same application. We discuss the implications of these results in Sec 7.0.2 in detail.

Table 4.1: Design space exploration of Histogram application

Number of CUs(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

CHAPTER 5: EXPLORING DDR OPTIMIZATIONS AND BURST TRANSFER

Optimizations are necessary on the OpenCL program for efficient data movement, kernel development for the best performance on FPGA. There are numerous OpenCL commands which are used for 'read', 'write', and 'copy' buffers and images. The host code and the kernel code needs to be changed for a better data transfer. A template for changing the code is provided in this chapter.

5.1 Using DDR Transfer

DDR (Double Data Rate) is the advanced version of synchronous dynamic random access memory that waits for clock signals before responding to control inputs. DDR uses both the falling edge and rising edges of the clock signal. So, it can transfer twice per each clock cycle. DDR is used for the application which has a very high bandwidth to the global memory. The device with multiple DDR bank can be targeted so that kernels can access all available memory banks simultaneously as shown in Figure 5.1. SDAccel supports multiple DDR banks. To take advantage of multiple DDR banks, the user needs to assign CL memory buffer to different banks in the host code as well as configure the XCL binary file to match the bank.

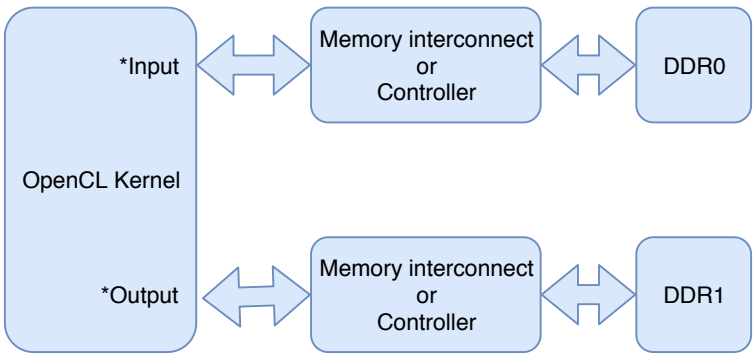


Figure 5.1: Global memory to DDR Banks

In the figure, we see that the OpenCL kernel's global memory is connected to two separate DDR and controlled by memory or interconnect controller. Without DDR banks both the global memory access the same DDR memory to transport the data to and from the global memory. SDAccel accelerator card support 1,2 or 4 DDR Banks and up to 80 GB/s raw DDR bandwidth. For kernels moving a large amount of data between the FPGA and DDR Xilinx recommends that you direct the SDAccel compiler and runtime library to use multiple DDR banks to move data effectively between the kernel and global memory.

Listing 5.1: Generic template of adding DDR Channel

```
#include <CL/cl_ext.h>
int main(int argc, char** argv)
{
 cl_mem_ext_ptr_t inExt, outExt;
 inExt.flags = XCL_MEM_DDR_BANK0;
 outExt.flags = XCL_MEM_DDR_BANK1;
 inExt.obj = input_data.data() ; outExt.obj = output_data.data();
 inExt.param = 0 ; outExt.param = 0;
 int err;
cl_mem buffer_inImage = clCreateBuffer(world.context,
CL_MEM_READ_ONLY| CL_MEM_EXT_PTR_XILINX,image_size_bytes,
    &inExt, &err);

 cl_mem buffer_outImage = clCreateBuffer(world.context,
CL_MEM_WRITE_ONLY | CL_MEM_EXT_PTR_XILINX,image_size_bytes,
    &outExt, NULL);
}
```

In listing 5.1, we create an OpenCL memory pointer and initialize it to DDR Bank 0 and 1. Then we connect the inputs and the output to each DDR bank. The OpenCL

buffers that is "buffer_inImage" and "buffer_outImage" are created using "cl_mem". While creating OpenCL buffer it is important to pass CL_MEM_EXT_PTR_XILINX, the pass is the Xilinx extension for using DDR bank.

5.2 Using Burst transfer

Burst transfer helps in hiding memory latency as well as improve the bandwidth utilization and efficiency of the memory controller. It is recommended to infer burst transfers from successive requests of data from consecutive address locations. This achieves the best efficiency of the memory controller and keeps the CU inside the FPGA device busy all the time.

The memory layout of the data object is a key factor to consider for improving the data transfer efficiency. Considering a 4x4 matrix, an example conceptually it is a two-dimensional array as shown in the matrix logical layout. In C/C++ programming, arrays are physically stored in row-major order that all the data in the row is stored in consecutive order. If the program reads column-wise the burst transfer cannot be done.

Listing 5.2: Generic template of adding Burst transfer

```
kernel __attribute__ ((reqd_work_group_size(1, 1, 1)))
void vadd(
 const __global uint16 *in1, // Read-Only Vector 1
 const __global uint16 *in2, // Read-Only Vector 2
 __global uint16 *out, // Output Result
 int size // Size in integer
 )
{
 local uint16 v1_local[LOCAL_MEM_SIZE]; // Local memory to store
vector1
 int size_in16 = (size-1) / VECTOR_SIZE + 1;
 ...
 int chunk_size = LOCAL_MEM_SIZE;
```

```
 //boundary checks
 if ((i + LOCAL_MEM_SIZE) > size_in16)
 chunk_size = size_in16 - i;
 //Each chunk of data is loaded into the global memory
 v1_rd: __attribute__((xcl_pipeline_loop))
 for (int j = 0 ; j < chunk_size; j++){
 v1_local[j] = in1 [i + j];
 }
...
}
```

In listing 5.2, We calculate the boundary for each chunk of data that needs to be transferred to the local memory. Then loops are used to access each data from the global memory and transfer it to local memory. The latency for accessing the data from the local memory is slower than accessing it from global memory. Each CU has its own local memory. So, each CU should transfer the data that it needs to process on to its local memory. It is important to know that the size of local memory is limited.

CHAPTER 6: EXPLORING THE EFFICIENCY OF OPENCL PIPES FOR HIDING MEMORY LATENCY

OpenCL-HLS enables parallel programmers to develop a customized data path that best fits the application. The performance benefits of FPGA not only comes from the inherent pipeline architecture that allows temporal parallelism but also comes through the advantage of spatial parallelism. OpenCL is primarily based on GPU architecture that benefits from Single Instruction Multiple Thread (SIMT). FPGAs architecture does not have a multi-level cache hierarchy or run-time scheduler to make use of SIMT. This means that there is a lack of data that arise from a single thread waiting for the memory in the pipeline. The execution path of the pipelined thread has to wait for the previous thread to receives the data. This memory stall leads to the under-utilization of memory and bandwidth.

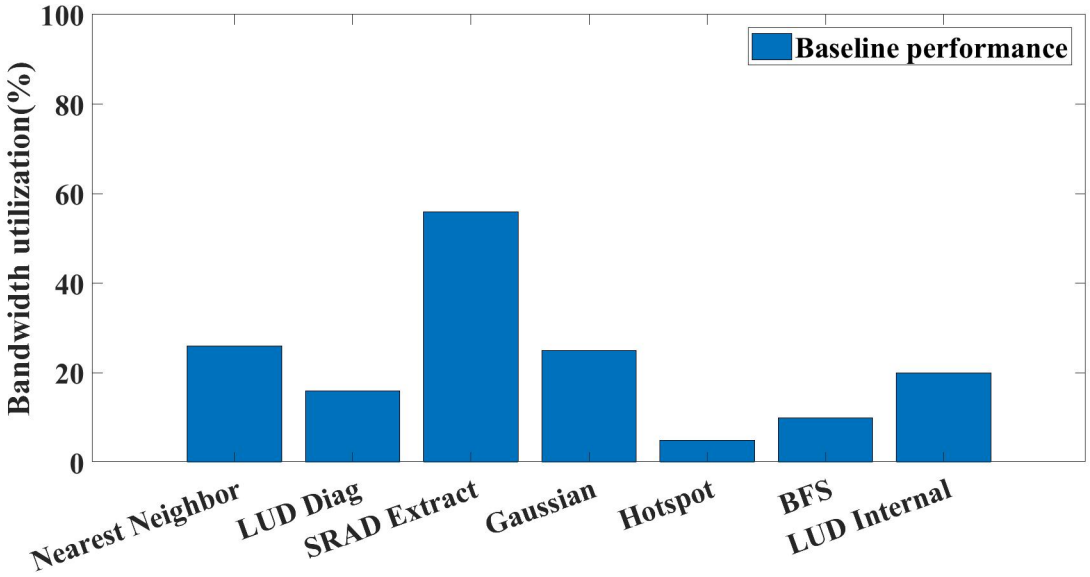


Figure 6.1: Baseline bandwidth utilization(%)

Figure 6.1 shows the overall bandwidth utilization of all the applications from Rodinia benchmark on Xilinx VU9P FPGA. The average bandwidth utilization of all

the seven application has less than 25% bandwidth utilization. This motivates us to identify a solution that mitigates the memory wall problem.

6.1 OpenCL pipes

OpenCL pipe standard was introduced in OpenCL 2.0 and it is incorporated into the Xilinx SDAccel environment. A pipe is a memory buffer that can be used to communicate from one kernel to another so that kernels can avoid accessing the external memory which leads to external memory access latency. A pipe is a memory object that has an associated data buffer alongside it. The pipe memory is used to store the inter kernel communication data while data buffer is used to synchronize data communication to the thread id. Pipe objects are accessed by built-in pragmas that are vendor-specific and cannot be accessed by the host.

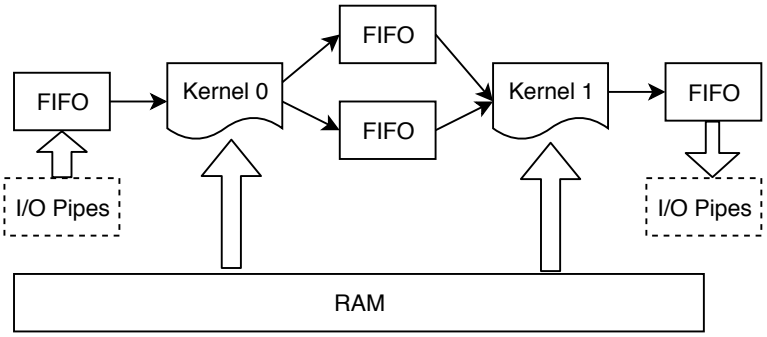


Figure 6.2: OpenCL pipes

Pipes communicate in the first in first out fashion i.e, the producer will fill the pipe with data that is to be used by the consumer. There is a constant synchronization between the producer and consumer. The depth of the FIFO pipe is decided by the programmer. Figure 6.2 shows kernel communication using FIFO pipes. We notice that kernel 0 uses 2 FIFO pipes. The number of pipes initialized is based on the number of global variables that need to be sent to the other kernel.

6.2 Methodology

The first step is to identify the memory access pattern. This can be done by using LLVM based automated tool which could do static analyses on the complex access behavior and data dependencies and the next step is to identify the global variables that can be decoupled from computation[40]. Then we use this information to split-up into read/write and compute sub kernels. Then we run each of these kernels concurrently. The figure 6.3 shows a conceptual architecture of the memory

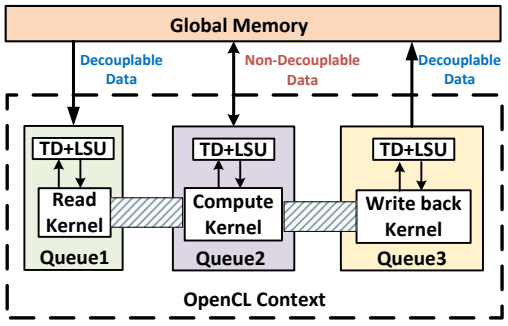


Figure 6.3: Kernel communication through OpenCL pipes

decoupling approach in which the kernel runs concurrently overlapping memory access and thread execution at the runtime. The single kernel is split into sub kernels: (1) Read kernel, (2) Compute kernel, (3) Write back kernel and each sub kernel is executed in separate CU. Since all three sub kernels run concurrently, they must have synchronization between all CUs.

We also introduce a generic template for decoupling the memory access and computation. To decouple the memory access first we must identify if the application kernel can be decoupled i.e., the global variable whose access pattern is statically identifiable is categorized as decouplable (Example - streaming pixel, fixed stride access) while non-decouplable variables are runtime dependent access (Example - next data index, quick sort algorithm) [41]. After classifying the data as static predictable, we propose a template with which the programmers can split the kernel baseline.

In Figure 6.4, the OpenCL code is converted to its intermediate representation and

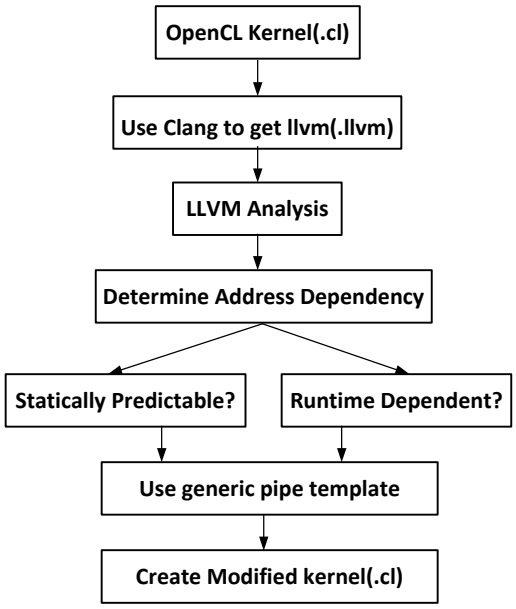


Figure 6.4: Flow Chart

sent to the LLVM analyzer. The LLVM analyzer determines the address dependency and categorizes in to statically predictable and Runtime dependent. if it is statically predictable use a generic pipe template to modify the code.

Table 6.1 shows the application we took for this paper. Most of the mathematical and image processing applications are memory decouplable.

Table 6.1: Application list

Application	**Class of application**
NN	Data Mining
SRAD_Extract	Image processing
Gaussian	Linear Algebra
Hotspot	Physics Simulation
BFS	Graph Algorithms
LUD_Diag	Linear Algebra
LUD_Internal	Linear Algebra

6.2.1 Generic Template

Listing 6.1 shows the baseline kernel of the Nearest Neighbour application. The LLVM static analysis tool sees if all the global variables are decouplable. Then we remove all local variables from the port list and its relevant computation.

Listing 6.1: 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));

...}
```

Table 6.2 shows the baseline profiling information for all applications. It lists the resource utilization, execution time and average bandwidth of each OpenCL kernel. As shown in table 6.2 the resource utilization of all the applications except hotspot is low. The Gaussian kernel takes the maximum time to execute, followed by SRAD. The computation and read/ write memory transfers of all kernel contribute to the execution time. A similar pattern is shown for SRAD's bandwidth as well. The average bandwidth utilization of NN and Gaussian are somewhat similar and follows the SRAD while the least is by hotspot.

OpenCL 2.0 specification introduces a new memory object called a pipe. A pipe stores data organized as a FIFO and can be used to stream data from one kernel to another inside the FPGA device without having to use the external memory, which greatly improves the overall system latency. The depth of the pipe is the length of

Table 6.2: 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 FIFO. OpenCL 2.0 allows us to specify the length of the pipe.

Listing 6.2: 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 6.3: 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));
}
```

Next, Listing 6.2 and 6.3 show us the template used for splitting the kernels. Listing 2 is just used for instantiating OpenCL pipes with a user-defined specific Pipe depth. Xilinx SDAccel toolchain allows us to fix pipe depths in powers of two up to 32768.

Listing 4.3 shows the baseline kernel splitting into 'kernel read', 'kernel compute' and finally 'kernel write'. The transformed kernel code is called in the host source code encapsulated in the same OpenCL context. Inter-kernel communication and synchronization between threads are inherent to the property of OpenCL. Each queue is mapped into individual CU and have their own Thread dispatcher (TD) and Load store unit (LSU) and run concurrently.

CHAPTER 7: RESULTS

Table 2.1 lists the parameters of our FPGA platform. We use Xilinx SDAccel HLS for OpenCL[19] for compiling and synthesizing the OpenCL code. The SDAccel profiler collects kernel performance data, bandwidth efficiency of global memory, resource utilization and power. We also use the AMD FirePro W7100 GPU to report our GPU results for kernel performance and the AMD CodeXL Power profiler for power information.

A total of 15 massively parallel applications comprising of different classes and access patterns (Table 7.1) from the Xilinx SDAccel repository[42] and the Rodinia Benchmark suite[6] are used for our experiments. This gives a fair chance to compare performance across single work-item kernels favoring FPGAs and multiple work item kernels written for GPUs. The baseline profiling information is shown in Table 7.1.

7.0.1 Optimization performed over baseline

The Xilinx SDAccel source repository and Rodinia benchmarks provide some optimizations on the naive OpenCL codes making the algorithm efficient[19]. We include these optimized codes as our baseline implementation. These optimizations (Table 7.1) includes loop pipeline (A), asynchronous workgroup copy (B), array partition (C), function inline (D), loop unroll (E), DDR channel (F) and burst transfer (G).

7.0.2 Performance Evaluation

Table 7.1 shows the class of each application where it belongs to and its optimization information along with the memory access pattern. It shows that the application with really low bandwidth has an irregular access pattern. The application with irregular memory access has longer memory access latency. Table 7.2 shows the

Table 7.1: Baseline profiling information for each application

Benchmarks	**Class of application**	**Baseline optimization**	**Memory access pattern**
Affine	Image processing	A, B	Irregular
AES	Security	C, D	Regular
Median Filter	Non-linear digital filtering	A, B, E	Irregular
Histogram	Image processing	A, C	Regular
Tiny Encryption	Security	A, B, C	Regular
Watermarking	Image processing	A, E, F	Regular
Systolic Array	Array architecture	A, C, G	Regular
Large Loop OCL	Convolution layer of CNN	A, G	Regular
Nearest Neighbor	Data mining	A	Regular
LUD Diag	Linear Algebra	A, G	Irregular
SRAD Extract	Image processing	A	Regular
Gaussian Fan1	Linear Algebra	A	Regular
Gaussian Fan2	Linear Algebra	A	Regular
Hotspot	Physics Simulation	A, C, E	Regular
BFS	Graph Algorithms	A, E	Irregular

Table 7.2: Baseline profiling information for each application

Benchmarks	Resource utilization (%)				Execution Time (ms)	Avg bandwidth (%)	Power (Watt)
	LUTs	LUTMem	REG	BRAM			
Affine	1.18	0.25	0.79	0.75	8.04	39.09	35.1
AES	0.66	0.26	0.36	2.06	2.21	100	34.81
Median Filter	0.74	0.26	0.5	3.05	1.14	6.79	35.01
Histogram	4	0.38	2.27	8.2	2.52	96	36.68
Tiny Encryption	4.86	0.18	2.24	3.66	7.33	59.1	36.9
Watermarking	0.45	0.25	0.31	0.91	0.29	81	37
Systolic Array	2.65	0.73	1.1	0.09	0.4	0.51	35.14
Large Loop OCL	0.56	0.22	0.48	21.72	2391	100	39.08
Nearest Neighbor	0.37	0.22	0.25	0.09	5.68	9.6	34.9
LUD Diag	0.34	0.19	0.2	0.23	7.74	16.9	35
SRAD Extract	0.33	0.19	0.16	0.05	316.5	56.7	35
Gaussian Fan1	0.84	0.44	0.46	0.09	1.52	24.9	34.7
Gaussian Fan2	0.87	0.36	0.56	0.19	0.2	19.7	35
Hotspot	0.69	0.21	0.37	8.06	47.63	4.58	34
BFS	0.73	0.36	0.45	0.19	2.49	9.6	35

report generated by the Xilinx SDAccel software. We see that the average power is 36 Watt. These are the performance result of a single compute unit.

7.1 Automatic Compute unit replication on cloud FPGA

To improve the performance we split the compute unit into multiple compute unit and 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 7.1. We get a maximum performance improvement of 53.1× over baseline implementation for the LUD Diag application and a 6.4× on average. However, we either get a

constant or very little speed up for a few of the applications like AES, Watermarking, Gaussian Fan1, etc., This can be attributed to 3 factors-baseline execution time, optimizations, bandwidth usage.

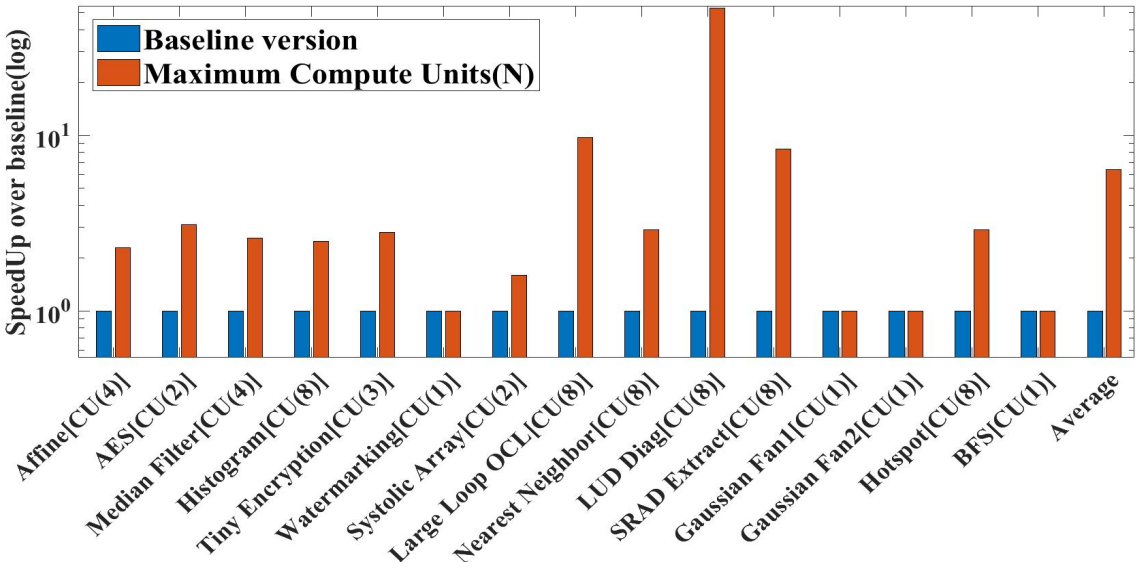


Figure 7.1: Performance improvement over the baseline

As an illustration, the Gaussian Fan2 application's baseline performance numbers 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 counterproductive to its performance.

LUD Diag, on one hand, has low bandwidth utilization (Table 7.2) and uses burst transfer (Table 7.1) where the data transfer happens in larger chunks and therefore benefits the most (21× improvement) from this approach. AES, on the other hand, has a very high baseline bandwidth utilization Table 7.2. This makes bandwidth a limiting factor hindering its ability to improve speed up.

Next, we observe the total design time that our tool takes to reach the optimal CU number for maximum speedup in Figure 7.2. The total design time for the tool to reach the optimal number of CUs is given as in Eq. 7.1:-

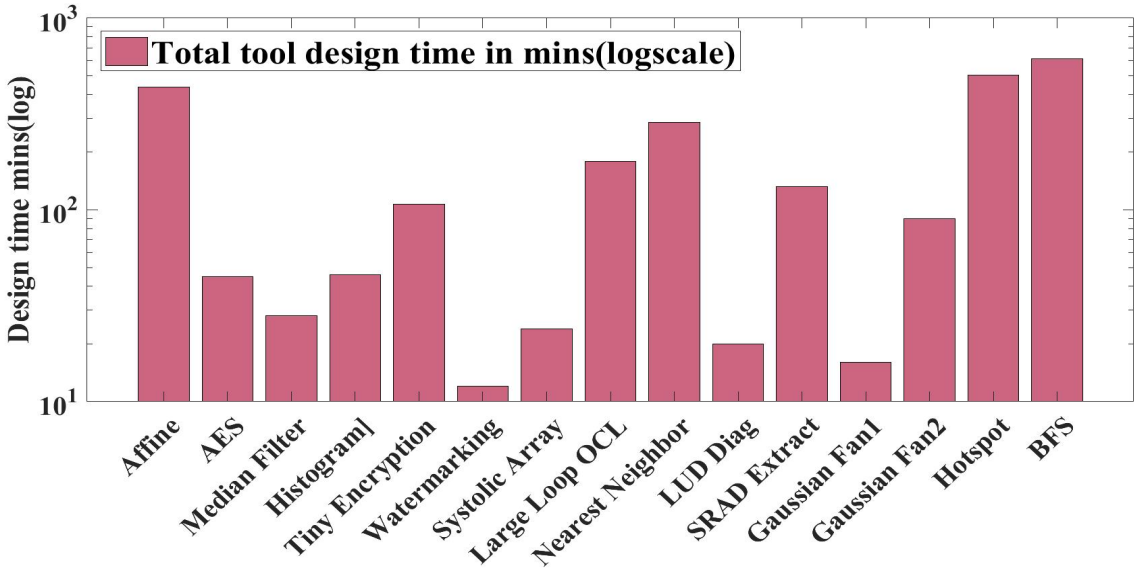


Figure 7.2: Total tool design time

$$TotalD_{\mathrm{T}} = \sum_{i=1}^{N} D_{\mathrm{T}}(CU) + D_{\mathrm{T}}(CU_{\mathrm{N+1}}) \tag{7.1}$$

where,

D_{T} (CU) shows the tool design time for each iteration of the compute unit.

We correlate this with the design time of the Histogram application (Figure 7.2) that takes 46m30s and additional synthesis time (for max CU[8]) of 300m totaling 346m30s. In contrast, the total design space exploration time (Table 4.1) is 1095m. Our tool achieves over **31%** design time improvement for the Histogram application.

The tool design time (D_{T}) is a factor of the size of the application (dataset), computation demands and memory access patterns. Overall, we observe that applications with regular memory accesses (Table 7.1) 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 (D_{T}).

7.1.1 Resource 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 7.3 shows the average percentage of 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.

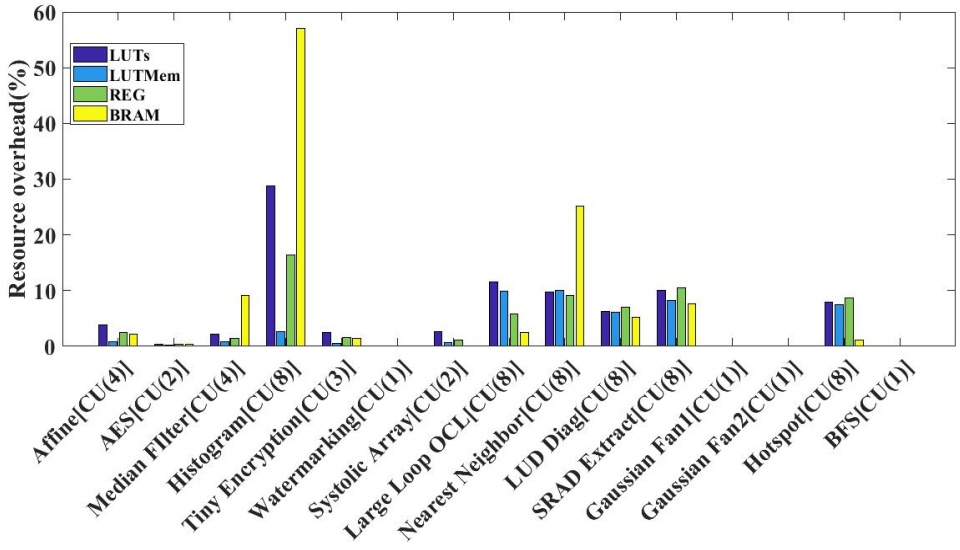


Figure 7.3: Percentage resource utilization overhead over baseline

7.1.2 Power Overhead

Power results (Figure 7.4) 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 3.6× over baseline owing to its larger BRAM utilization (Table 7.2 and Figure 7.3). However for most of the applications with very miniscule baseline resource utilization (Table 7.2), adding more CUs does not affect power as much. The average power overhead was thus reported a mere 1.33× over baseline.

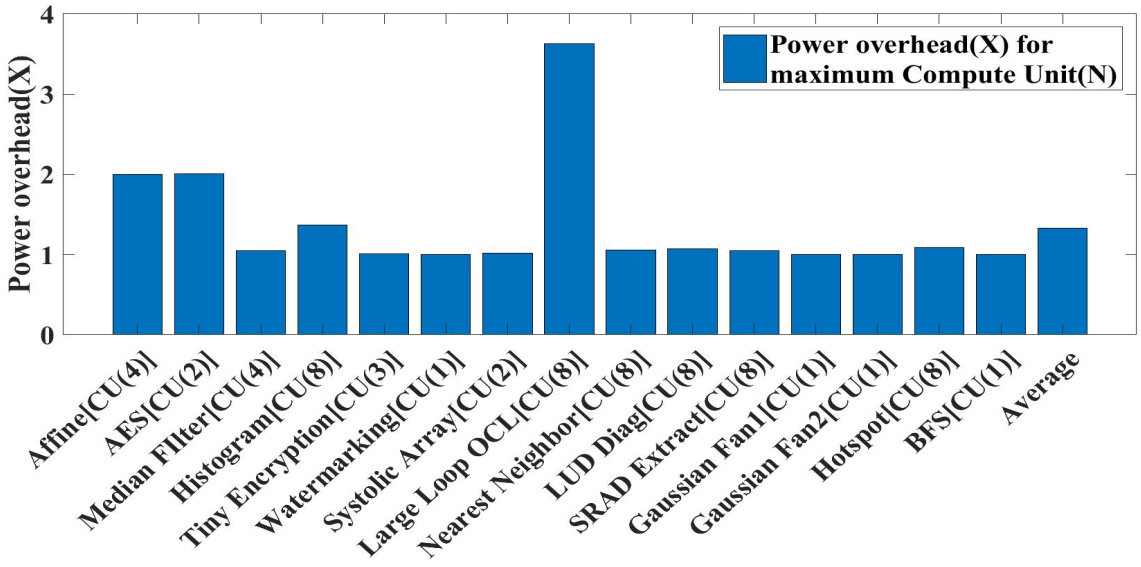


Figure 7.4: Power overhead over baseline

7.1.3 Bandwidth utilization increase over baseline

Bandwidth utilization increase Figure 7.5 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 7.2). 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 21× rise in bandwidth leading to 2.8× speed-up. On average we observe a 3.8× improvement in bandwidth utilization numbers.

7.1.4 FPGA vs GPU comparison

In this section (Table 7.3), we give a performance perspective for all of our applications. While Rodinia applications [6] 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 [43] to the FPGA.

While our FPGA best performance beats baseline across most of the applications,

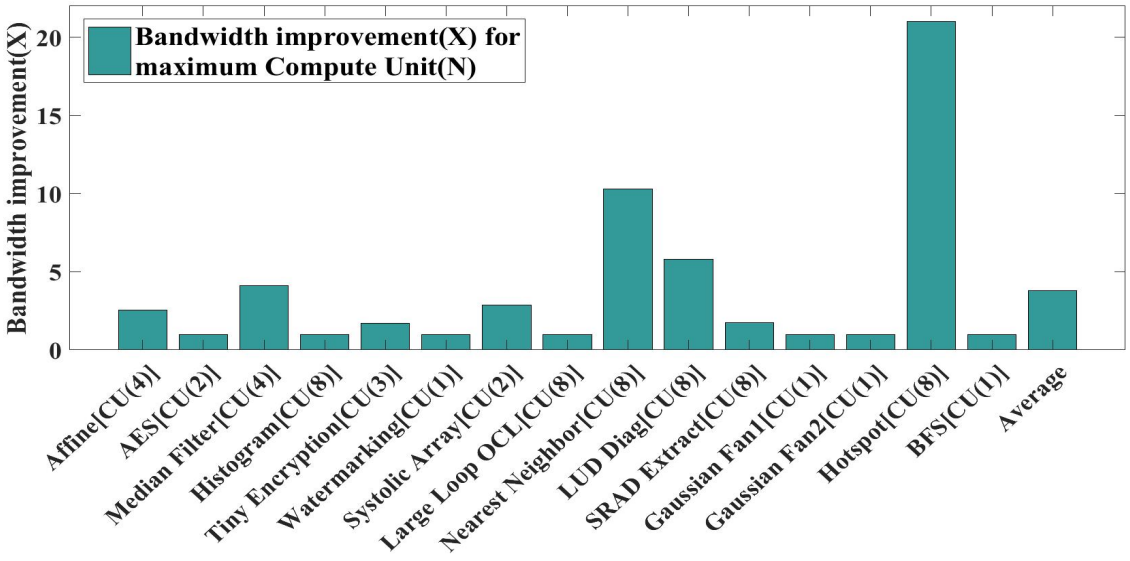


Figure 7.5: Bandwidth improvement over baseline

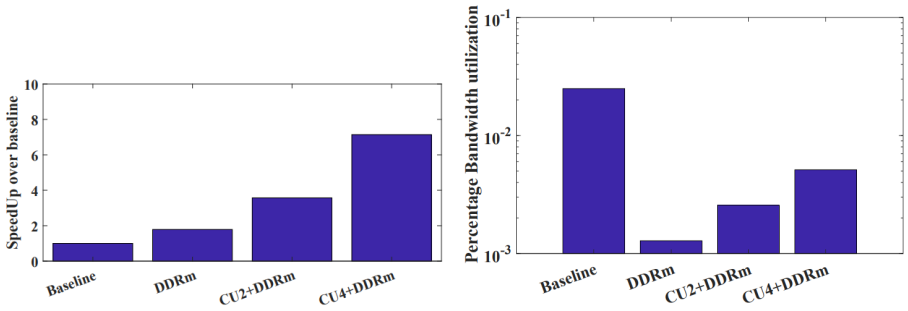
GPU performance is comparable to the FPGA numbers except for a few of the applications where GPU beats FPGA by a huge margin. Dynamic power(W) numbers of the applications on both the platforms listed in Table 7.2 are comparable in nature.

Table 7.3: 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

7.2 Exploring DDR optimization and burst transfer

Optimization enables efficient transfer of OpenCL data. The host code and the kernel code needs to be changed for a better data transfer. There are multiple optimizations available for better data transfer. Double Data rate and Burst transfer are some of the data transfer optimizations.



(a) Speed Up over Baseline (b) Bandwidth Utilization

Figure 7.6: Large Loop OCL with SpeedUp and Bandwidth

7.2.1 Using DDR

Double Data Rate (DDR) is used for the application which needs larger data to be transferred. An application such as Large Loop OCL is a data-hungry neural network application. The efficiency of the data transfer can be improved by splitting up the global memory inside the kernel to separate DDR.

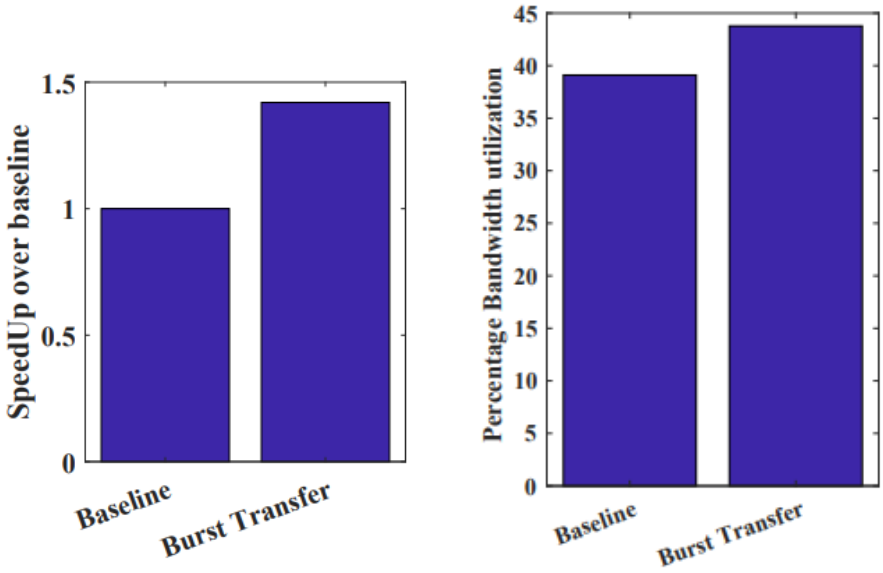
Figure 7.6(b) shows that there is a sudden drop in the bandwidth because of the additional DDR added to it. Adding a compute unit to increase the amount of data transferred per each DDR channel so there is a significant increase in the percentage of Bandwidth when compute unit increases.

The Large Loop OCL shows a better speed up when read and write channels are separated. Increasing the compute unit along with the DDR improves the speedup as shown in Figure 7.6(a)

7.2.2 Using Burst Transfer

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. Figure 7.7(b) shows that there is an increase in the bandwidth because of continuous access to memory from global to the local memory.

The figure 7.7(a) shows that the performance increase because of burst transfer.



(a) Speed Up over Baseline (b) Bandwidth Utilization

Figure 7.7: Affine with Burst transfer

The compute unit accessed the data needed for computation from the local memory.

7.3 Exploring OpenCL pipes

We used eight standard applications from Rodinia Benchmark suite. They namely are Nearest Neighbors, Srad base (Srad extract application), Gaussian, B+Tree, LUD Diagonal, LUD internal and Hotspot. Our FPGA implementations are synthesized on the Virtex Ultrascale FPGA while we have used the AMD FirePro W7100 device for our GPU implementation.

7.3.1 Performance and Resource overhead

In Figure 7.8, the performance improvement of Rodinia benchmark with the pipeline implemented over the baseline performance is shown. The average speedup of performance improvement is 6 times over the baseline. LUD diagonal achieved maximum performance improvement, whereas the LUD internal showed the least performance improvement. LUD internal has more local variable which causes stalls when compared to LUD diagonal. As a result, stalls increase the execution time of the kernel

and thus reducing the performance. We can see that applications have a considerable increase in performance based on the number of a local variable in the applications.

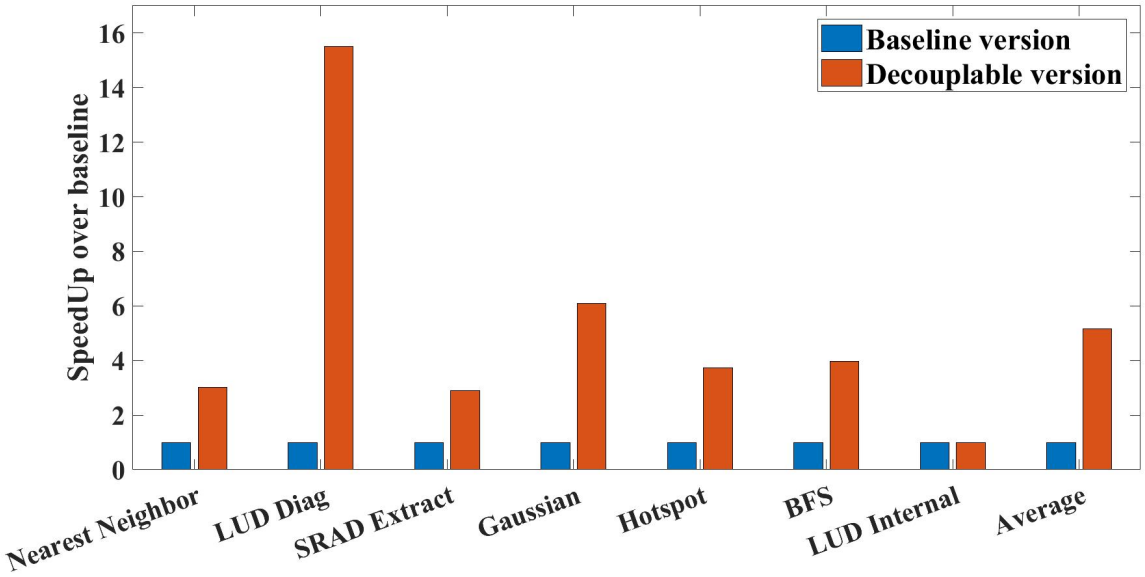


Figure 7.8: Performance improvement over baseline

The total utilization of LUT, Registers, BRAM and LUT memory is considered for resource utilization. LUT, Registers, BRAM and LUT memory are required for the pipe construction. As a result, there is an increase in resource utilization when compared to the baseline. The average utilization of LUT and LUT Memory is around 1.75% while register and BRAM is 1.5% and 0.5% as shown in Figure 7.9. The resource utilization is based on the number of pipes created and the depth of the pipe. Gaussian shows the highest resource utilized because of it uses 7 pipes for kernel communications. BFS and Hotspot have the lowest resource utilization because of less number of pipes.

Multiple global variables are accessed using different pipes which also contributes to the increase in register blocks, memory blocks, and many logic gates. Figure 7.10 shows the average Bandwidth increase of the pipeline version over the baseline version. Overall, we observe a significant increase in bandwidth utilization which leads to a reduction in memory stalls. The average bandwidth consumed by the pipeline

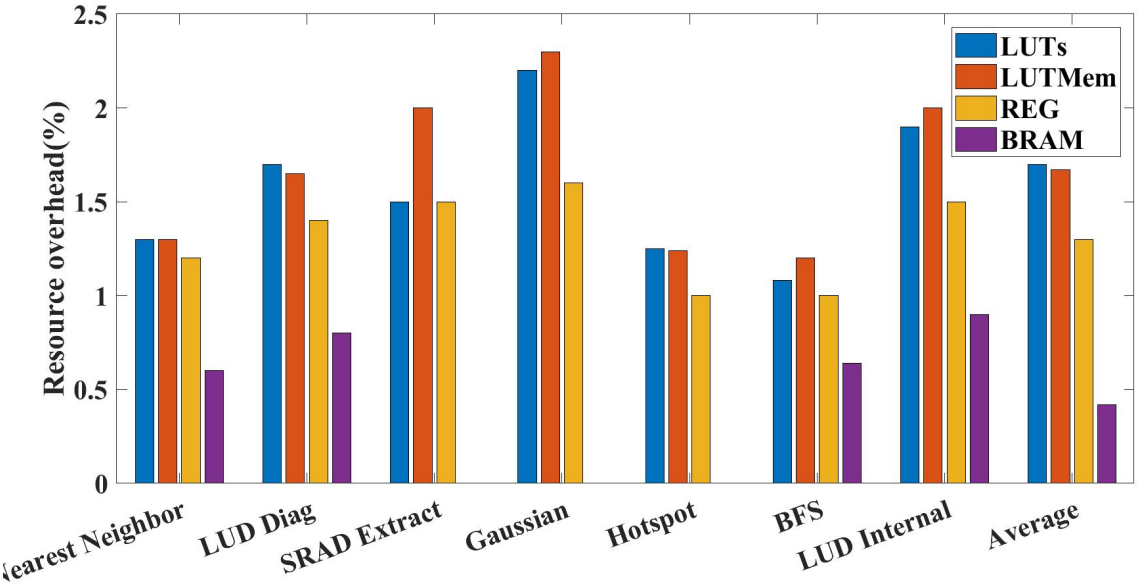


Figure 7.9: Resource overhead over baseline

version is 2× more than that of the baseline. The LUD Diag and LUD Internal consume the maximum bandwidth while Gaussian consumes the least among the seven applications.

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.

Table 7.4 compares the performance numbers of FPGA-baseline, proposed decoupled execution (FPGA best), and GPU. The GPU performance that we observe is comparable to the FPGA numbers except for a few of the applications where GPU beats FPGA by a huge margin. This concludes that despite FPGAs are bandwidth limited they perform comparably well alongside GPUs. With an increase in band-

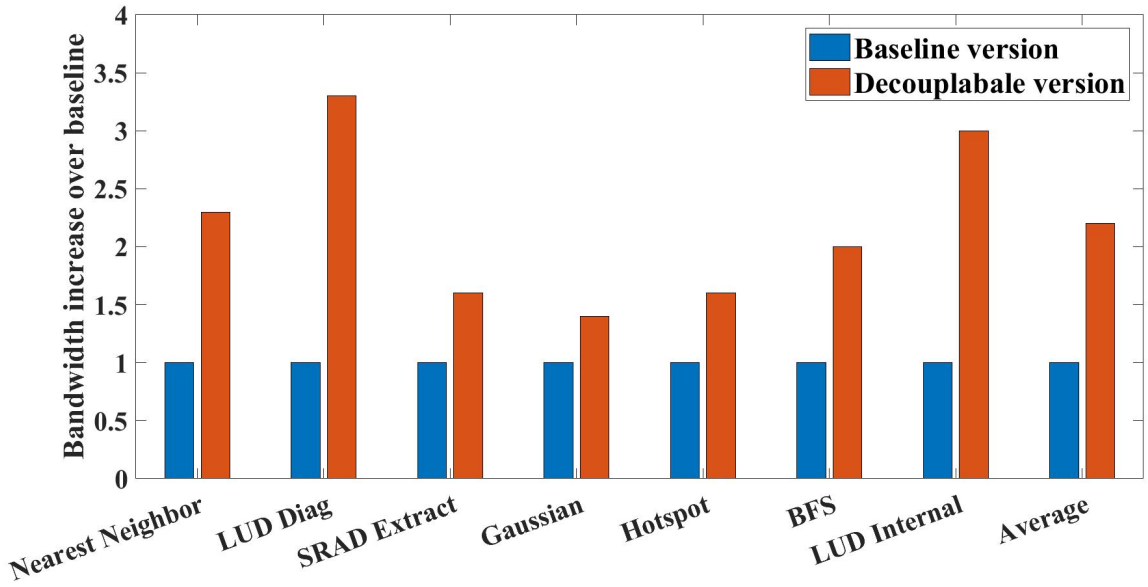


Figure 7.10: Bandwidth improvement over baseline

width utilization capacity, FPGAs can surely outperform GPUs.

Table 7.4: 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

CHAPTER 8: CONCLUSIONS

This research explores various optimization techniques that can be applied to improve thread-level utilization, performance, and occupancy on FPGA. The focus of our study is to exploit the spatial parallelism on top of the temporal parallelism on FPGA. The work initially focused on exploring CU replication with the same kernel. We also studied that replicating multiple CUs may not give the maximum performance. In order to help the design space exploration, a fully automatic tool is used to identify the number of CUs that can be used for an application. At the same time, a systematic OpenCL programming template for streamlining CU replication at the source level is proposed. Overall, our results demonstrate a maximum of 53X and 6.4X average speedup over 15 massively parallel OpenCL applications. Further, our tool achieves over 31% design time improvement for the Histogram application that serves as an illustration in Chapter 4. Then we explore other optimizations like Double Data rate to increase the channel bandwidth and Burst transfer to increase Bandwidth. With DDR, we could achieve 7.5× speedup. Affine which is a image processing application is optimized with burst transfer which produced 1.5× performance improvement in chapter 5. Further proposed sub-kernel parallelism to hide the memory access latency of massively parallel applications running on FPGA. The memory access latency is hidden by prefetching the data which will be used by future threads and concurrently executing the current thread. This is done by decoupling the kernels into read/write kernel and computation kernel. The LLVM based OpenCL analyzer is used to identify the kernel's statically prefetchable data. The inter-kernel communication between the read/write kernel [41] and computation kernel is done by constructing OpenCL pipes. Our experimental results over seven

Rodinia applications running on Xilinx VU9P Cloud FPGAs demonstrate an average of 5.2× speedup with a 2.2× increase in bandwidth utilization and just under 2.5× resource utilization overhead compared to baseline implementation.

REFERENCES

[1] B. McCullough, "An eye-opening look at the dot-com bubble of 2000 â and how it shapes our lives today." `https://ideas.ted.com/an-eye-opening-look-at-the-dot-com-bubble-of-2000-and-how-it-shapes-our-lives-today`, 2018. Last accessed 2020.

[2] M. F. P. C. J. Lato; and P. Paul Syers, "The Death of Moores Law." `http://www.potomacinstitute.org/steps/images/PDF/Articles/FritzeSTEPS_2016Issue3.pdf`, 2016. Last Accessed 2020.

[3] R. Weber, A. Gothandaraman, R. J. Hinde, and G. D. Peterson, "Comparing hardware accelerators in scientific applications: A case study," *IEEE Transactions on Parallel and Distributed Systems*, vol. 22, pp. 58–68, Jan 2011.

[4] K. O. W. Group, "Introduction to the opencl programming model." `http://aiweb.techfak.uni-bielefeld.de/content/bworld-robot-control-software/`, 2010.

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

[6] 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, (USA), p. 44â54, IEEE Computer Society, 2009.

[7] D. Chen, "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.

[8] J. Andrade, G. Falcao, 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.

[9] 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.

[10] 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 '17, (New York, NY, USA), pp. 25–34, Association for Computing Machinery, 2017.

[11] C. Zhang, P. Li, G. Sun, Y. Guan, B. Xiao, and J. Cong, "Optimizing fpga-based accelerator design for deep convolutional neural networks," in *Proceedings of the 2015 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, FPGA '15, (New York, NY, USA), pp. 161–170, Association for Computing Machinery, 2015.

[12] 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 sdk," in *2014 International Conference on Field-Programmable Technology (FPT)*, pp. 326–329, Dec 2014.

[13] A. Pacholik, M. Muller, W. Fengler, T. Machleidt, and K. . Franke, "Gpu vs fpga: Example application on white light interferometry," in *2011 International Conference on Reconfigurable Computing and FPGAs*, pp. 481–486, Nov 2011.

[14] M. Yih, J. M. Ota, J. D. Owens, and P. Muyan-Ozcelik, "Fpga versus gpu for speed-limit-sign recognition," in *2018 21st International Conference on Intelligent Transportation Systems (ITSC)*, pp. 843–850, Nov 2018.

[15] H. R. Zohouri, N. Maruyama, A. Smith, M. Matsuda, and S. Matsuoka, "Evaluating and optimizing opencl kernels for high performance computing with fpgas," in *SC '16: Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, pp. 409–420, Nov 2016.

[16] "Altera SDK for OpenCL.." `https://www.altera.com/products/design-software/embedded-software-developers/opencl/overview.html`, 2015. Last accessed 2020.

[17] "Xilinx OpenCL." `http://www.xilinx.com/products/design-tools/softwarezone/sdaccel.html`. Last accessed 2020.

[18] "Intel SDK for OpenCL Applications." `https://www.altera.com/en_US/pdfs/literature/hb/opencl-sdk/aocl_programming_guide.pdf`, 2017. Last accessed 2020.

[19] "SDAccel Environment Optimization Guide." `https://www.xilinx.com/support/documentation/sw_manuals/xilinx201_4/ug1207-sdaccel-optimizationguide.pdf`, 2017. Last accessed 2020.

[20] F. M. Vallina and S. Gilliland, "Performance optimization for a sha-1 cryptographic workload expressed in opencl for fpga execution," in *Proceedings of the 3rd International Workshop on OpenCL*, IWOCL '15, (New York, NY, USA), Association for Computing Machinery, 2015.

[21] 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), Association for Computing Machinery, 2018.

[22] G. Guidi, E. Reggiani, L. D. Tucci, G. Durelli, M. Blott, and M. D. Santambrogio, "On how to improve fpga-based systems design productivity via sdaccel," in *2016 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*, pp. 247–252, May 2016.

[23] R. J. Halstead and W. Najjar, "Compiled multithreaded data paths on fpgas for dynamic workloads," in *2013 International Conference on Compilers, Architecture and Synthesis for Embedded Systems (CASES)*, pp. 1–10, Sep. 2013.

[24] J. Zhang and J. Li, "Improving the performance of opencl-based fpga accelerator for convolutional neural network," in *Proceedings of the 2017 ACMInternational Symposium on Field-Programmable Gate Arrays*, FPGA '17, (New York, NY, USA), p. 25â34, Association for Computing Machinery, 2017.

[25] R. Halstead, J. Villarreal, and W. Najjar, "Exploring irregular memory accesses on fpgas," pp. 31–34, 11 2011.

[26] 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.

[27] S. Wang, Y. Liang, and Wei Zhang, "Flexcl: An analytical performance model for opencl workloads on flexible fpgas," in *2017 54th ACM/EDAC/IEEE Design Automation Conference (DAC)*, pp. 1–6, June 2017.

[28] G. Zhong, A. Prakash, Y. Liang, T. Mitra, and S. Niar, "Lin-analyzer: A high-level performance analysis tool for fpga-based accelerators," in *2016 53nd ACM/EDAC/IEEE Design Automation Conference (DAC)*, pp. 1–6, June 2016.

[29] J. Zhao, L. Feng, S. Sinha, W. Zhang, Y. Liang, and B. He, "Performance modeling and directives optimization for high level synthesis on fpga," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, pp. 1–1, 2019.

[30] A. Momeni, H. Tabkhi, Y. Ukidave, G. Schirner, and D. Kaeli, "Exploring the efficiency of the opencl pipe semantic on an fpga," *SIGARCH Comput. Archit. News*, vol. 43, p. 52â57, Apr. 2016.

[31] S. O. Settle, "High-performance dynamic programming on fpgas with opencl," 2013.

[32] M. Shevgoor, S. Koladiya, R. Balasubramonian, C. Wilkerson, S. H. Pugsley, and Z. Chishti, "Efficiently prefetching complex address patterns," in *2015 48th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pp. 141–152, Dec 2015.

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

[34] 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, (New York, NY, USA), p. 1â11, Association for Computing Machinery, 2004.

[35] 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 â14, (New York, NY, USA), Association for Computing Machinery, 2014.

[36] 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, (GBR), p. 19â26, IFIP Working Group on Algol, 1995.

[37] A. C. Klaiber and H. M. Levy, "An architecture for software-controlled data prefetching," *SIGARCH Comput. Archit. News*, vol. 19, pp. 43–53, Apr. 1991.

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

[39] 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), Association for Computing Machinery, 2017.

[40] A. Purkayastha, S. Rogers, S. Shiddhibhabi, and H. Tabkhi, "LLVM-Based Automation of Memory Decoupling for OpenCL applications on FPGAs," *Microprocessors and Microsystems*, p. 102909, 10 2019.

[41] 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), Association for Computing Machinery, 2018.

[42] "Xilinx SDAccel Repository." `https://github.com/Xilinx/SDAccel_Examples/tree/2017.4/getting_started`. Last accessed 2020.

[43] "AMD FirePro W7100 specifications." `https://www.amd.com/en/products/professional-graphics/firepro-w7100`, 2019. Last accessed 2020.