1-27-2016

FPGA Based Acceleration of Matrix Decomposition and Clustering Algorithm Using High Level Synthesis

Qing Yun Tang
University of Windsor

FPGA Based Acceleration of Matrix Decomposition and Clustering Algorithm Using High Level Synthesis

By

Qing Yun Tang

A Thesis
Submitted to the Faculty of Graduate Studies
through the Department of Electrical and Computer Engineering
in Partial Fulfillment of the Requirements for
the Degree of Master of Applied Science
at the University of Windsor

Windsor, Ontario, Canada

2016

FPGA Based Acceleration of Matrix Decomposition and Clustering Algorithm Using High Level Synthesis

by

Qing Yun Tang

APPROVED BY:

_____

T. Bolisetti
Department of Civil and Environmental Engineering

_____

R. Rashidzadeh
Department of Electrical and Computer Engineering

_____

M. Khalid, Advisor
Department of Electrical and Computer Engineering

January 12th 2016

Declaration of Co-Authorship / Previous Publication

I. Co-Authorship Declaration

This thesis incorporates the outcome of a joint research undertaken in collaboration with Ian Janik under the supervision of Dr. Mohammed Khalid. The collaboration is covered in Chapter 2 of the thesis. In all cases, the key ideas, primary contributions, experimental designs, data analysis and interpretation, were performed by the author and co-authors as result of joint research.

I am aware of the University of Windsor Senate Policy on Authorship and I certify that I have properly acknowledged the contribution of other researchers to my thesis, and have obtained written permission from each of the co-author(s) to include the above material(s) in my thesis.

I certify that, with the above qualification, this thesis, and the research to which it refers, is the product of my own work.

II. Declaration of Previous Publication

This thesis includes 2 original papers that have been previously published/submitted for publication in peer reviewed journals, as follows:

Thesis Chapter	Publication title/full citation	Publication status*
Chapter 1 and 2	I. Janik, Q. Tang, and M. Khalid, "An Overview of Altera SDK for OpenCL: A User Perspective," in Electrical and Computer Engineering (CCECE), 2015 IEEE 28th Canadian Conference on, pp. 559-564, 3-6 May 2015.	*published*
Chapter 3	Q. Tang and M. Khalid, "Acceleration of K-means Algorithm using Altera SDK for OpenCL"	*In preparation*

I certify that I have obtained a written permission from the copyright owner(s) to include the above published material(s) in my thesis. I certify that the above material describes work completed during my registration as graduate student at the University of Windsor.

I declare that, to the best of my knowledge, my thesis does not infringe upon anyone's copyright nor violate any proprietary rights and that any ideas, techniques, quotations, or any other material from the work of other people included in my thesis, published or otherwise, are fully acknowledged in accordance with the standard referencing practices. Furthermore, to the extent that I have included copyrighted material that surpasses the bounds of fair dealing within the meaning of the Canada Copyright Act, I certify that I have obtained a written permission from the copyright owner(s) to include such material(s) in my thesis.

I declare that this is a true copy of my thesis, including any final revisions, as approved by my thesis committee and the Graduate Studies office, and that this thesis has not been submitted for a higher degree to any other University or Institution.

Abstract

FPGAs have shown great promise for accelerating computationally intensive algorithms. However, FPGA-based accelerator design is tedious and time consuming if we rely on traditional HDL based design method. Recent introduction of Altera SDK for OpenCL (AOCL) high level synthesis tool enables developers to utilize FPGA's potential without long development time and extensive hardware knowledge.

AOCL is used in this thesis to accelerate computationally intensive algorithms in the field of machine learning and scientific computing. The algorithms studied are k-means clustering, k-nearest neighbour search, N-body simulation and LU decomposition. The performance and power consumption of the algorithms synthesized using AOCL for FPGA are evaluated against state of the art CPU and GPU implementations. The k-means clustering and k-nearest neighbor kernels designed for FPGA significantly out-performed optimized CPU implementations while achieving similar or better power efficiency than that of GPU.

Acknowledgements

First and foremost, I would like to express my deepest gratitude to my supervisor Dr. Khalid for his patient support over the past two years. His encouragement and kind guidance helped me to surpass difficulties encountered during study and research. I am really fortunate to have him as my mentor and advisor.

I would like to thank Dr. Rashid Rashidzadeh and Dr. Tirupati Bolisetti for taking the time from their busy schedule to be part of my thesis committee, and for providing insightful comments to improve this work.

I would like to give special thanks to Ian Janik for his technical assistance and suggestions on improving this work. It was my pleasure working with him.

I am also grateful to Dr. Roberto Muscedere for his help in maintaining the workstations in the lab and providing encouragement and insights on this research.

Most importantly this thesis is dedicated to my parents. This work would not be possible without their continued support and care.

This research was supported by Natural Sciences and Engineering Research Council of Canada (NSERC), the Canadian Microelectronics Corporation (CMC) and Altera Corporation. I would like thank these organizations for providing us with funding, equipment, and CAD software necessary for this research.

Table of Contents

List of Tables

List of Figures

List of Acronyms

ALM	Adaptive Logic Module
AOCL:	Altera SDK for OpenCL
AOC:	Altera Offline Compiler
API:	Application Programming Interface
ASIC:	Application Specific Integrated Circuits
CPU:	Central Processing Unit
CUDA:	Compute Unified Device Architecture
DSP:	Digital Signal Processor
FPGA:	Field Programmable Gate Array
GPU:	Graphics Processing Unit
HDL:	Hardware Description Language
HLS:	High Level Synthesis
HPC:	High Performance Computing
KNN:	K-Nearest Neighbors
LAB:	Logic Array Block
LE:	Logic Element
LUT:	Lookup Table
OpenCL:	Open Computing Language
OpenMP:	Open Multi-Processing
OpenMPI:	Open Message Passing Interface
SIMD:	Single Instruction Multiple Data
SPMD:	Single Program Multiple Data

Chapter 1

Introduction

1.1 Motivation

Ever since the invention of the first silicon integrated circuit, performance and capabilities of microprocessors have been increasing at a staggering pace. In 1965, Golden Moore made the prediction [1] that the number of transistors in a single integrated circuit would double every eighteen months. During the course of the last fifty years, the trend in semiconductor development has proved him correct. As a result microprocessor with unprecedented computation power has become increasingly cost effective.

However, in recent years shrinking down the transistor size has become increasingly difficult [2]. At the same time the demand for high performance yet power efficient microprocessors is increasing, due to emerging applications in various fields such as mobile computing, machine learning, data mining and computer graphics. In future, simply adding more computational devices and memory into a processor may no longer be the best way of increasing performance. Thus smarter alternative solutions will be necessary. Introduction of recent generation of Field Programmable Gate Arrays (FPGAs) with built in floating point DSP blocks enables FPGAs to accelerate computationally intensive problems, and compete with traditional CPU and GPU based computing platforms. Unlike a CPU or GPU, an FPGA does not have an instruction set or fixed pipeline built in. Instead it has large amount of reconfigurable logic that could be configured to perform any kind of digital logic function. The advantage of FPGA is that

when solving different problems, an FPGA could be customized to efficiently solve each of the problems, and potentially achieve much faster speed and energy efficiency than CPU or GPU. At the same time comparing to Application Specific Integrated Circuits (ASICs), FPGA is much more flexible and cost far less to develop. The down side of FPGA is that traditionally, FPGA requires low level hardware description languages (HDLs) to program and is very tedious to debug. Essentially the developer has to make highly detailed description of the hardware architecture that they want the FPGA to implement. Thus FPGA development requires extensive hardware knowledge, and the development time is often far longer than developing software for CPUs or GPUs.

High level synthesis tools such as Altera SDK for OpenCL aim to reduce the difficulty of deploying FPGA computing solutions and makes an FPGA a more favorable computing platform. OpenCL stands for Open Computing Language, which is an industry standard parallel programming language for heterogeneous system. The OpenCL is supported by most CPU and GPU vendors in the past, and the recent introduction of Altera SDK for OpenCL (AOCL) extended its support to FPGA as well. In AOCL the developer writes the computationally intensive portion of the program into kernels. The program setup and the synchronization and control of kernels are written into the host program. The kernels are compiled by AOCL compiler and Quartus II into FPGA images prior to execution and are used to configure FPGA as the accelerator. The host program is compiled by GCC or visual C++ compiler into CPU binary and runs on the CPU. Since OpenCL is a high level programming language and the AOCL compiler takes care of generation the hardware description, the difficult of developing on AOCL SDK is much lower than hand coding HDL. As a result, the AOCL would allow developers to explore

more difficult algorithms to accelerate and try out more problem configurations in shorter amount of time.

Machine learning is one of the fastest growing areas of computer science today, and its applications span every facet of our daily life. Machine learning is already applied in fields such as search engines, data mining, computer vision, natural language processing, robotics, medical science and trading, with new applications being discovered every day. However, most machine learning algorithms are computationally intensive. In recent years a lot of research was done on porting machine learning algorithms to parallel and heterogeneous computing platforms. In many machine learning applications, running parallelized programs on GPU could give large speedup verses sequential or multi-threaded programs on CPU. However, high performance GPUs often consume considerable amount of power, and require a lot of effort to design cooling systems to effectively handle excessive heat dissipation. In addition, many types of computations are difficult to parallelize and have to run on CPU, thus incurring extra overhead to transfer data and synchronize between CPU and GPU. FPGA based acceleration may avert some of those problem due to low power nature of the FPGA, and the fact that efficient customized pipelines could be constructed on FPGA fine-tuned for the algorithm to be accelerated. Another advantage of FPGA high level synthesis platform is that AOCL allows the execution of sequential code and management of FPGA computing resources to be done on embedded ARM processor that is packaged into the FPGA. This enables lower latency memory access and sharing of memory between CPU and FPGA. In addition, due to low power consumption of ARM processor, the overall power profile of

FPGA accelerator could be far lower than CPU – GPU heterogeneous computing platforms.

However, high level synthesis also has limitations. The high level synthesis essentially designs hardware based on high level description of algorithms. The hardware that is generated automatically by software may not be as efficient as hardware designed by skilled computer engineers. Also, due to limitations of FPGA hardware such as much lower operation frequency and lower numbers of floating point units than GPU, not all algorithms will be efficient for FPGA acceleration and high level synthesis.

1.2 Thesis Objectives

The goal of this research is to accelerate computationally intensive applications such as matrix decomposition, clustering algorithms, and other machine learning and scientific computing related algorithms using Altera SDK for OpenCL high level synthesis tool on FPGA. The results in terms of throughput, total processing time and energy efficiency are compared with traditional multi-core computing platforms such as CPU and GPU. The advantages and disadvantages of AOCL along with CPU, GPU and FPGA platforms are also evaluated during this research. The research goals were achieved through six phases:

1. The fundamentals of parallel programming and Altera SDK for OpenCL platform were studied.
2. A survey of parallelizable computationally intensive algorithms was conducted and suitable algorithms for implementation using AOCL on FPGA were selected.
3. The algorithms were implemented on CPU directly for study.

4. Those algorithms were implemented on FPGA using AOCL and their correctness was verified with the CPU implementations.
5. Improvements were made to the base line FPGA implementations in order to achieve the best performance we could obtain.
6. The best versions of FPGA implementations of the algorithms were tested with available CPU and GPU implementations to compare performance and efficiency.

1.3 Thesis Outline

The reminder of the thesis is structured as follows: In Chapter 2, the background on high level synthesis and heterogeneous computing as well as architectures of multi-core processors and FPGA is discussed. A short introduction to AOCL and the algorithms that are implemented in this thesis is also given in Chapter 2.

Chapter 3 introduces the K-means clustering algorithm. A detailed report on the AOCL implementation of this algorithm that was designed during this research is given. A summary of the state of the art implementations is also provided. The results from those implementations are compared with the state of the art and discussed at the end of the chapter. Chapter 4 follows the same format as chapter 3, and describes the research done to accelerate k-nearest neighbor algorithm. Chapter 5 describes acceleration of N-body simulation. Chapter 6 describes acceleration of LU decomposition algorithms. Only brief discussion of the implementation and short comparison of synthesis result will be given for N-body simulation and matrix decomposition, as their result was not as good as expected. Lastly, the Chapter 7 provides a summary of the thesis and provides directions for related future work.

Chapter 2

Computing Platforms and CAD Tools

2.1 Parallel and Heterogeneous Computing

Traditionally, the performance of a processor could be increased in two simple ways: either through instruction level parallelism (ILP), which requires more complex and longer pipelines or by increasing the clock frequency of the processor. However, lengthy and complicated pipelines are often less efficient. At the same time increase in clock frequency for processors has almost stalled in recent years [3], due to the breakdown of Dennard scaling [4]. Dennard scaling predicts that as the size of transistor shrinks, the power efficiency would increase while the transistors could be clocked faster. However, since the release of Pentium 4 processors in 2005, increasing clock frequency has become very difficult due to excessive power consumption such action entails. This is known as the power wall. As a result, engineers turned to multi-core designs to increase performance of the processor, and parallel computing is becoming increasingly important ever since.

In 2006, researchers from University of California at Berkeley published "The Landscape of Parallel Computing Research: A View from Berkeley [5]," in which they reviewed major problems of computing, and summarized common programming models of parallel computing into 13 kernels that they called "dwarfs." The kernels cover most widely used applications of high performance computing. The techniques used to parallelize those 13 kernels could be applied to most parallel programming application. It turns out that pattern recognition and machine learning algorithms mostly use 6 out of 13

of these “dwarfs”. Namely dense linear algebra, sparse linear algebra, dynamic programing, MapReduce, backtrack and branch-and-bound, and graphic traversal. For image processing applications structured grid and spectral algorithms such as FFT and DCT are also very important. The algorithms that are used in FPGA acceleration in this research involves dense matrix, MapReduce and structured grid computational patterns.

Not all kinds of algorithm could be parallelized; some computation could be very difficult to parallelize and thus may run more efficiently on CPU. Heterogeneous computing systems solve this problem by allowing different kind of processors to work together. For instance, parts of the computation that is more suited to CPU will be computed on CPU, while the inherently parallel parts of the computation can be computed on GPU. Thus processors with different kinds of architectures could be utilized efficiently by only doing the work that they are best at. Altera SDK for OpenCL is a high level synthesis tool that extends heterogeneous computing to FPGAs. In this chapter, a brief overview of hardware and software used in high level synthesis and parallel computing will be given along with introduction to Altera SDK for OpenCL.

2.1.2 CPU and Multi-threading

Architecture

Central processing unit (CPU) is the most common computing device today. CPUs are optimized for latency. They usually have very high maximum clock frequency, and thus are able to execute instruction with very little latency. CPUs utilize instruction level parallelism (ILP) to increase performance. By exploiting pipeline parallelism and utilizing superscalar pipeline, a CPU could theoretically execute many instructions every clock cycle. Perfectly pipelined execution for instructions is not always possible. Data

dependencies or branches may cause pipeline to stall. In order to maximize utilization of the pipeline resources, CPUs often employ out of order execution to dynamically schedule instructions in the most efficient way possible. Sophisticated branch predictor is also used to speculate the outcome of branch instructions using statistics to prevent pipeline stalling. To further increase parallelism, CPUs usually support single instruction multiple data (SIMD) instructions for vectored operations. Those instructions allow concurrent execution of the same operation across multiple data. For example, modern Intel processors support MMX, SSE and AVX instructions. The main memory have rather limited bandwidth and high access latency, thus they are one of the most common limiting factor for performance. Therefore, CPUs often have large amount of high speed on chip cache along with complicated caching scheme to minimize memory operations to the lowest level of memory hierarchy. An example for CPU architecture is Intel Nehalem architecture shown in Figure 1.

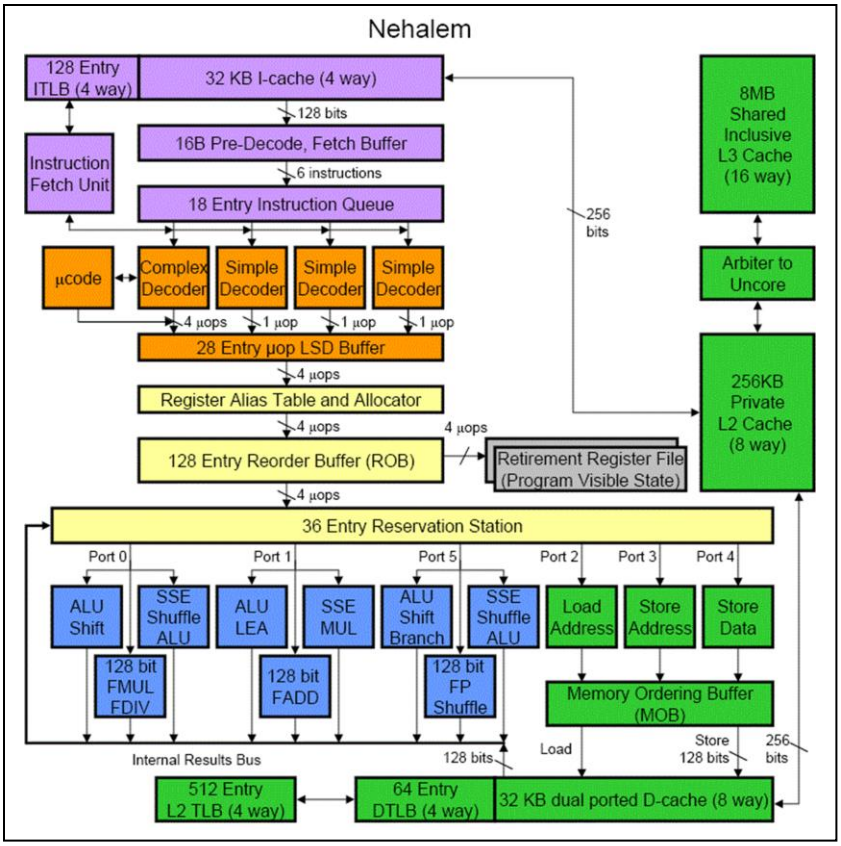

Figure 1. Intel Nehalem Architecture [6]

OpenMP and OpenMPI

To best utilize the capabilities of the CPUs and simplify writing parallel programs, Application Programming Interface (API) such as OpenMP [7] can be used. OpenMP stands for Open Multi-Processing [8], which is an API for parallel computing maintained by OpenMP Architecture Review Board. This API is an industry standard for multi-thread parallel programming with shared memory model across multiple platforms. It supports C/C++ and FORTRAN programming languages on most CPU architectures and operating systems. Most major compilers supports OpenMP, and it could be enabled by simply turning on a flag. In shared memory model, multiple processers share the same main memory resources. OpenMP API could be used through compiler directives and

library routines. The execution of OpenMP program could be controlled via environmental variables during runtime. When writing parallel programs with OpenMP, parallelism is expressed explicitly by forking and joining threads. The process of OpenMP program first starts with a single thread called the master thread. When the part of the computation designated for parallel execution is reached, parallel threads are launched and executed in parallel concurrently. The parallel threads synchronize and terminate after the parallel computation is complete, whereas the master thread continues to execute until next parallel region is reached. OpenMP could be used with other parallel and heterogeneous computing APIs such as OpenCL to help them to utilize CPU more efficiently.

Open Message Passing Interface (OpenMPI) [9] is a library for exchanging data between processors with distributed memory. In distributed memory model, each processor has its own independent main memory, and data exchange between processors has to be done explicitly; as opposed to shared memory model, where processors share main memory resources. Message passing and distributed memory will not be discussed in detail since this research focus on shared memory system with one processor and one FPGA.

2.1.1 GPU and Heterogeneous Computing

Architecture

GPUs are optimized for throughput. One of the most important performance metrics of a GPGPU is peak floating-point operations per second (FLOPS). Modern GPUs can compute thousands of floating point multiply and add each clock cycle. A state-of-the-art GPU can achieve throughput in the range of teraflops, due to its massively

parallel architecture, coupled with moderately high clock frequency of around 1 GHz. The rate for GPU performance progress outpaces CPU by a wide margin with no sign of slowing down any time soon.

Graphics processing units (GPUs) are traditionally used to provide hardware acceleration for 2D and 3D computer graphics applications. They are massively parallel. Due to the demand of increasingly realistic computer graphics, the performance of GPUs has been growing exceptionally fast. GPUs used to have dedicated hardware resource for processing different type of graphics computations, where each part of the GPU hardware maps to one stage in graphics pipeline. This design and lack of a user friendly programming language made programing GPUs for parallel computing very difficult. In late 2000s, GPUs started to adopt unified shader model, where different stages of graphics pipeline are processed by identical generic SIMD processors inside GPU. Together with the introduction of Compute Unified Device Architecture (CUDA) and OpenCL API has made GPU a powerful general purpose computing device. GPU architecture is significantly different from CPU architecture. In order to utilize task parallelism, CPUs dedicate large amount of transistors to complicated control units and cache. The floating point / integer execution and SIMD units which perform the actual computation occupy relatively small area of the CPU chip. On GPU however, majority of the silicon area is dedicated to SIMD units that are responsible for actual computation. A core in GPU has different meaning compared to CPU as well. While CPU cores are independent processors, a GPU core is similar to a single ALU in CPU. For example, in NVIDIA's Kepler architecture, 16 cores are grouped together in SIMD fashion. A Kepler equivalent of CPU core is called Streaming Multiprocessor (SMX), which contains 192

cores together with other memory and computation related resources. Each SMX is able to schedule concurrent execution of up to 8 wraps of SIMD instructions per clock cycle. Similar to CPUs, GPUs also utilize cache to minimize access of main memory. However, GPUs have much higher main memory bandwidth, but far smaller cache compare to CPUs. The NVIDIA Kepler architecture is shown in Figure 2.

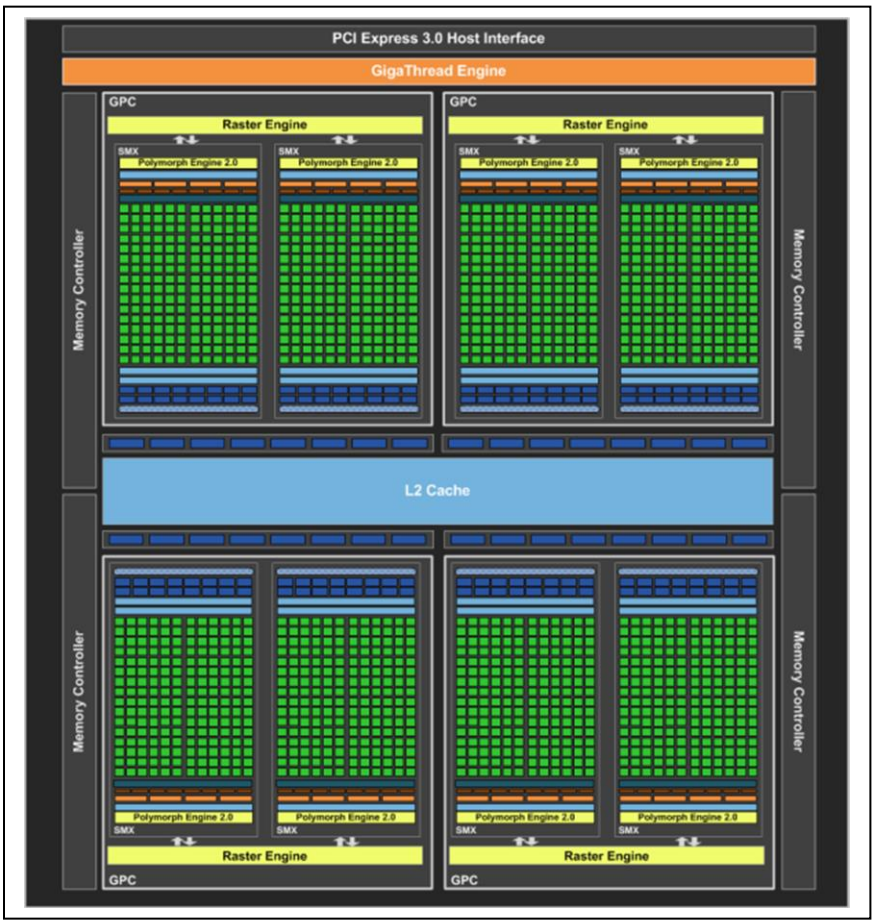

Figure 2. NVIDIA Kepler Architecture [10]

OpenCL and CUDA

Both OpenCL and CUDA are APIs that facilitates heterogeneous and parallel computing. The basic idea of CUDA and OpenCL is very similar. However, while CUDA is proprietary standard that only supports NVIDIA GPUs, OpenCL is an open

standard that has been adopted by most hardware manufactures. In this research, CUDA was only used in performance comparison, whereas Altera's implementation of OpenCL was used to construct all the parallel programs. Thus CUDA programming model will not be discussed in this thesis.

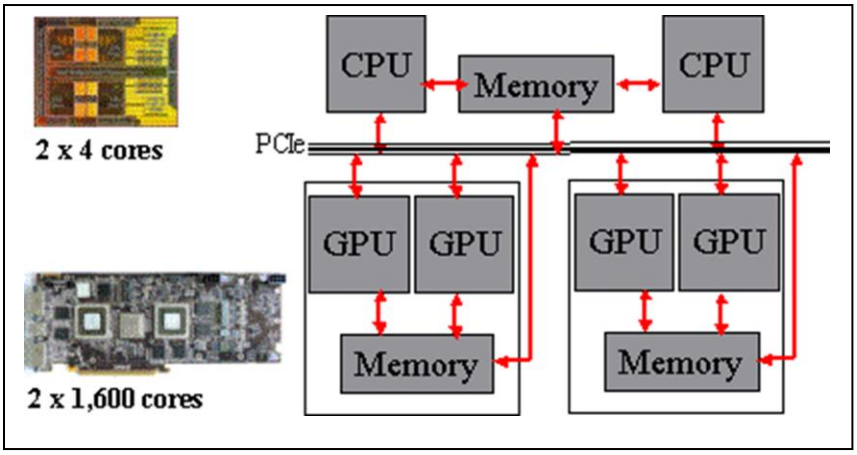

Figure 3. An Example of OpenCL Heterogeneous Computing Model [11]

OpenCL standard [12] was originally proposed by Apple, but is now maintained by Khronos Groups. Most CPU and GPU manufactures already implemented OpenCL API for developing parallel programs on their hardware, and recently FPGA and DSP vendors are starting to follow suit. In OpenCL programming model, programs are divided into two parts: the host program that runs on the CPU, and the kernels that run on the accelerators. The host program could be written in standard C or C++, where OpenCL specific functions are accessed through including OpenCL header file. It is mainly responsible for managing the memory and computational resource. The parallel computing kernels are written in a restricted subset of the C99 language, and are executed on accelerators. An example of OpenCL heterogeneous computing model with multiple CPUs and GPUs is shown in Figure 3.

The host could launch kernels in a way analogous to calling functions. To exploit the parallel architecture, kernels are usually launched in SPMD (single-program multiple data) fashion, where multiple instance of one kernel are organized into work-groups that runs on multiple processors in parallel, but each processing a different part of the data. Each kernel instance in the workgroup is called a work-item or thread. Work-items could be arranged in in one, two, or three dimensions, called the N-Dimensional range. In order to manage the kernel, context is defined in the host program. The context encapsulates computational resources including devices, kernels, program objects, and memory objects. It is created and can be modified by using OpenCL API functions. The type of accelerator is specified in device. The program object includes a set of kernel source and executable, where each kernel is a function that is to be executed on the device. The memory objects are created to move data between the host program and the kernel. The order of which the kernel execution and data transfer commands will proceed is controlled by command queues. Commands placed on the queue can be blocking or non-blocking, meaning a certain command could be halted until some commands have been completed, or run without waiting for anything. Each command placed on to a command queue is executed consecutively. In order to execute multiple kernels concurrently, multiple command queues are needed. Event objects can be used to synchronize concurrent tasks or profile performance.

There are four different types of memory available in OpenCL memory model: global, constant, local, and private. Global memory can be accessed by every work-item in all work-groups. It is both readable and writeable but transfer between host and kernel needs to be managed explicitly through OpenCL buffer objects and functions. Global

memory has very long access latencies compared to other memory types and can be the cause of bottlenecks in performance. Constant memory is optimized for high speed read only operations. It is faster comparing to global memory, but is not writeable by any kernel work-items. Local memory is usually allocated from on chip cache. It is relatively limited in size, but is has much lower access latency and far higher bandwidth than the global memory. This type of memory is only shared by work-items in the same workgroup, and is not accessible by the host program. Unlike global memory, local memory allows random access without heavy performance penalty. Finally, private memory is an area of memory that is accessible by only a single work-item. It is usually implemented by registers, and thus is the fastest type of memory available.

2.2 FPGA Architecture and Accelerator Hardware

2.2.1 FPGA Architecture

FPGA stands for Field Programmable Gate Array. Unlike CPU and GPU, an FPGA does not have fixed pipeline or instruction set, but instead can be programed to act as any kind of digital logic circuit. FPGA is mostly composed of LABs (Logic Array Blocks) arranged in arrays connected by programmable routing structures. Each LAB contains a number of Logic Elements (LEs) which are the most important building block of an FPGA. Logic element consists of a Lookup Table (LUT), a D Flip-Flop or register, and sometimes additional circuits such as carry logic for increased functionality or flexibility. The LUT is made up of a tree of multiplexers with array of memory elements as input. Dependent of what data was written to the memory element during configuration, a logic element could perform any kind of desired combinational logic functions. On the other hand the register or D Flip-Flop allows the logic element to

perform sequential logic functions. The lookup tables, interconnects, and any other programmable functions in FPGA are controlled by control bits made up of SRAM cells. Before an FPGA could be used, the FPGA must first be configured, which means data must be written to the SRAM cells to set the functionality of the FPGA. Since SRAMs are rewritable, FPGAs can be reprogrammed to adapt to different kinds of applications.

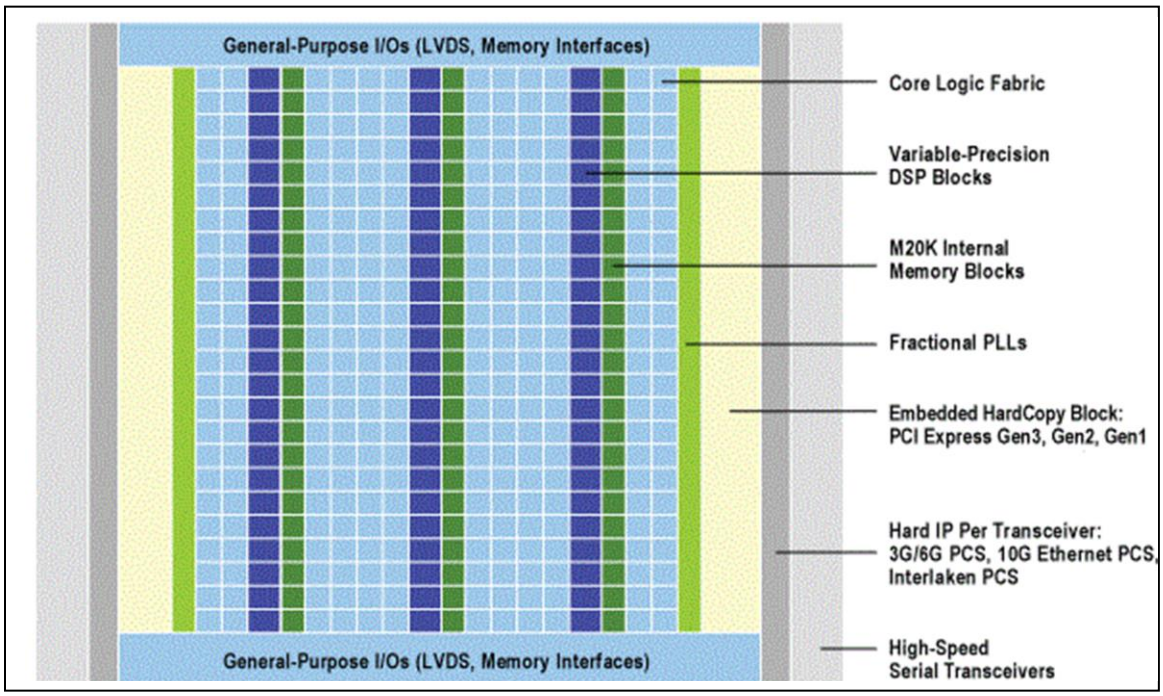

Figure 4. Stratix V FPGA Architecture [13]

Modern FPGAs usually have more complex logic cells with multiple LUTs, and dedicated hard logic such as blocked memory, DSP blocks or even embedded processors for more efficient logic utilization and higher performance. The basic layout of the Altera Stratix V FPGA use in this research is shown in Figure 4. The type of logic fabric used in Stratix V FPGAs is called adaptive logic modules (ALM), with contains 8 input fractural LUTs, and multiple embedded adders and registers. The block diagram of ALM and its LUT layout is shown in Figure 5.

FPGAs are usually programmable by using Hardware Description Languages (HDL). A synthesis tool is required to compile the design described by HDL into hardware binary called image, which can be used in configuration to write to the SRAM blocks.

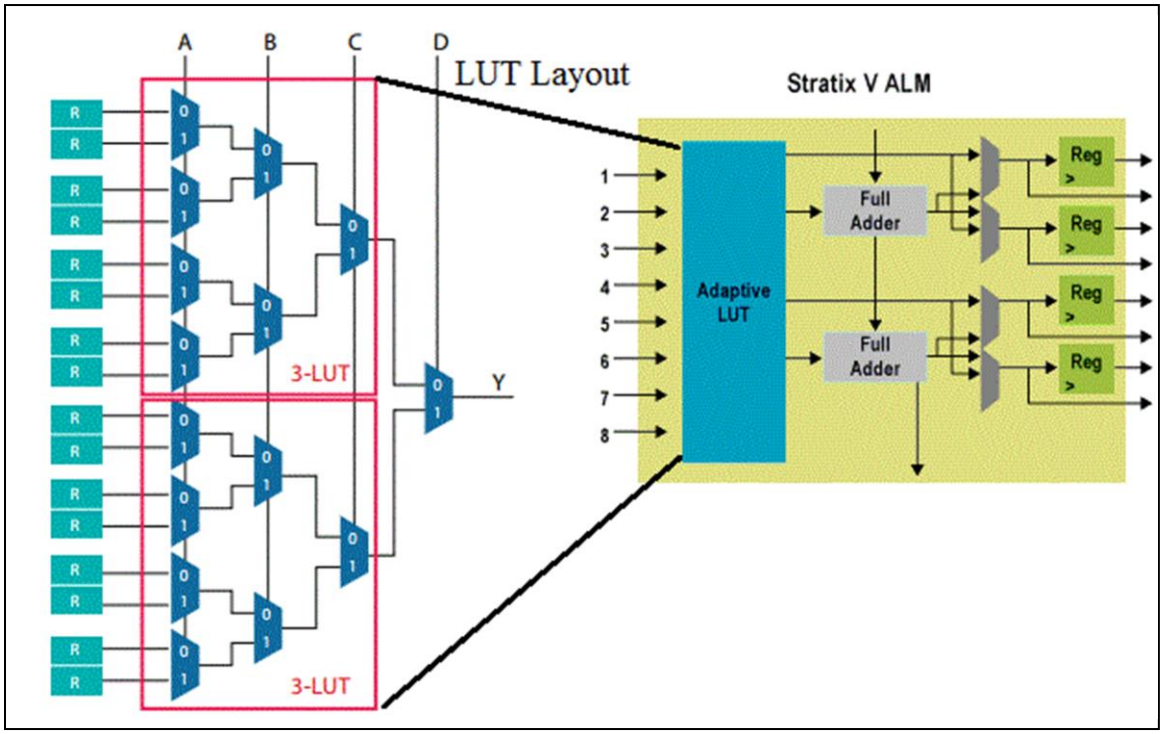

Figure 5. Stratix V FPGA ALM Layout [14]

2.2.2 FPGA Accelerators

Altera SDK for OpenCL currently supports Stratix V, Cyclone V and Arria 10 FPGAs. FPGAs by themselves cannot directly interface with the host. The FPGA accelerators comes in the form of a PCIe card, which include one or more FPGAs, along with main memory, various types of other memories, high speed data channels, and configuration circuitry. Use of PCIe interface allowed easy addition of FPGA accelerators into existing host systems. Off the shelf FPGA accelerator cards are available from companies such as Nalltech, Terasic and Bittware. Developers could also

modify the reference board design and create their own accelerators [15]. Various types of FPGAs that supports Altera SDK for OpenCL contain different amount of reconfigurable hardware in different configurations. They are suitable for different applications, and are marketed at different price range. For example, the Stratix V FPGA contains much more reconfigurable hardware on chip, but is also quite expensive; whereas the Cyclone V FPGA has less reconfigurable hardware on chip, but is also cheaper. Cyclone V also contains an ARM processor that could act as host. Each accelerator board comes with board support package (BSP) software that has to be installed into Altera OpenCL SDK. The board support package contains the necessary drivers, libraries and utilities for the Altera OpenCL SDK to interface with device. A list of available development boards can be found on Altera Cooperation website [16].

The main FPGA accelerator card used in this research is DE5-Net made by Terasic Inc. It contains a single Stratix V A7 FPGA, along with 4GB of DDR3 SDRAM as main memory. StratixV A7 FPGA [17] includes 622,000 Logic Elements (LEs), 939,000 registers, and 256 DSP blocks. The DSP blocks could be used to perform high speed variable precision multiplications, additions and other fixed or floating point operations. It also includes 50 Mbits of M20K memories, and 7.16 Mbits of memory logic array blocks (MLAB). Those memories are located very close to the logic fabric, thus could offer very high throughput if used as local memory for OpenCL kernels. In addition, the FPGA includes hard PCIe Gen 3 IP blocks and 14.1-Gbps transceivers for high speed host to device and device to peripheral communication. The layout of DE5-Net Accelerator board is shown in Figure 6.

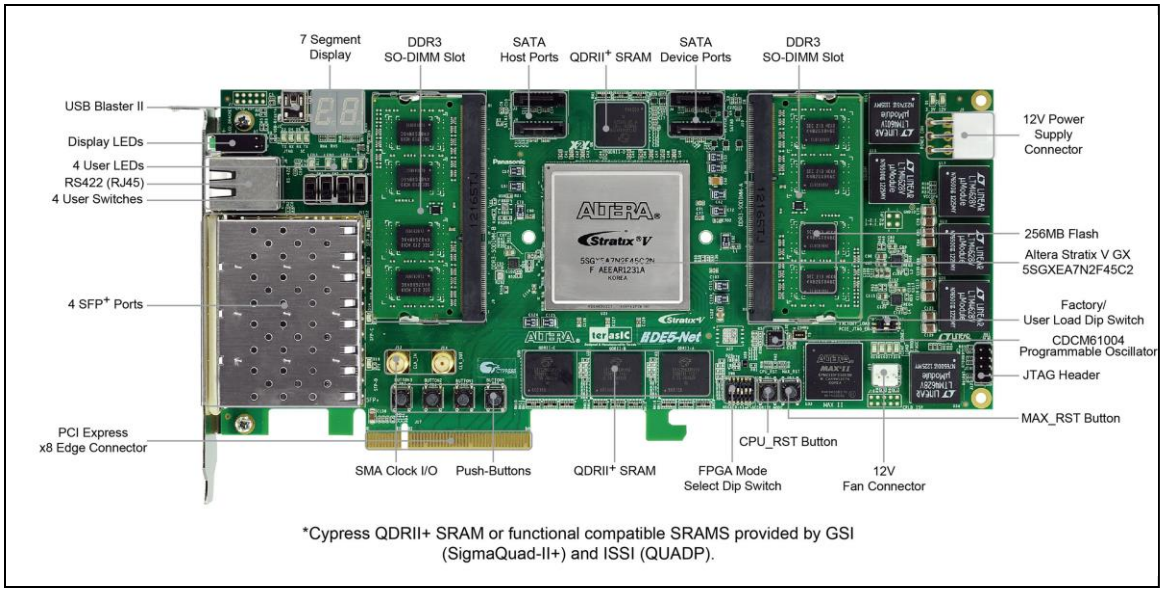

Figure 6. DE5-Net Accelerator Board Layout [18]

Nallatech 385-A7 [19] accelerator was also available for this research. The 385-A7 contains identical Stratix V FPGA, but with 8GB of DDR3 RAM, and consumes less power. The performance of the two boards is similar; however, the 385 board seems to use slightly more reconfigurable hardware to implement the memory controller, thus slight less hardware is available for kernels.

2.3 High Level Synthesis

Due to the fact that FPGAs does not have fixed pipeline and can be configured based on requirement of specific problems, in many applications they could potentially generate orders of magnitude increases in performance when programmed properly. However, traditionally applications on FPGAs were developed using hardware description languages such as VHDL or Verilog, which requires developers to have in depth hardware knowledge. Long development time and tedious debugging process made developing on FPGA much more costly comparing to developing software for CPU or

GPU. This greatly limited the applications of FPGA. High Level Synthesis (HLS) tools could solve this shortcoming by automatically synthesizing codes that are written in high level programming language such as C or C++ directly into hardware descriptions. HLS makes FPGA more favorable to developers and extended its range of applications to areas that were previously unthinkable for FPGA-based acceleration.

There are several different types of HLS tools. One type is for synthesizing C code directly to RTL-level design based on user specified constraints for generic applications. They are mostly used in speeding ASIC and FPGA design process, not accelerating a specific algorithm or application. Catapult C developed by Mentor Graphics is an example of such tool. Another type of HLS involves utilization of soft core or hard core processor to compute sequential or resource management part of the program; whereas the parallel part of the program is synthesized into RTL design. An example for this type of HLS tool is LegUp [20] developed by the University of Toronto. LegUp compiles programs into a binary that runs on soft core MIPS processor implemented in FPGA, and a set of accelerator kernels that also runs on FPGA. The resultant soft core processor, accelerator kernels and interconnects expressed in Verilog are compiled into FPGA binary. During runtime the MIPS processor performs the computation with the help of accelerator kernels. There also exist special languages that are specifically designed for HLS but they are less common. Recently introduced Altera SDK for OpenCL is a relatively new type of HLS tool that uses the same explicit parallel programming language commonly used by GPUs and CPUs.

2.4 Altera SDK for OpenCL

2.4.1 Overview

The Altera Software Development Kit (SDK) for OpenCL (AOCL) [21] was developed to lower the difficulty, time and cost to develop parallel computing programs on FPGAs. It is a high level synthesis tool that takes code written in the OpenCL language and converts it into description of the accelerator hardware written in Verilog. The AOCL is designed to be integrated with Altera Quartus design software, which can compile the Verilog code to FPGA hardware image. The Altera Offline Compiler (AOC) automatically synthesizes dedicated custom hardware for each OpenCL kernel, and takes care of the overhead of interfacing the FPGA with the host programs. This lets developers to focus on designing the parallel programs, instead of having to come up with the hardware design for their applications.

The AOCL complies with OpenCL 1.0 standard and supports many of the features in newer versions of OpenCL [22]. It includes an offline compiler for compiling OpenCL kernel source code to Verilog hardware descriptions and generating Quartus compilation scripts. In addition, the SDK also include reference board designs that allow board vendors to develop customized FPGA accelerator boards. To streamline the software development process, AOCL includes an emulator and a profiler. The emulator can execute a kernel on x86 processor to check for correctness, whereas the profiler helps the developer to analyze the performance of the program. Altera Runtime Environment (RTE) is also provided starting from version 14 of AOCL, which allows end user to build host program and execute precompiled OpenCL kernels without the Altera SDK for OpenCL.

Similar to other OpenCL platforms, a typical AOCL application includes two parts: the host code and the kernel code. The host source code is compiled into executable using GCC or Visual Studio. The kernel source code must be compiled by the Altera Offline Compiler (AOC). The compilation time for the OpenCL kernels is in the order of hours. Therefore, it must be compiled offline before the execution of the host program. The compilation flow of the Altera OpenCL follows the Altera OpenCL to FPGA Framework as shown in Figure 7.

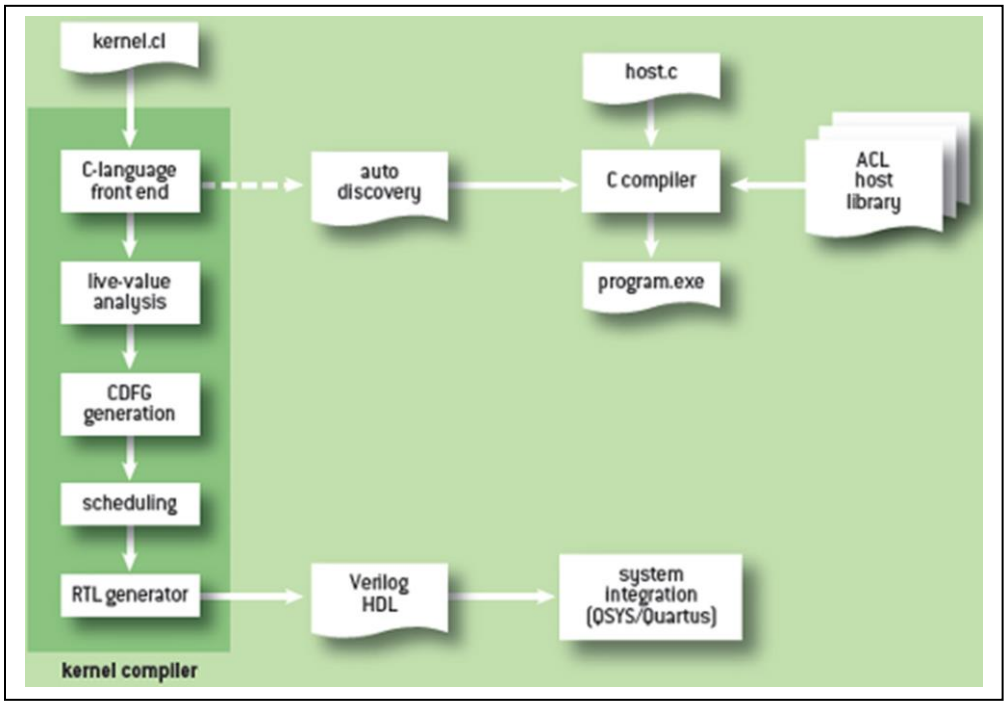

Figure 7. Altera OpenCL to FPGA Framework [23]

Inside the AOCL compiler, the kernel code first pass through C language front end and LLVM compiler infrastructure to generate intermediate representation (LLVM IR). The LLVM IR is then optimized and converted to Control-Data Flow Graph (CDFG). The CDFG is optimized further and processed by a RTL generator to generate Verilog hardware description for the kernel along with interface to host and off chip memories

[23]. In the end Quartus software compiles the hardware description into a binary file that can be used to configure the FPGA at runtime. An example of the hardware architecture synthesized by Altera SDK for OpenCL for FPGA accelerator is shown in Figure 8.

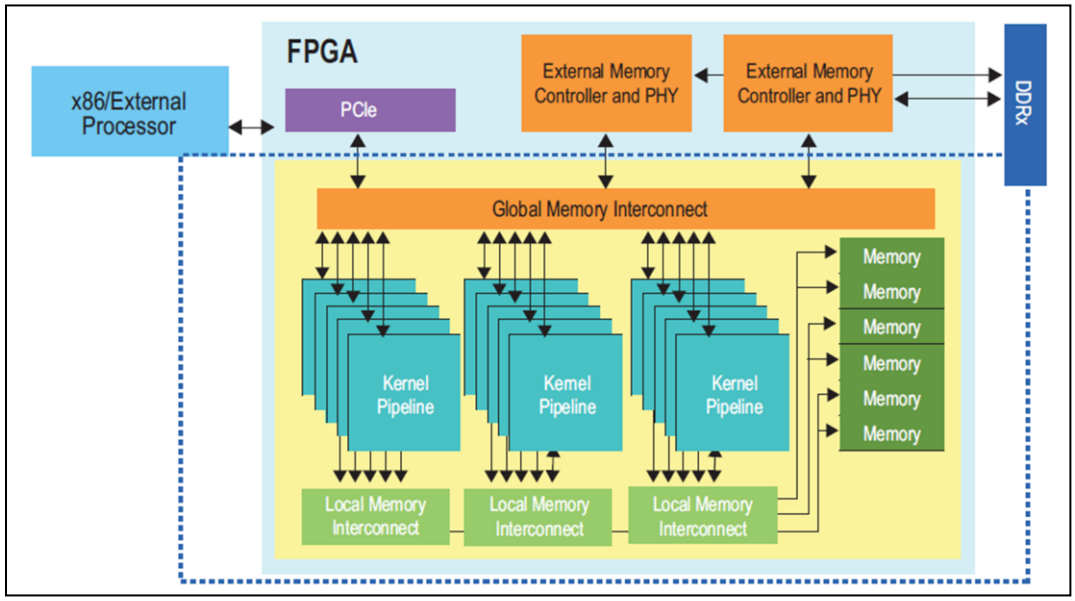

Figure 8. Example Hardware Architecture Synthesized by AOCL [25]

AOCL is designed specifically to accelerate various computationally intensive tasks, unlike most C to Gate High Level Synthesis tools, which are designed for speeding up FPGA development for generic applications. It allows programmer to target heterogeneous platforms, and utilize FPGAs alongside GPUs and CPUs. This also means OpenCL programs already written for other computation platforms such as GPU or CPU could be ported to FPGA. However, in most cases the programs have to be modified or rewritten due to architectural differences. Compare to OpenCL for GPU, Altera SDK for OpenCL generates custom hardware pipeline for kernels, which is more flexible then the GPUs that have fixed hardware. The FPGAs supported by the SDK also contain more on chip memory than current GPUs, which means more data can fit into the high speed local

cache memory and registers. Thus, AOCL can offer higher performance and energy efficiency for algorithms that can take advantage of more flexible pipelines and memory architectures.

2.4.3 AOCL Specific Features

Altera SDK for OpenCL supports many unique features to help utilizing the full potential of FPGAs [26]. Also, due to the architectural difference between FPGA and GPU/CPU, parts of the OpenCL standard are implemented differently in AOCL. One of the major differences between AOCL and other OpenCL platforms is that the kernels must be compiled offline. When building the kernel program, the targeted FPGA device is configured by the binary file. This process may take seconds. If the OpenCL program contains multiple kernels, it may be beneficial to put all kernel source code into a single source file and compiles into a single binary image. That way the overhead of reconfiguring the FPGA for different kernels could be minimized, at the same time it allows all the kernels to execute concurrently and communicate with each other during runtime.

To take the advantage of the flexibility of FPGAs architecture, Altera Offline Compiler (AOC) generates customized pipelines tailored to fit specific kernel program. As a result, it could extract parallelism from both multi-work-items (NDRange) and single work-item (Task) kernels by using pipelining. GPUs however, could not execute single threaded task kernels efficiently due to their architecture. Operations inside a NDRange kernel could be implemented as stages of pipeline, where each stage of the pipeline operates on a different work-item at the same time. Similarly in task kernels that contain loops, each stage of the pipeline processes a single iteration of the loop in parallel.

Ideally when the pipeline is filled, it could execute one work-item or loop iteration every clock cycle.

Task kernels is usually easier to code due to its resemblance to sequential programs, and at the same time they may cost less FPGA resources than multiple threaded kernels due to lack of need for synchronization barrier. However, task parallelism may not deliver good performance when the kernel contains a lots of data dependent operations. In this case the pipeline may be stalled due to data dependencies. Altera SDK for OpenCL provides shift register inference feature that could relax some of the data dependencies. The shift register inference is especially useful in applications such as performing reduction operation on an array, or performing convolutions. In those applications variables need to be constantly updated or read by different *for* loop iterations. To utilize shift register inference in reduction sum for example, a shift register array needs to be declared to hold the intermediate results from different iterations. During each iteration of the loop, shift register shifts right, summation is performed on the last element and the result is stored in the last element of the shift register. After all the input data are used, a final reduction operation is performed on the shift register to get the total sum. This process is shown in Figure 9.

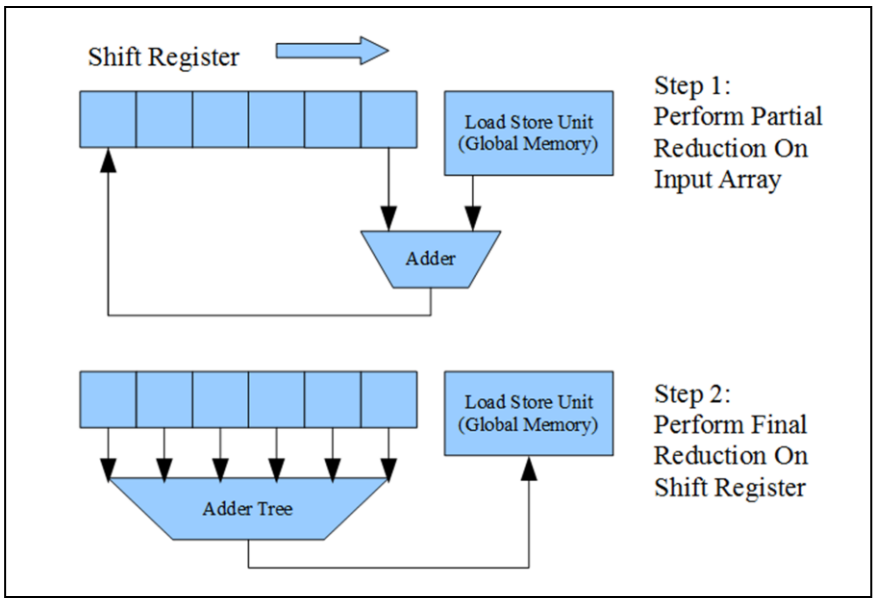

Figure 9. AOCL Shift Register Inference

In many applications task parallelism may not be effective in realizing the full potential of FPGA. In that case the multi-threaded NDRange kernels that are often seen in CPU and GPU OpenCL programs are used. Similar to OpenCL implementation for general purpose processors, in AOCL kernels vectorization could be applied to increase throughput. The simplest way to vectorize is to utilize OpenCL vector data types such as float4 or int8, although Altera OpenCL also supports SIMD style vectorization and replication of Compute Units. Setting the attribute for number of SIMD work-items for a kernel will allow AOC to replicate its datapath, and the resultant kernel will be able to process multiple work-items in parallel. On the other hand, modifying the number of Compute Units will allow the kernel to execute multiple work-groups concurrently. Increasing the number of SIMD work-items is usually more efficient than increasing the number of Compute Units, because SIMD vectorization generates less load store units for global memory and the memory accesses are coalesced.

To take the advantage of the flexibility of FPGA architecture, AOCL supports compiler flag enabled floating point optimizations. In HDL design of floating point units, normalization and rounding usually takes a lot of FPGA area. When multiple floating point operations are performed in succession, *--fpc* compiler flag could be enabled to allow AOC to eliminate the normalization and rounding in between the floating point units, thus saving FPGA space and reduce latency. AOC could also reorder the floating point operations to balance the operations and reduce number of stages in the pipeline when *--fp-relaxed* flag is enabled.

AOCL also added a unique feature called channel extensions, which allows the direct data transfer between different kernel without use of global memory or host program. The channels are implemented using first in first out (FIFO) buffers inside FPGA chip. Thus low latency high bandwidth memory transfer could be achieved through use of channel. Due to the fact that Stratix V FPGAs has rather limited global memory bandwidth comparing to GPUs, the use of channel extension could be essential for AOCL applications that require large amount of global memory data transfer to achieving high performance. However, kernel with channels could not utilize SIMD or multi-Compute Unit vectorization. This tradeoff needs to be considered when developing kernels using AOCL.

2.5 Detailed Analysis of AOCL

2.5.1 Cost of Floating Point and Integer Operations

In order to study the latency and hardware utilization for different types of floating point and fixed point operations, various vector operation kernels was compiled

and the testing results was generated as shown in Table 1 and 2. The resource utilizations of various types of operations are obtained by compiling identical NDRange vector operation kernels and reading the resultant *.area* resource estimate file. The logic element (LE), Register (Reg), Block RAM, and DSP counts are calculated by subtracting resource utilized by load store unit (LSU) from the total kernel estimated utilization. Whereas the latency is estimated by compiling task based kernels designed to repeatedly perform operations in a data dependent loop, and reading the optimization report returned by the compiler.

Table 1. Cost of Floating Point Operations in AOCL

	Precision	LE	Registers	RAMs	DSP	Latency
add	single	2380	3501	3	0	7
	double	2732	3024	3	0	9
mul	single	1929	2942	0	1	3
	double	2063	1928	1	4	6
div	single	2227	3435	8	5	14
	double	3031	4825	13	12	45
sqrt	single	2113	3148	6	2	11
	double	2497	4602	11	10	31
rsqrt	single	2108	3135	6	2	11
	double	2553	4776	11	9	23
exp	single	2560	3299	7	9	16
	double	6359	5147	11	22	30
log	single	2523	4042	6	3	21
	double	4054	6908	23	14	38
log10	single	2564	4162	6	4	25
	double	4011	6871	23	11	38
cos	single	4026	4896	6	7	35
	double	5725	9515	12	30	45
sin	single	4089	5885	6	7	36
	double	5791	9923	12	30	46
tan	single	4637	7122	12	13	56
	double	11331	15385	30	74	100
min	single	1906	3043	0	0	3
	double	2407	3834	13	0	1

Table 2. Cost of Fixed Point Operations in AOCL

	Precision	LE	Registers	RAMs	DSP	Latency
add	char	1751	2533	0	0	1
	short	1779	2548	0	0	1
	int	1835	2587	0	0	1
div	char	2884	4541	18	4	32
	short	2900	4548	18	4	32
	int	2932	4571	18	4	32
min	char	1751	2533	0	0	1
	short	1772	2616	0	0	1
	int	1803	2652	0	0	1
mul	char	1749	2605	0	1	2
	short	1773	2628	0	1	2
	int	1822	2715	0	2	3

Note that the latency is measured in clock cycles. From the tables we can see that double precision floating point operations cost a lot more than single precision operations both in terms of FPGA area used and latency. Operations such as finding minimum and multiplication are the least costly, whereas division, square root, logarithm, and trigonometry operations cost the most FPGA area and time. Note that Stratix V A7 FPGA used in this research only has a total of 256 DSP units. High cost functions such as double precision tangent should be avoided if possible.

2.5.2 Kernel Launch and Transfer Overhead

Other performance evaluations done on AOCL are summarized below. One of the performance metrics that we are interested in is the speed of data transfer between kernel and host. According to [27] the bandwidth of host to device data transfer for the GPU is about 2.82 GB/s, with latency of 50~60 us whereas the bandwidth of device to host data transfer is about 3.29 GB/s with latency of 140~150 us. The AOCL memory diagnostic program gave comparable result of about 1.75 GB/s write to device and 2.92 GB/s read

from device for Terasic DE5-net accelerator. The Nallatech 385 accelerator has slightly higher global memory throughput of 2.46 GB/s host to device and 2.95 GB/s device to host. There is no easy way to determine the latency, but FPGA should have comparable latency to GPU. The throughput for data transfer between kernel and host is far lower than the 25.6GB/s peak bandwidth of global memory, which means communication to host should be minimized. The peak bandwidth of private and local memory is dependent the kernel because FPGA does not have a fixed architecture, thus could not be accurately determined.

Another important performance metric is the overhead of launching a kernel. The test methodology used is to launch an empty kernel repeatedly, both with and without synchronization between each kernel launch. For time keeping, the submitted and execution time returned from OpenCL build-in profiler function as well as the wall clock time recorded by the OS timer are both recorded and compared. The result is that when launching a single kernel, the queued to submitted time is 0.004 ms and submit to start time is 0.016 ms. When launching the kernel repeatedly for a large number of times and synchronize after every kernel launch, although submit to start time increases linearly with respect to the number kernel launches, the wall clock time increases very rapidly (20 s for 10000 launches). When the *clFinish* function used to synchronize the kernel launches is replaced with *clFlush* function that issues the kernel launch command without waiting for operations to finish, the wall clock time is reduced to more acceptable 7 s for 10000 launches. When launching kernels without synchronization, the wall clock time reduces further to 4.7 s for 10000 launches. Therefore, synchronizations during kernel

launch should be minimized to reduce overhead. At the same time it is often more effective to use multiple workgroups kernel, than launch the kernel multiple times.

2.5.1 Effective Reduction

Reduction is one of the common patterns in parallel computing. It reduces an array of data into a single output by repeatedly performing some type of reduction operation. The reduction operation could be summation, product, or finding min/max. The computational complexity of reduction is O (N), and since the number of operations performed is equal to data size, reduction speed is bounded by global memory bandwidth. For the accelerator that we have, the theoretical maximum single precision reduction throughput is 6.4 GFLOPS. This is calculated from dividing 25.6 GB/s maximum bandwidth by 4 byte per floating point value.

Altera recommends [28] performing reduction by using a single threaded kernel. If a simple *for* loop is used to perform reduction, one iteration could only start after the pervious iteration is completed. This is due to memory dependency on the partial result. Since most operations take multiple clock cycles to complete, the performance will suffer greatly as a result. At the same time loop unrolling could not be effectively applied to increase the throughput. Without any optimization the loop version of reduction could only achieve 0.035 GFLOPS throughput. To relax data dependencies, Altera recommends replicating the partial sum storage register and implementing a shift register to perform reduction. The parallelism is extracted by unrolling the loop to ensure multiple reduction operations are done concurrently. Test shows that the throughput for this method is only around 0.25 GFLOP, or 1 GB/s equivalently. This is better than un-

optimized version, but still far from maximum throughput because it could not saturate the global memory bandwidth of the FPGA.

Inspired by multi-thread solutions introduced by GPU vendors, various test kernels were developed. The most efficient way to implement reduction is determined to be using two kernels. In this method, the input data is partitioned into equally sized blocks. An NDRange kernel first processes different blocks of the input array simultaneously, then a second task kernel reduces the partial sum into a single value. The second kernel is implemented the same way as Altera programming guide recommended, but it will only perform a small portion of computation, whereas the vast majority of the calculation is done by the first kernel. The two kernels are connected via channel to avoid wasting global memory bandwidth. There are a few different ways the first kernel could be implemented. The simplest way is by launching single work-item work-groups, each work-item loops through one block of data and computes the partial sum. The loop could be unrolled to increase throughput effectively. Since in NDRange kernel each pipeline stage processes a different work-item instead of loop iteration, there is no data dependency. The block diagram for this implementation is shown in Figure 10. The optimal block size is dependent on input data size. The first NDRange kernel is only efficient when data block is large enough to fill the pipeline, and the number of blocks has to be small enough so that the less efficient task kernel does not take too long to finish. For example, when performing sum reduction on 1GB data, the best block size is 32, which will produce 3.70GFLOPS throughput on Terasic DE5-net accelerator or 4.03GFLOPS throughput on Nallatech 385 accelerator. Those throughput numbers indicates that 58 to 63% of the theoretical global memory bandwidth has been reached,

which is satisfactory. The full source code for reduction kernel with addition operations can be found in Appendix A.

In an attempt to improve memory access efficiency, another NDRange kernel with multi-threaded work-groups was also developed. The kernels use multiple work-items to reduce each block instead of single work-item. In order to ensure coalesce memory access, consecutive input data is accessed by successive work-items. This version of the parallel reduction did not outperform the simple implementation during test and thus was discarded.

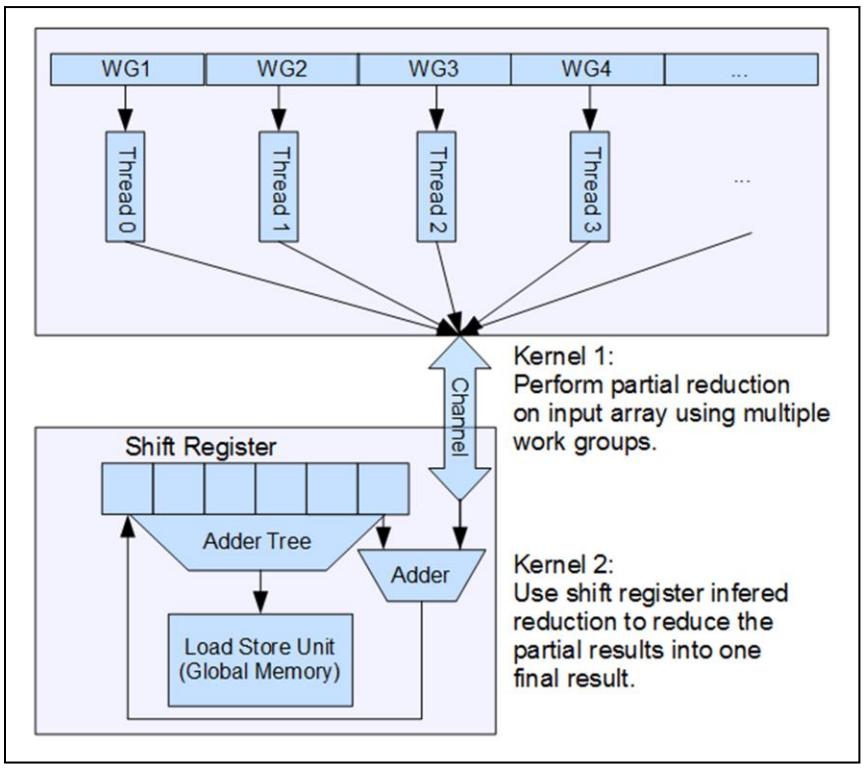

Figure 10. Optimized Two Kernel Reduction Block Diagram

Very little hardware was needed to generate enough performance to saturate the relatively small FPGA global memory bandwidth. On DE2-net accelerator the fully optimized reduction kernel for summation used 29% logic, 8% Block RAMs and 2 DSP

blocks, and runs at 236.85 MHz. On Nallatech 385 accelerator, similar amount of FPGA resources were used, but the kernel is clocked at slightly higher 260.89 MHz. The knowledge gained from developing the reduction kernel here was also used in designing the k-means kernels in chapter 4.

2.6 Brief Summary of Algorithms used in Acceleration

The algorithms explored in this thesis include k-means clustering, k-nearest neighbor, N-body algorithms, and LU matrix decomposition. This thesis devotes one chapter to each of the listed algorithms.

K-means clustering algorithm is one of the most popular data mining algorithms used in image processing and machine learning. It is very time-consuming for large data and cluster sizes. In this research, an optimized implementation of k-means clustering algorithm on FPGA was developed using Altera SDK for OpenCL. Performance and power consumption of FPGA implementation are measured and compared against CPU and GPU implementations.

K-nearest neighbor (kNN) is another popular machine learning algorithm that classifies the query points by compares their distance between training points. The classification of a query point is determined by the classes of *k* training points closest to the query point. It is commonly used in machine learning and data mining applications. This research focused on implementation of brute force k-nearest neighbor algorithm using AOCL and the results are compared with best published works.

N-body simulation simulates dynamic interaction of particles. It is often used in the field of astrophysics and chemistry. The N-body simulation algorithm implemented in

this research is pair-wise method with time complexity of O (N^2) for each iteration. The results are compared with optimized CPU and GPU implementation.

LU decomposition factorizes a matrix into the product of a lower triangular matrix and an upper triangular matrix, hence the name. This method is useful in solving linear systems of equations and finding inverse, and has been implemented in many computing libraries such as LAPACK, cuBLAS, MKL etc. This research tries to determine if the blocked LU decomposition algorithm could be implemented on FPGA using AOCL to achieve performance comparable with existing optimized CPU and GPU implementations provided in numerical libraries.

For consistency, all OpenCL kernels developed and tested on FPGA in the following chapters were compiled using Altera SDK for OpenCL version 15.0.

Chapter 3

Acceleration of K-Means Clustering Algorithm

3.1 Introduction to K-Means Clustering Algorithm

3.1.1 Introduction

K-means clustering algorithm involves partitioning of data iteratively into *k* clusters. It is among the most popular data mining algorithms [29], and is used in many other applications such as image processing and machine learning. However, k-means is highly time-consuming when data or cluster size is large. The k-means clustering algorithm operates on a set of *d* dimensional data set $X = \{x_1, x_2, \ldots, x_n\}$ to partition them into *k* clusters, where *n* is the total number of data points. The end result is a set of *d* dimensional centroids for the clusters $C = \{c_1, c_2, \ldots, c_k\}$, along with a membership set $M = \{ m_1, m_2, \ldots, m_n\}$ that records which cluster each data point is the closest to. A set of initial clusters centroids must also be supplied. There are many ways to determine initial clusters. Depending on which method is used, the resultant centroid and convergence speed could be vastly different. The most common way is to randomly choose *k* data points as initial clusters. A more optimized way of selecting initial clusters called k-means++ was proposed [30] which allows faster convergence. However, for simplicity and consistency the implementation used in this thesis chooses the first *k* data points as initial cluster.

3.1.2 Sequential Algorithm

In each iteration of the k-means algorithm, the distance between data points and centroids are compared. Each data point is then assigned to the closest cluster. There are

a few different way of measuring distance as well. The squared Euclidean distance is most commonly used in k-means, which is simply the sum of squares of the difference between data point and cluster center in each dimension. Since we are only comparing the distances, the square root is omitted to save computing time.

$$dist(x, c) = \sum_{i=1}^{d} (x_i - c_i)^2$$

Manhattan distance measures the distance between cluster center and data points as the sum of absolute value of the difference between data point and cluster center in each dimension.

$$dist(x, c) = \sum_{i=1}^{d} ||x_i - c_i||$$

When all the data points are processed, new cluster centroids are obtained from average of data points belong to the same cluster. Assuming the number of objects in cluster *i* is defined as s_i, the formula for the cluster update step is shown below:

$$c_i = \frac{1}{s_i} * \sum_{j=1}^{s} x_j \, \mathrm{a_i} = \mathrm{G}^* \sum_{\mathrm{j}=1}^{\mathrm{N}} \left(\frac{\mathrm{m_j r_{ij}}}{(||\mathrm{r_{ij}}||^2 + \varepsilon^2)^{3/2}} \right)$$

This process is repeated until a predefined maximum number of iterations is reached or the number of changes in data point membership drop below a certain threshold. The pseudo code of the algorithm is shown below.

ALGORITHM 1. Sequential K-means Algorithm

```
input: initial clusters, objects, problem dimensions N, D, K
output: cluster centroids and membership (index)

load objects
initialize clusters
while delta < threshold do
    set clusters_new[K][D] array to 0;
    set clusters_size[K] array to 0;
    for each object n do
```

```
        for each cluster k do
            for each dimension d do
                dist ← dist + (objects[i][d] – clusters[k][d])^2;
            end
            if dist < min_dist then
                min_dist ← dist;
                index ← k;
            end
        end
        update delta and membership;
        for each dimension d do
            clusters_new[index][d] ← clusters_new[index][d] + objects[i][d];
        end
        clusters_size[index] ← clusters_size[index] + 1;
    end
    for each cluster k do
        for each dimension d do
            clusters[k][d] ← clusters_new[k][d] / clusters_size[k];
        end
    end
end while
```

The computational complexity for each iteration of k-means algorithm is O (d*n*k + n*k + n*d). The distance calculation step is the most computationally intensive part of the algorithm and the total number of operations is roughly equal to iterations*d*n*k*3, because it takes one addition, one subtraction and one multiplication/absolute value to calculate distance partial sum for each data element.

3.2 Related Works

Various works had been done on acceleration of k-means algorithm on CPU, GPU and FPGA. MineBench Benchmark Suit [31] was published in 2006, which included an OpenMP / OpenMPI multi-threaded CPU implementation of parallel k-means. The OpenMP version of the k-means benchmark code is used in this research for speed comparison with CPU.

Che et al. from University of Virginia presented a CUDA implementation of k-means algorithm [32], which achieved up to 35x speedup on GTX 260 GPU compare to

an OpenMP implementation running on CPU. The reduction process was done by CPU, but major part of cluster update was done on GPU. The data was stored in texture memory and the cluster was stored in constant memory, which limited the maximum problem size the GPU could compute without hitting the memory bandwidth barrier. Later in the same year researchers from Hong Kong University of Science and Technology presented GPUMiner [33] parallel data mining system. This implementation used bitmap technique optimized for multi-threaded SIMD GPU architecture. Compared to Che et al.'s work, the GPUMiner spent more time on computing instead of data transfer, and achieved up to 5x speedup.

Wu et al. from the HP Labs reported another CUDA implementation of k-means algorithm [34] designed to process large data sets, including those cannot fit into memory of GPU. With 2 dimensional data points and 1000 clusters, they achieved more than 11 times speedup over CPU on a GTX280 GPU. The centroid update portion of the computation is done on CPU in this implementation.

Li et al. presented another GPU implementation of the k-means algorithm [35] in 2010. Two different implementations were developed separately to optimize for low dimensional data sets and data with higher dimensions. For low dimensional data they utilized register to reduce memory access latency, while for high dimensional data they applied parallel programming pattern used in matrix multiplication to accelerate the k-means algorithm. Overall they were able to obtain 3 to 8 times speedup over previous best GPU-based implementations. However, the performance was heavily dependent on problem size. For example, on a GTX280 GPU while processing 8 dimensional data, they were able to achieve 676 GFLOPS throughput; but while processing 34 dimensional data,

the throughput dropped to 137 GFLOPS. In this implementation the reduction part of centroid update is done on CPU. This is the best performing k-means implementation published to date.

Dhanasekaran et al. from AMD presented a novel k-means GPU implementation [36] that used Irregular reductions and performed computation completely on GPU. All previous GPU implementations published before computed reduction on the CPU. They achieved more than 35 times speedup compared to four-core CPU for large data size and claimed to be 3.2X faster, 1.5X faster, and equally fast as CPU-GPU hybrid implementations for cluster size of 10, 100, and 400 on ATI HD 5870. However, as cluster size increases, the performance speedup decreases for this implementation, and work of Li et al. [35] was not being compared with.

Many researchers also presented various FPGA implementations of k-means algorithm. However, majority of those works were done using fixed point data type with relative small data sizes that is intended for use in image processing applications. Thus they are not directly comparable to the implementation presented in the thesis. However, there are some exceptions. For example, A FPGA implementation [37] of k-means algorithm using MapReduce was presented in 2014. The k-means computations are divided into map and reduce functions and implemented into separate FPGAs. With two Mapper FPGAs and one Reduce FPGA, it was 15.5 to 20.6 times faster when compared to Hadoop MapReduce framework baseline software implementation. The FPGA used is Xilinx Kintex-7 XC7K325T and the test data used is UCI Machine Learning Repository's individual household electric power consumption data set. Kintex-7 is comparable to Stratix V used in this research, but with different architecture. The XC7K325T

particularly has 326,080 logic cells, 16,020 Kb Block RAMs and 840 DSP Slices. The dimension was limited to 2 and 4 while cluster size is relatively small. The throughput was not as high as other GPU implementations, but the design is scalable to multiple FPGAs.

An Altera SDK for OpenCL implementation [38] was introduced in the same year. This implementation was used to demonstrate the APARAPI Java Framework, which automatically ports higher level Java code to OpenCL. It was able to achieve 6.2 ~ 7 times speedup versus a CPU implementation with 65 ~ 80 percent power reduction. However, the design is only implemented for data sets with dimension of up to 8.

3.3 Synthesis Using AOCL

3.3.1 Single Threaded Implementation

The sequential algorithm is ported to Altera SDK for OpenCL with minor modifications. All parts of the algorithm are enclosed in a single thread task based kernel. Parallelism is achieved through pipelining and unrolling the loops. It turns out that this kernel is quite slow and inefficient. When one dimensional data is used to test the kernel only maximum of 5 GFLOPS performance was achieved. When the dimension of the data is increased, the performance also slightly improved, however the memory bandwidth was far from being saturated. Various optimizations were attempted. For example, different ways of organizing data and cluster such as row major, column major, vectored type, and user defined type were tried. Different combinations of loop unrolling and optimized reduction with shift register inference were also attempted. However, they were either slower or not much faster than the original version, likely due to unresolvable memory dependencies. It became clear that task (pipelining) based parallelism could not

produce satisfactory level of performance. Therefore it is necessary to parallelize the kernel, and break major parts of the algorithm into different kernels. While the memory utilization will significantly increase, the kernels could be better optimized to suit the parallel patterns they implement.

3.3.2 Parallel Multi-Kernel Implementation

The k-means computation can be broken down into three major components: the calculation of distance, assignment of objects to clusters, and the update of new clusters. The cluster update step is effectively a histogram operation, where objects are summed into different bins/clusters and their average is taken after the summation as new cluster centroids. Unlike normal histogram, however, the data in this case is multi-dimensional. In order to compute histogram in parallel, reduction operations could be applied to sum the objects to new cluster partial sums while counting the number of objects belonging to each cluster. When that's done the partial sums could be merged and averaged with scalar division to obtain the new clusters. It is also possible to compute histogram in parallel using atomic operations that combines read, compute and write into one indivisible operation, thus avoiding memory access race conditions. However, the AOCL best practice guide [28] suggested avoiding atomic operations due to inefficiency of such operations on FPGA.

The distance calculation, cluster assignment, reduction and averaging could be done either in separate kernels, or some of them could be merged into the same kernel. For example, the cluster update could be done using two stage reductions, where a multi-threaded kernel is first used to compute partial sum of the objects while a second single-threaded kernel is used to sum the partial sums. Finally a third kernel is used to average

the result to obtain the updated cluster centroids. Each time a new kernel is added to the FPGA binary, there are overheads on resource usage and memory transfer between kernels. Thus the number of total kernels in a single design should be minimized if possible, in order to save memory bandwidth and hardware resources for actual computation rather than wasting them on kernel overhead.

The final parallel implementation of k-means algorithm uses two kernels. The first assignment kernel is a NDRanged kernel with block size equal to the number of objects in the cluster. Each thread loads and performs calculation for one object point. It calculates the distance between each pair of the objects and clusters, and saves the index of the cluster that the object is closest to. To save global memory bandwidth, each workgroup only loads the cluster once into the local memory and shares it among all the threads in workgroup. The second reduce kernel is a single thread task kernel which takes the membership information from the first kernel and uses it to sum up all the objects in a cluster. In the end it takes the partial sum and divides it by the number of objects in cluster to obtain the new cluster. This design offered best performance over other attempted variants. To further reduce memory transfers between kernels and the global memory, AOCL Channel Extension was used to directly transfer membership data between kernels. This saves global memory access and decreases hardware resources utilized by the memory load and store unit. The channel extension in this case is also used to provide synchronization between kernels without utilizing the host. This way the kernels could be executed concurrently, and total execution time for a single iteration is reduced. The object data could also be transferred to the second kernel, but only when dimension size is very small because otherwise the design will not fit on FPGA. During

testing, it turned out that using channel to transfer object itself to the reduction kernel did not offer performance improvement in general, thus it was not included in the final design. Depending on the feature and cluster size, the speedup gained by using the channel was between 5 to 50%. The depth of the channel does not noticeably affect the performance or hardware utilization. An earlier version of the kernel separated the reduction and averaging of the cluster update step into two different kernels. In this variant one kernel sums the objects to separate copies of clusters using independent threads while the other kernel merges and takes average of the resultant partial sums. However, test revealed that it is better to merge the reduction and averaging kernels because the last averaging kernel took insignificant amount of time to complete. The hardware recourses originally utilized by the third kernel were used to support larger cluster size and faster kernel clock frequency in the final version.

In addition, since the distance between cluster and object is squared during calculation, the sign of the total distance is always positive. Therefore, during the process of determining the minimum distance, the distance sum was casted to unsigned integer type before comparison, which makes it less costly then floating point comparison operation. The pseudo code for the parallel k-means cluster assignment kernel and update kernel is shown below. The block diagram of the kernels is shown in Figure 11. The full source code for the kernels can be found in Appendix B.

ALGORITHM 2. Parallel K-means Algorithm

```
define preprocessor directives;
enable channel extension;

workgroup_size ← D*K;
number_threads ← N;
kernel kmeans_assign( objects, clusters, members, problem sizes )
    load cluster and synchronize threads;
```

```
    load one object per thread;
    for each cluster k do
        for each dimension d do
            dist ← dist + (object[d] – cluster[k][d])^2;
        end
        if (unsigned)dist < (unsigned)min_dist then
            min_dist ← dist;
            index ← k;
        end
    end
    check if membership changed;
    write member and change information to channels;
end kernel

workgroup_size ← 1;
number_threads ← 1;
kernel kmeans_update( objects, members, clusters, delta, problem sizes )
    set clusters_new[K][D] ← 0;
    set clusters_size[K] ← 0;
    delta ← 0;
    for (n = 0; n < N; n++) do
        read member and change info from channels;
        write member to global memory;
        delta ← delta + change;
        clusters_size[member] ← clusters_size[member] + 1;
        for (d = 0; d < D; d++) do
             clusters_new[member][d] ← clusters_new[member][d] + objects[i][d];
        end
    end
    write delta to global memory;
    for (k = 0; k < K; k++) do
        for (d = 0; d < D; d++) do
            clusters_new[k][d] ← clusters_new[k][d] / clusters_size[k];
           write clusters_new to clusters global memory;
        end
    end
end kernel
```

In the host, the data are initialized and copied to device. The two kernels are enqueued into two separate commands queues in *for* loop, and thus are executed concurrently. After each iteration of *for* loop, the membership change counter value is copied back to the host from device in order to determine if the kernel execution should stop or not. The loop is terminated either when the maximum iteration time is reached or when the change count falls below threshold. After which the result is copied back to the host for verification.

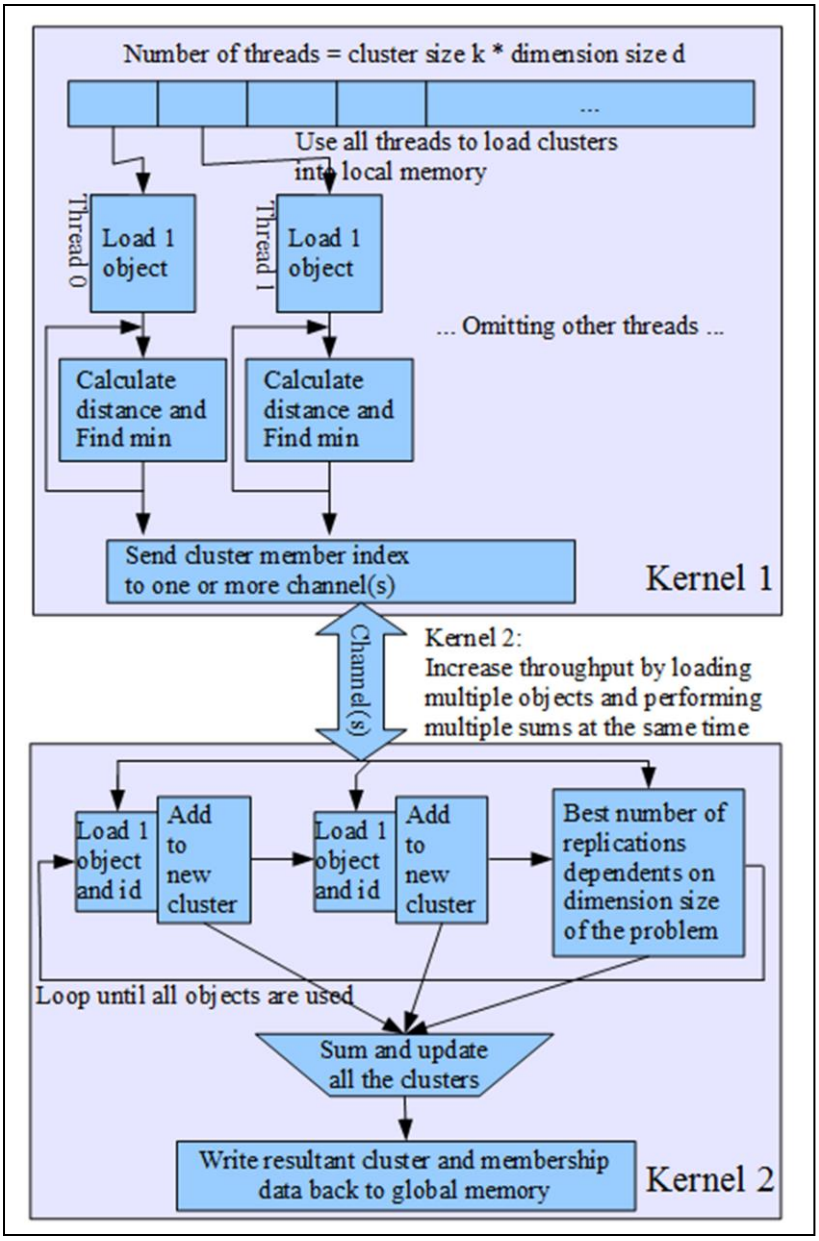

Figure 11. Block Diagram of Parallel K-means Kernels

Other parallel k-means implementations were also attempted. For example, a kernel which merges the cluster assignment and partial sum portion of the reduction computation was developed and tested. But its performance was much slower than the kernel that does the computations separately. Also, various cluster assignment kernels with two dimensional work-groups threads instead of one were developed, where the

second dimension is of the same size as the dimension of data. However, they did not offer any advantage over the kernel with one dimensional thread, due to increased hardware resource usage. They could not fit in one FPGA if same dimension or cluster size was used.

3.3.3 Optimization for Different Problem Sizes

Interestingly the parallel multi-kernel implementation discussed earlier is more suitable to data with mid to high dimensions. This is contrary to most popular CPU and GPU based parallel implementations, which favors lower dimensional data. The maximum feature dimension that could fit on Stratix V A7 FPGA with the proposed implementation is around 160. When higher dimension is used, the design will not fit even with reduced unroll size. At size 160, local memory replication needs to be turned off and the cluster size has to be reduced to 128 in order to fit the design on FPGA. As a result the performance is poor at this size. Larger sized data could be processed with a kernel which only cache part of the cluster into the local memory. However, the global memory access will increase dramatically. Due to the relatively small global memory bandwidth available for the FPGA it will not be competitive against GPU with this problem size. If the distance calculation loop and cluster partial summation loop is only partially unrolled instead of fully unrolled, it could also fit slightly larger sized problems. However, this resulted in higher memory transfer time and inefficient pipeline during test, and was much slower.

In the case of low dimension, while the assignment kernel still performed very well, the reduction kernel in our implementation could not achieve sufficient parallelism by unrolling the loop alone. Therefore it was necessary to implement a different version

of the reduction kernel so that consistent performance could be achieved across all dimension sizes. After some trial and error it turned out that memory dependencies in the reduce kernel could be negated by manually replicating some of the data dependent memory resource and reduce the replicated copies in the end of the kernel. This is better than use of *pragma unroll* directive because loop unrolling requires full unroll in order to be efficient in the case of this implementation, which wastes a lot of resources. In addition, in order to make the kernel work with different cluster sizes, the number of threads for the assignment kernel is limited to the predefined maximum cluster size that could be fitted to the FPGA rather than a constant size. It turned out that the best (> 100 GFLOPS) performance could be obtained when cluster size is sufficiently large. Significant amount of time was spent on improving performance at cluster size smaller than 32. As a result some kernels developed was able to achieve faster result at *k* smaller than 16 at the cost of lower performance at higher cluster sizes, but the performance was still not satisfactory at low *k* size. The proposed kernel is only faster than CPU at *k* size larger than or equal to 8, and is only significantly faster than CPU at *k* size larger than or equal to 32.

3.3.4 Distance Calculation

Manhattan distance and fixed point versions of the k-Means algorithm were also attempted. However they do not offer significant performance improvement. Since Manhattan distance calculation uses absolute value instead of multiplication, the multipliers in DSP units that are already built into the FPGA are under-utilized. In the case of replacing the floating point data with unsigned integer data type, each unsigned integer multiplication used in distance calculation actually needed two DSP units,

whereas floating point multiplication only used one. Therefore the number of multiplication operations that could be executed concurrently is reduced to half. Thus neither Manhattan distance nor unsigned data type offer any meaningful performance improvements for AOCL implementation of k-means.

3.3.5 Verification

In order to ensure the accuracy of the k-means implementation, a sequential version of the k-means algorithm is implemented in host program. This implementation can be chosen to run after the kernel is executed and provide reference results. After the kernels are executed, the membership information along with the resultant clusters are copied from global memory back to host to compare with the reference results. Mean squared error (MSE) is used to evaluate the difference between the reference result generated by CPU and FPGA. This sequential k-means verification code generates identical result as MineBench OpenMP implementation of k-means.

3.4 Synthesis Results

3.4.1 Performance

Multiple tests with different data, cluster, and dimension sizes were conducted. Data was randomly generated. The number of iterations required to reach the steady state threshold varies depending on the data provided. For random floating point data between -100 to +100 and threshold of less than 0.1% object membership change, the average number of iterations is about 20. This may be lower than the iterations usually required for the clusters to settle. However, because in FPGA implementation each iteration takes consistently the same amount of time, 20 iterations are sufficient to measure the peak performance. The performance is measured by both the execution time in seconds and

throughput in GFLOPS. In order to test for different feature, cluster and data sizes, automatic testing scripts were written to launch the host program with different problem sizes and kernel names. The results are shown in Figures 12 and 13. Figure 12 shows execution time for computing around 2,097,152 objects for kernels of various dimensions and different cluster sizes. Figure 13 shows the throughput achieved by kernels of various dimensions while processing problems with different cluster sizes. The data transfer time between host and device is not included because they are insignificant (less than 0.5ms) compared to kernel run time. Different colored lines represent results from kernels of different dimensions, and the horizontal axis represent different cluster sizes.

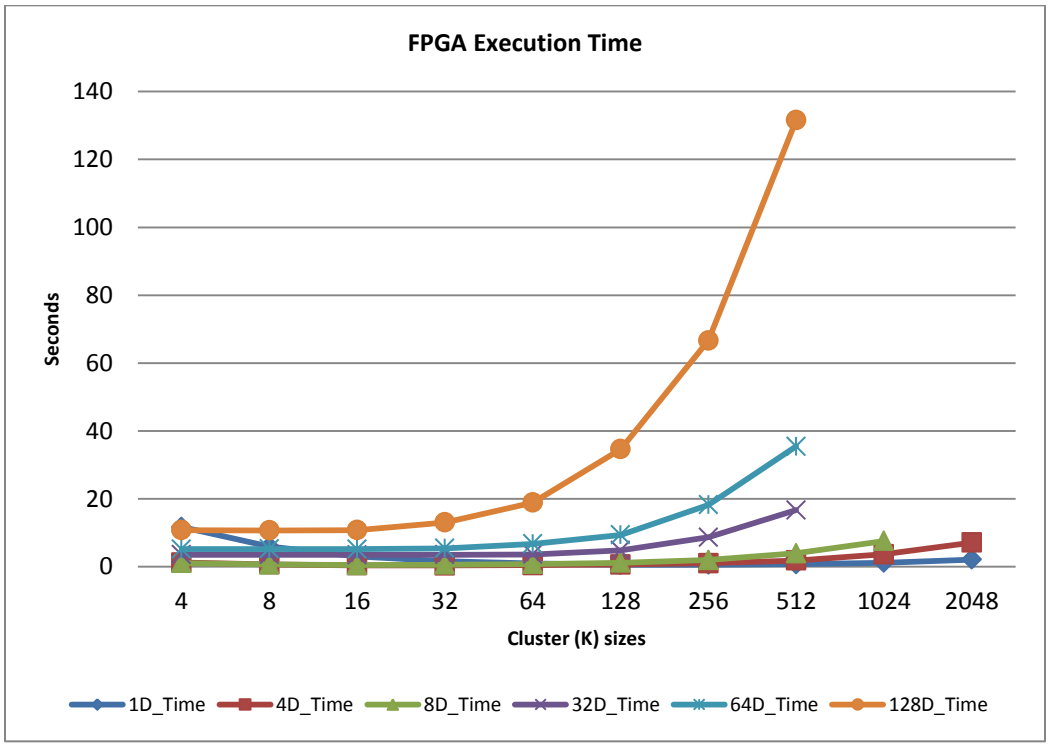

Figure 12. Execution Time for Computing 2 Million Objects on FPGA

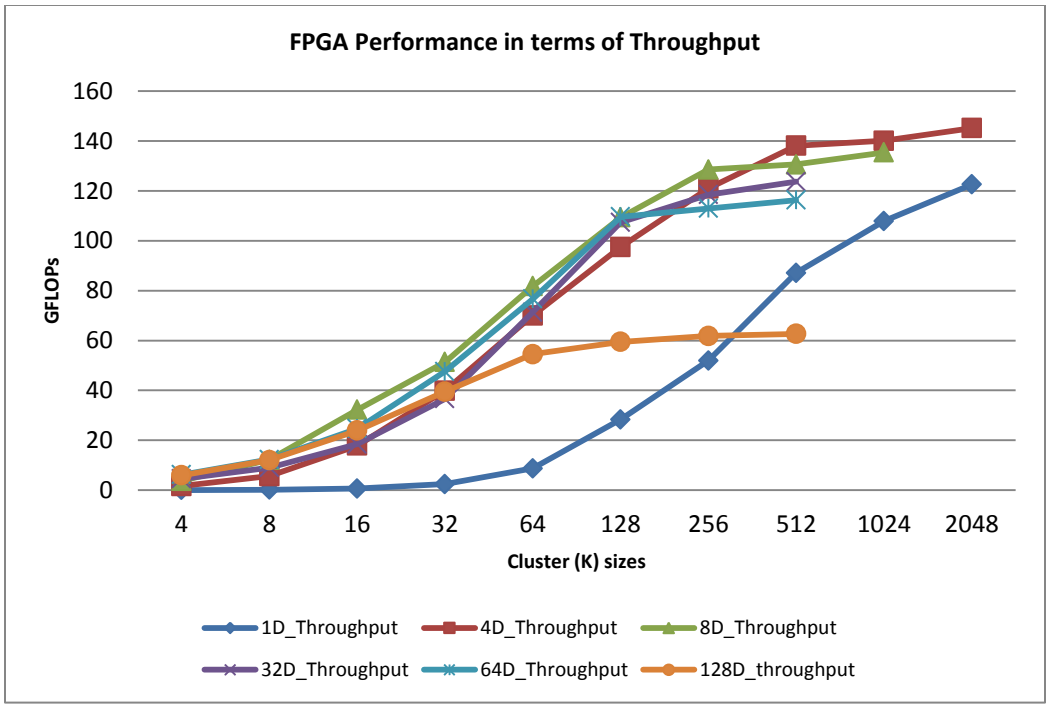

Figure 13. Peak Throughput for Computing 2 Million Objects on FPGA

The AOCL FPGA implementation of k-means performs best at medium dimension and large cluster sizes, as indicated in Figures 12 and 13. The peak performance is nearly 150 GFLOPS. For problems with 4 to 64 features, and cluster size of greater than 128, the AOCL implementation consistently obtained greater than 100 GFLOPS throughput.

The CPU implementation used is fully optimized MineBench 3.0.1 based on OpenMP. The CPU used to run MineBench comparison code is a six core Intel Xeon W3670 with 12.288 MB of cache and clocked at 3.2GHz. All available threads are utilized at 100% during the execution of the Minebench program. The data used and number of iterations are identical to the testing condition set for testing the FPGA implementation. The results are shown in Figures 14 and 15. Figure 14 shows execution time for computing around 2,097,152 objects with various dimensions and cluster sizes

on CPU, whereas Figure 15 shows the throughput achieved by CPU. These figures indicate that the CPU implementation of k-means favors large dimension and cluster sizes; however, the peak throughput achievable on the six cores Xeon CPU is less than 18 GFLOPS.

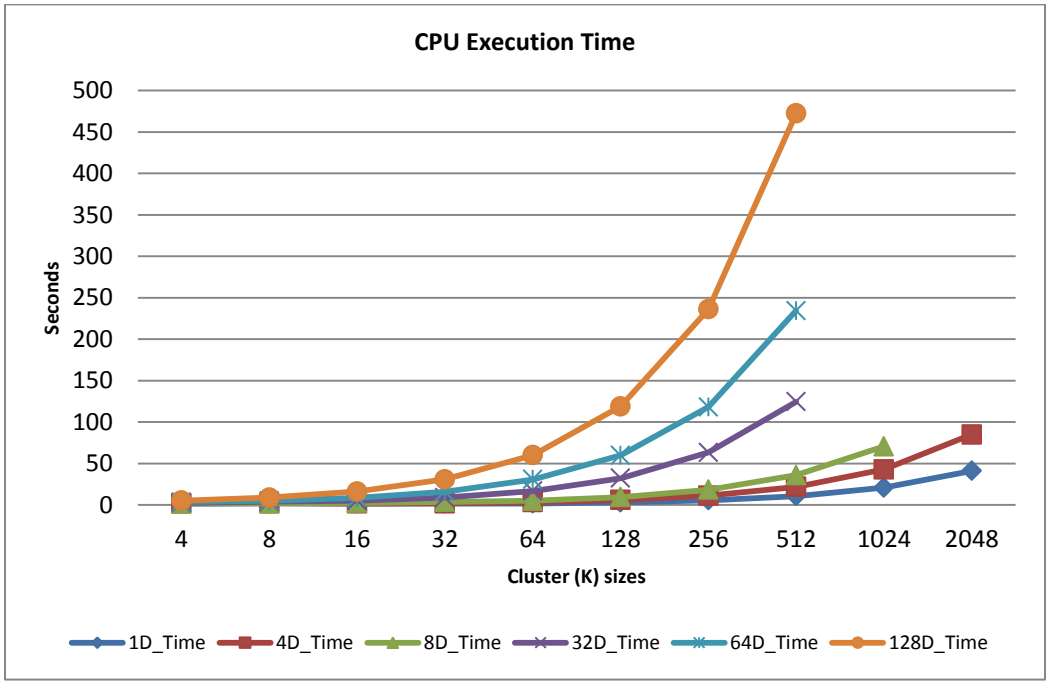

Figure 14. Execution Time for Computing 2 Million Objects on CPU

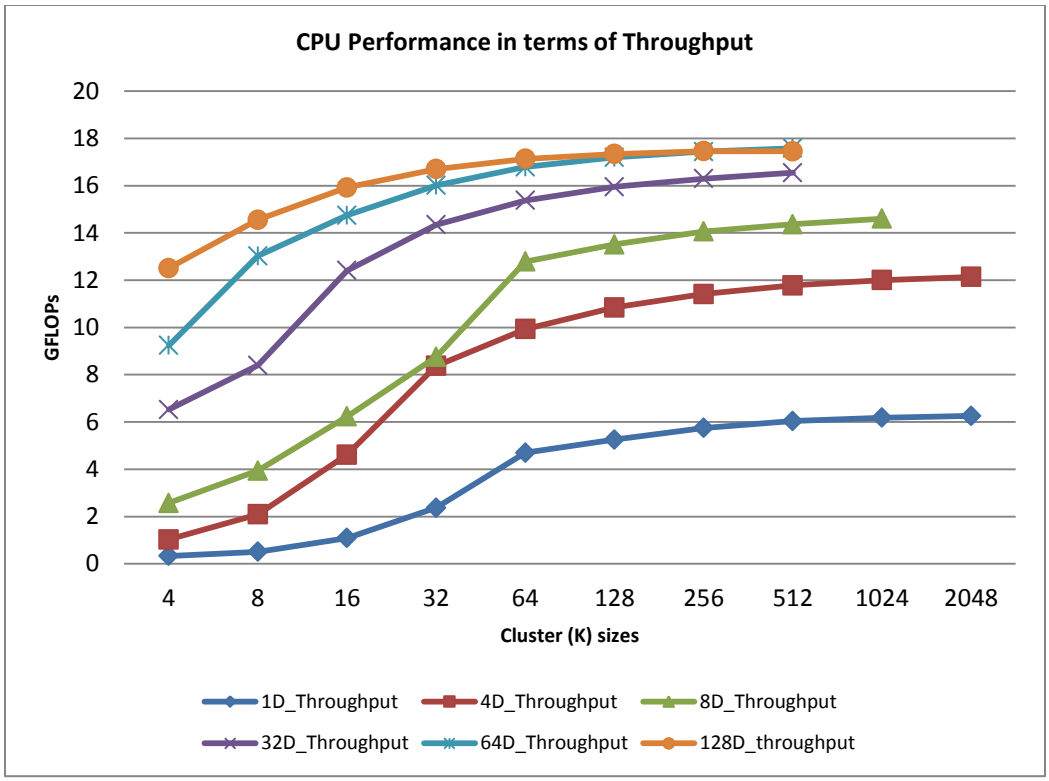

Figure 15. Peak Throughput for Computing around 2 Million Objects on CPU

To better compare with the FPGA implementation, the speedup of FPGA over CPU implementation for identical condition is shown in Figure 16. From this figure we can see that the FPGA gains the most speedup when the dimension size is small and the cluster size is large. The speedup of FPGA implementation over the CPU version could reach up to 19 times. When the cluster size is large the speedup decreases, but overall FPGA is still faster than CPU by multiple times. When cluster size is smaller than 16 or 8 however, the FPGA is not as competitive as CPU. Clearly, when processing larger data sets, the FPGA will outperform CPU more.

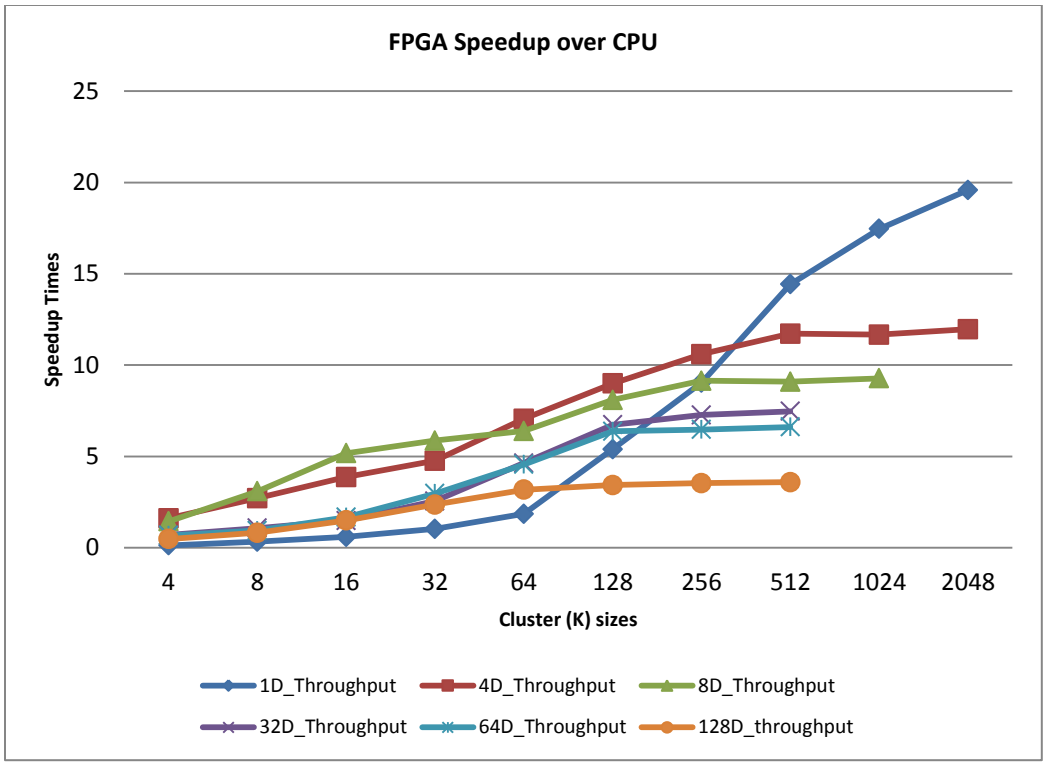

Figure 16. Speedup of FPGA over CPU in Term of Throughput

In order to study how FPGA and CPU implementation perform when processing data with different cluster size *k*, object size *n*, and iteration sizes *iter*. The throughput of processing 4 dimensional data with various cluster, object and iterations sizes is shown in Figures 17, 18 and 19. Conclusions could be drawn that as soon as cluster size went over 32 or object went over 4 thousand the FPGA starts to outperform CPU significantly. The FPGA performs consistently across any iteration size, while performance of CPU slightly improves when more iterations are needed. But improvement is insignificant after number of iterations is greater than 32.

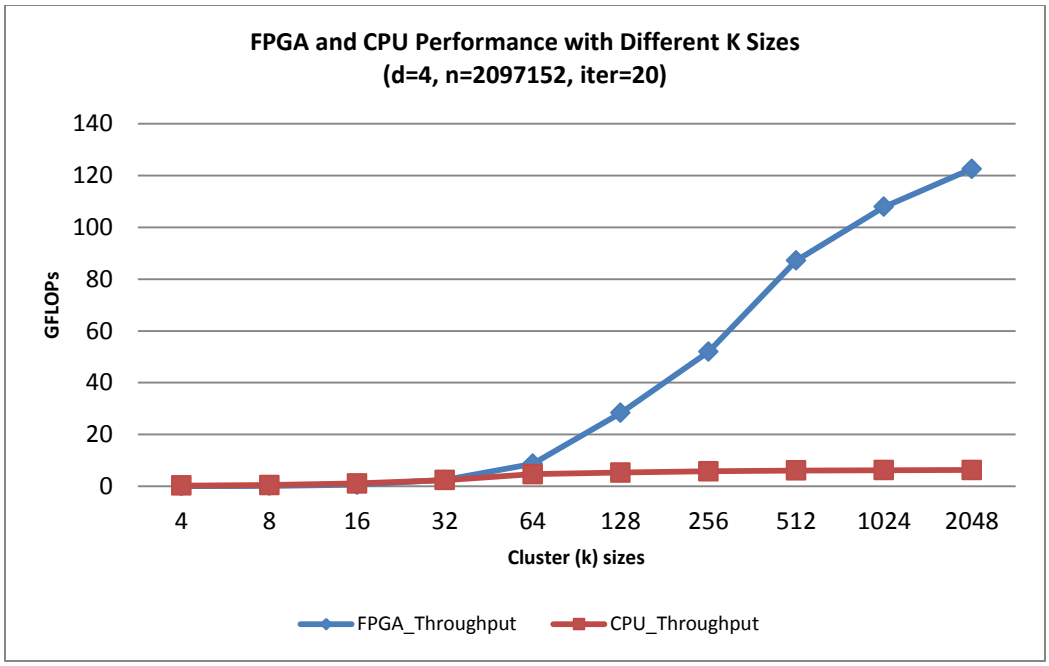

Figure 17. CPU and FPGA Throughput with Varying Cluster Sizes

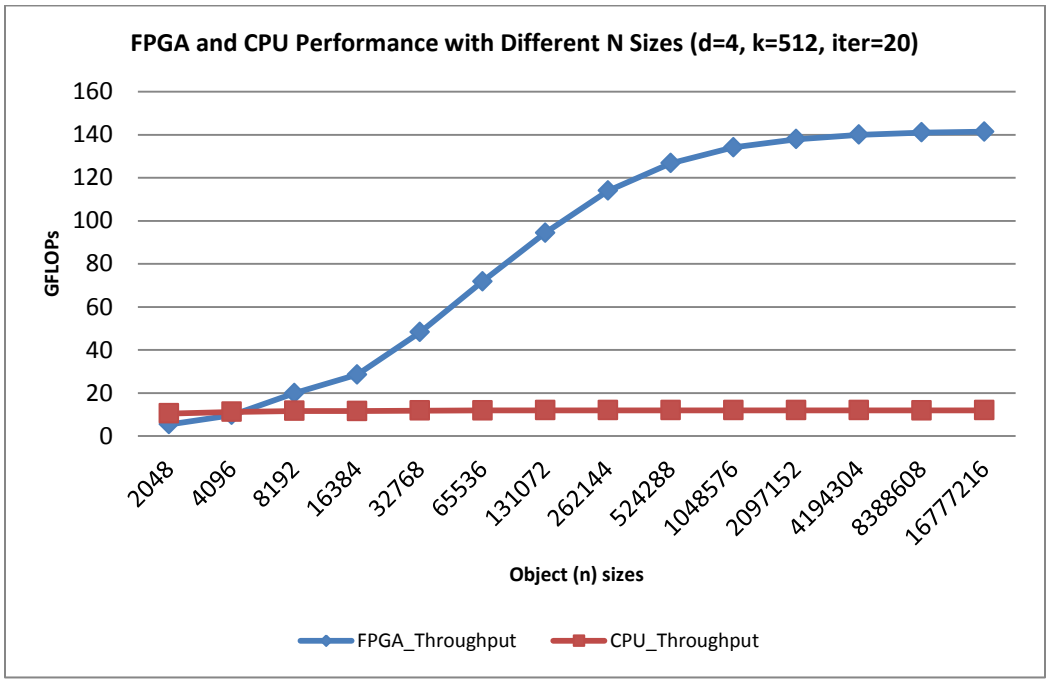

Figure 18. CPU and FPGA Throughput with Varying Object Sizes

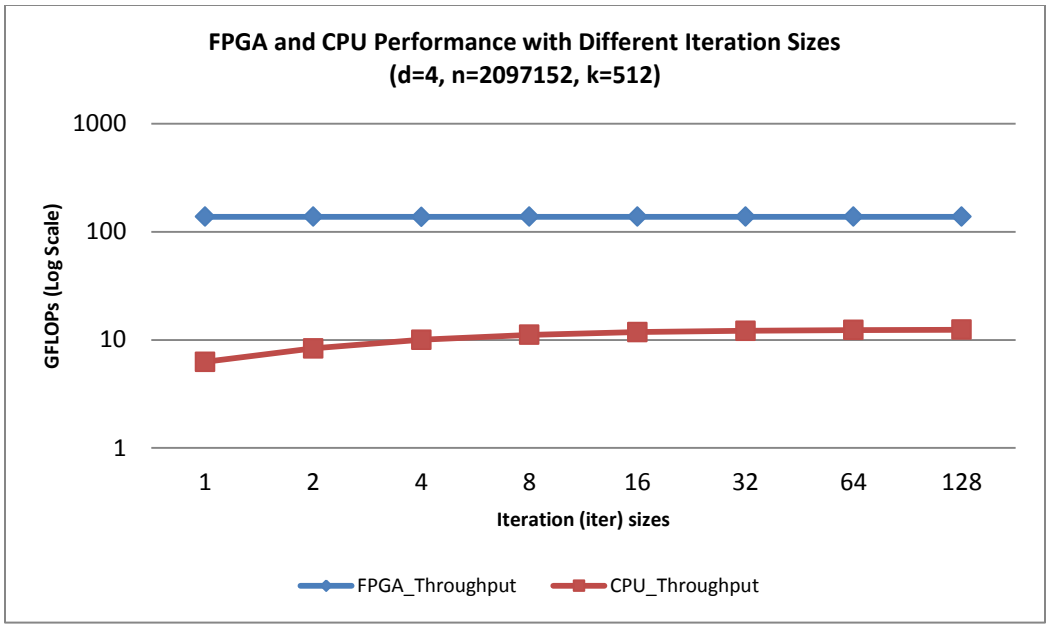

Figure 19. CPU and FPGA Throughput with Varying Iteration Sizes

The peak throughput for this AOCL implementation for various feature and *k* size is summarized in Table 3. The peak throughput for CPU implementation is also listed for comparison. The peak throughput is determined by feeding the FPGA kernels the maximum amount of data that could be held in the global memory of the FPGA accelerator card and with largest possible cluster sizes for each kernel. The peak throughput for FPGA is consistent high from one dimensional feature kernel to 32 dimensional kernels, but starts to reduce after feature dimension exceeds 64. This is because it's no longer possible to fully utilize the DSP resources at large dimension sizes due to logic resource over utilization. As the table indicates, the maximum speedup of 21 times is reached when processing one dimensional data.

Table 3. K-means FPGA vs. CPU Implementation Peak Throughput Result

Feature Dimension	Maximum k	Peak Throughput FPGA[a] (GFLOPS)	Peak Throughput CPU (GFLOPS)	FPGA speedup on Peak Throughput
1	2400	132.77	6.31	21.04
4	2400	150.02	12.26	12.24
8	1200	139.73	14.71	9.50
32	512	123.63	16.71	7.40
64	512	116.78	17.77	6.57
128	512	62.70	17.60	3.56

[a.] Peak throughput of FPGA is measured at maximum *k* with largest possible data size (*n*).

For different data feature dimensions, different kernels need to be compiled. The clock frequency and hardware utilization such as memory blocks and DSP units are different for each kernel. Those utilizations and frequencies are shown in the Table 4. The clock frequency is dependent on the complexity of the HDL design generated by AOCL and the resource utilization of the FPGA. When the resource utilization is close to 100%, it would be much more difficult for Quartus software to fit the design on FPGA. As a result the frequency of the kernel will drop significantly and thus increase latency of the computations.

Table 4. K-means FPGA Implementation Hardware Utilization and Frequency

Feature Size	Max *k* size[a]	Logic Utilization	Memory Block Utilization	Frequency (Mhz)	DSP Utilization
1	2400	81%	85%	184.5	96 %
4	2400	84%	75%	208.46	96 %
8	1200	88%	71%	195.46	96 %
32	512	73%	58%	190.18	89 %
64	512	78%	56%	204.08	77 %
128	512	81%	93%	163.61	52 %

[a.] Problems smaller than or equal to maximum k size could be executed on kernel.

Regarding accuracy of the AOCL implementation, it turns out that it requires identical number of iterations to reach steady state compared to the CPU reference code. The FPGA implementation used floating point optimization flags during compilation to

eliminate redundant floating point rounding, thus the resultant cluster centroids are slightly more accurate than CPU implementation. Although all the data are classified into the same clusters on both FPGA and CPU implementation, difference exists between the cluster centroids calculated on CPU and FPGA. Higher number of iterations will result in slightly larger difference in cluster centroids. For 20 iterations the difference is less than 0.001%.

The current best published work on GPU acceleration of k-means was the CUDA implementation proposed by Li et al. [35] mentioned in related works. It achieved 137 GFLOPS with 34 dimensional data and 676 GFLOPS with 8 dimensional data on GTX280 GPU. The FPGA implementation designed in this research could achieve comparable throughput of 123.63 GFLOPS with 32 dimensional data. However, the FPGA could only achieve much smaller throughput of 139.73 GFLOPS with 8 dimensional data. The performance of FPGA verses GPU varies depending on problem size. When processing mid to high dimensional data set, the FPGA performance is comparable with the GPU results. At smaller data sizes, the FPGA is slower than GPU. Detailed comparison of GPU and FPGA performance was not included, due to the fact that the source code for the best GPU implementation in literature was not published. Other CUDA or OpenCL based k-mean program available either was outdated or could not work with large problem sizes that was used in this research.

3.4.2 Power

The Terasic FPGA accelerator board has a maximum power consumption of about 40W, while the Thermal Design Power (TDP) of CPU is 130W and the TDP of GTX280 GPU is 236W. Assuming full power utilization and ignoring the power

consumption of the rest of the system, the difference between energy efficiency of FPGA, CPU and GPU could be estimated by using FPGA speedup multiplied by FPGA energy consumption and divided by energy consumption of system under comparison. Theoretically at peak throughput FPGA implementation is up to 108 times more energy efficient in term of GFLOPS/Joule than CPU, and around 2 to 9 times more energy efficient than GPU.

With a *Watts up? PRO* [39] power meter we could take the power utilization of the whole system into account and calculate power savings more precisely. Before FPGA accelerator card is installed into the system, the idle system consumes 75 Watt power. When performing k-means using Mine-bench with all 6 cores, the power utilization increases to 175W on average. On the other hand, when the FPGA accelerator is added to the system, the idle power usage is increased to 96W. During execution of AOCL kernel, only one of the CPU core is active to execute the host program, thus the CPU utilized less power than before. But since the FPGA consumed more power when executing kernels, the total average power utilization increased to 126 W. Thus when utilizing the FPGA accelerator, the system overall power consumption is reduced by (175-126)/175=28.0%. Adding the fact that the FPGA implementation could finish up to 21x faster than the CPU version, the total energy reduction is (175*21-126)/(175*21)=97.5%; or equivalently, the FPGA implementation is 29.2 times more energy efficient. The power consumption of CPU and FPGA while executing k-means on 4 dimensional data with various clusters sizes is shown in Figure 10. The idling period before and after program execution is marked on the chart.

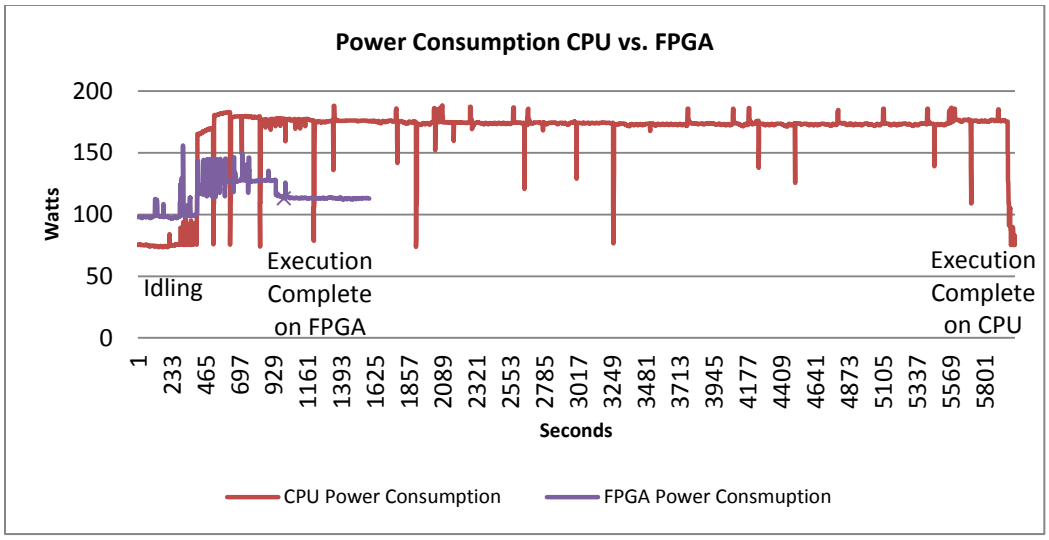

Figure 20. Power Consumption of CPU and FPGA

Due to the fact that all parts of k-means computation are performed on FPGA, it is possible to use a much weaker but more power efficient CPU for this particular application without affecting performance. This could reduce system power usage overhead and make the FPGA platform even more favorable for power sensitive applications. The kernels can be compiled to target Nallatech accelerator and similar performance could be achieved. However, due to Nallatech board BSP used more FPGA area to implement memory interfaces, only kernels with slightly smaller utilization could be fitted. The energy efficiency could be nearly twice as high as the Terasic accelerator, because the Nallatech accelerator uses less than 25W of power for computation.

3.5 Discussion

The AOCL implementation of k-means algorithm presented in this thesis running on Stratix V A7 FPGA is able to achieve 3 to 21 times speedup and is up to 29 times more energy efficient compared to an optimized CPU implementation running on six core Xeon processor. The performance of FPGA is comparable with state of the art GPU

implementation at mid to large data sizes, but is slower than GPU at smaller data sizes. The power efficiency of AOCL FPGA implementation is estimated to be better than that of best GPU implementation described in literature.

One of the limitations of the presented k-means implementation is that it is optimized toward getting the maximum peak performance. Therefore, they only work well in problems with large cluster size. When size of cluster is low, the kernel performs poorly. Also, the maximum problem dimension that the AOCL implementation supports before running out of FPGA resources is around 128 to 160. However, the kernels performs the best when the dimension size is smaller than 64. The kernels compiled for size above 64 did not perform as well as lower dimensional kernels. It may be possible to fix both problems with multiple FPGAs, where one FPGA performs reduction operations while the others execute cluster assignment operations. The cluster data and other temporary data could be transferred between FPGAs via 12.5 Gbps high speed transceiver.

In addition, it would be interesting to compile and test the kernels developed in this research on the newer FPGAs. The new generation 10 FPGAs has substantially more DSP resources than the Stratix V A7 FPGA used in this research. Each floating point operation would consume reduced logic, local memory and register resources due to improved architecture. At the same time due to newer 20nm fabrication technology used, it could run at higher clock speed with better power efficiency. The Global memory bandwidth is increased to a level comparable with GPUs as well. It is likely that the FPGA could outperform GPU for all problem sizes.

Chapter 4

Acceleration of K-Nearest Neighbor Search

4.1 Introduction to K-Nearest Neighbor Algorithm

Similar to k-means, the k-nearest neighbor or kNN algorithm is another one of the most popular machine learning algorithms [29]. It is mostly used in pattern recognition and data mining applications. kNN could be used to solve regression or classification problem. Depending on the problem, output of kNN could be either the most prevailing class of the k-nearest neighbors in the case of classification, or the average of the k-nearest neighbors in the case of regression. kNN is an example of supervised learning, where sample data with reference output is provided during the training phase. Query data is provided during the classification or regression phase. The goal is to find the k-closest neighboring reference points to the query points, and use the neighbors to predict the class or expected value of the query points.

While the training phase of kNN is as simple as remembering the sample data, the process of finding the k-nearest neighbors could take a long time when data dimensions, number of samples or number of querying data is large. The most computationally intensive part of kNN algorithm is finding of the nearest neighbors for query data, which is the main focus for acceleration. After the nearest neighbors are determined, their most frequent class or average could be easily computed on CPU.

The direct approach of kNN is the brute-force algorithm, which involves use of similarity function to measure the pair wise distances between the query point and every reference points, and then sorting the distances in ascending order to determine the k-

nearest reference points. For similar reasons mentioned in chapter 3, the similarity function used in this research is restricted to squared Euclidean distance. The simplified pseudo code for the algorithm is shown below.

ALGORITHM 3. Sequential Brute-Force KNN Algorithm

```
input: A set of reference points R and query points Q, dimension size D,
       query size M, reference size N, and cluster size K
output: A set of k nearest reference points for each query point q (indexes)

for each query point q in Q do
  dist [N]← 0;
  for each reference point r in R do
       for each dimension d do
           dist[r] += distance (q , r);
        end
  end
  sort(dist);
  select K reference point with smallest distance to query point q;
end
```

The brute-force algorithm is obviously not the most efficient solution for kNN. Since the reference points far away from the query point are unlikely to be neighbor points, optimally they could be eliminated for distance calculation. In training phase instead of storing reference points linearly, advanced data structure such as k-d tree or k-dimensional tree could be generated by recursively partitioning the sample space using the reference points. When query point is supplied, the nearest neighbors could be found by traversing the k-d tree and recursively searching for closer reference points. Due to the inherent property of the k-d tree, each time a lower level is reached, a large amount on unsuitable reference points are eliminated, and far few distance calculations are needed. However, due to their complexity, various tree-based kNN algorithms are very difficult to parallelize and thus are not implemented in this research.

4.2 Related Works

An optimized CPU based approximate k-NN algorithm [40] was proposed in 1998. The authors also published the source code of their algorithm along with a brute-force exact version of k-NN algorithm in the form of a library called ANN [41] library. The brute-force CPU implementation of kNN provided in this library is used in this research for performance comparison.

In 2008, Garcia et al. from University of Nice Sophia Antipolis proposed a CUDA implementation of the brute force kNN algorithm [42]. When comparing with the brute force kNN algorithm in ANN, they claim a speedup of one to two orders of magnitude could be achieved. An updated implementation [43] using CUBLAS API was proposed in 2010, which further improved the performance and was used to demonstrate an image feature matching application. The source code for this CUDA implementation of brute force kNN search was published on GitHub [44] and was used in this research for speed comparison.

A parallel implementation of kNN algorithm using truncated btionic sort [45] was presented in 2012. On GPU the proposed sorting algorithm was able to significantly outperform *thrust::sort* radix sort function provided in CUDA Toolkit. A summary of various truncated sorting algorithms was also provided in the paper.

A dynamically reconfigurable kNN classifier implementation [46] on Xilinx Virtex 4 FPGA was presented in 2012. The researchers claim that it was 68 to 76 times faster than sequential Matlab implementation running on a Pentium E5300 CPU.

Stamoulias et al. from University of Athens presented their design [47] of a flexible IP core kNN classifier for FPGA in 2013. They were able to achieve 1.369 GOPS on Xilinx Virtex XC2VP30-6 FPGA. While more power efficient, it was 10 times slower than an earlier GPU implementation published in 2008 and only works with small data set.

In 2014, Komarov et al. from University of Wisconsin-Milwaukee described a new brute force kNN algorithm [48] that uses quick select instead of sorting algorithms to determine the nearest neighbors. On problem with very large cluster sizes, they were able to achieve over 100 times speedup over the CUDA KNN CUDA published in 2008. However, the speedup was insignificant when cluster size is smaller than 64.

4.3 Altera OpenCL Implementation and Synthesis

The kNN algorithm implemented in this research divides the distance calculation and sorting process into two separate kernels. Their implementation and optimizations are discussed in the subsequent sections.

4.3.1 Distance Calculation

The distance calculations have the time complexity of O (M*N*D), where M is the number of query points, N is the number of reference points and D is the dimension size of the data. This is the most computationally intensive part of the kNN process, and should be placed in a separate kernel. Since the distance calculation for each query point is independent, it is rather easy to map the distance calculations to thread parallelism. There are two ways to calculate the distance in parallel. First, one dimensional NDRange could be used. The calculation of each query data could be mapped to a different thread.

A nested *for* loop could be used to loop through each of the query data point and their dimensions. The inner *for* loop iterating over the dimensions of the data could be unrolled in order to increase throughput. This is similar to what was developed for the k-means kernel, except now there is less data reuse. It is also possible to map the distance computation to two dimensional threads. In this paradigm each reference data point maps to the one work-item in the first dimension, and each query point maps to one work-item in the second dimension. Each thread passes through a for-loop over the dimensions of the data, which could be unrolled to increase throughput.

In order to reduce the number of global memory operations, local memory could be used to temporarily store multiple query or reference points, and share them with in a single work-group so the threads does not have to read them from global memory every time a new pair-wise distance needs to be computed. The blocking operations for the one dimensional work-group version of the distance calculation is illustrated in Figure 21, whereas the blocking operations for the two dimensional version of distance computation is illustrated in Figure 22. Notice that the two dimensional blocked distance calculation kernel is simply a scaled down version of the matrix multiplication. When dimensions for the data is too large and cannot fit into FPGA's local memory, a full scaled version of matrix multiplication could be used.

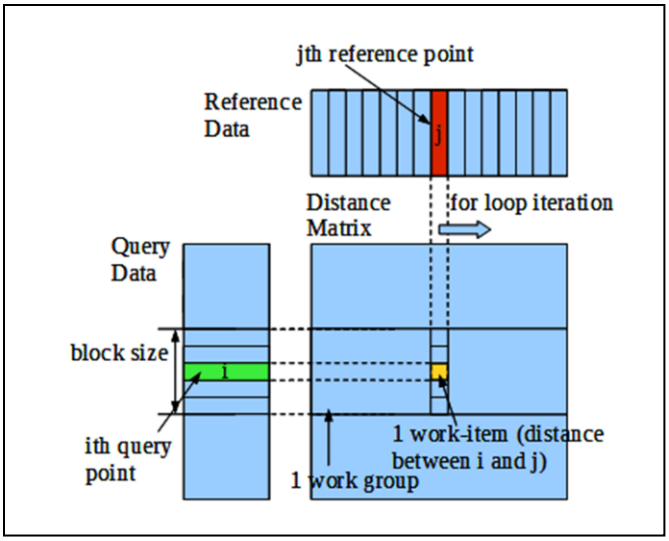

Figure 21. Visualization of 1D Blocked Distance Calculation Kernel

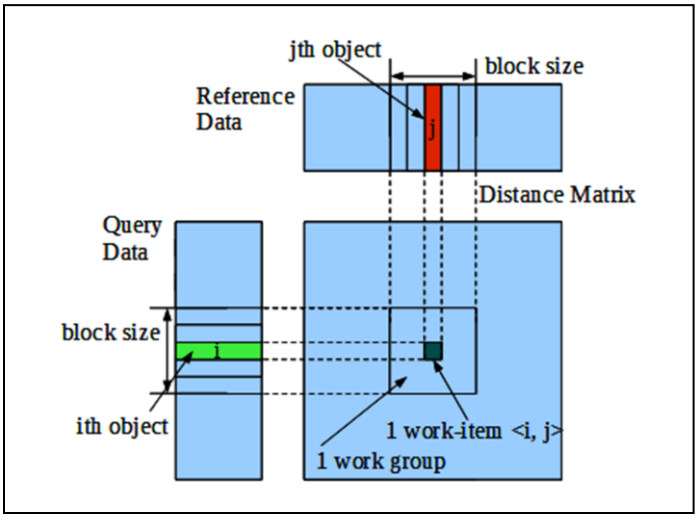

Figure 22. Visualization of 2D Blocked Distance Calculation Kernel

In one dimensional blocked distance kernel, one reference points is loaded into local memory and are reused by all the query data in the same work-group. Whereas in the two dimensional kernel, one block of query and reference points are loaded into local memory by all work-items in the work-group. All pair-wise distance are computed using local memory and the results are written back to global memory after the whole block is processed. The 2D threaded kernel is constructed based on the matrix multiplication

example [49] published by Altera. The saving in global memory operation is proportional to the block size in both cases. Larger block will use significantly less global memory bandwidth.

4.3.2 Sorting Algorithms

There a lot of different sorting algorithms that could be used in kNN. Sorting algorithms usually involve heavy global memory access. A lots of work has been done on optimizing sorting on CPU and GPU. In order to compete with CPU and GPU platforms, it is very important to take the advantage of the fact that the cluster size is usually much smaller than the total number of reference points, and thus the distances does not have to be fully sorted.

One of simplest sorting algorithm is insertion sort [50]. It has the same average and worst case time complexity of O (N^2) as the notorious bubble sort. However, instead of going through the whole array repeatedly and swapping values constantly, the insertion sort keeps track of sorted and unsorted list separately and only swap data when necessary. When sorting a new value, the insertion sort inserts it into the proper location in the sorted list. When applied to kNN search, the sorted list can be stored in local memory with size only as large as the number of clusters. At the same time insertion sort could be done completely locally and read the distance array exactly only once, which is good for FPGA implementation. In order to parallelize insertion sort, each query point is mapped to a different thread. The distance values from multiple query points are processed concurrently while the sorting process itself utilizes task parallelism. Additional optimizations are applied. For example, first K distance values were used to fill the sorted list, while subsequent distance values were only inserted into the array if it

is actually smaller than at least one of the values already in the sorted list. Pseudo code for insertion sort optimized for the AOCL FPGA implementation of kNN designed in this research is provided below.

ALGORITHM 4. Insertion Sort Algorithm (for one query point)

```
input: Array of distance values dist[] with size N
output: A set of nearest reference point indexes clusters[] with size K

initialize local memory dist_local[] to INF and index_local[] to 0
for i from 0 to N-1 do
  dist_new = dist[i];

  // determine whether filling the sorted list or inserting new value.
  if i < K  then
      set sort_limit to i;
  else
      set sort_limit to K;
  end

  // check where the new value should be added into the sorted list
  for j from sort_limit down to 1 do
      if  dist_new < K then Break;
      dist_local[j] =  dist_local[j-1];
      index_local[j] =  index_local[j-1];
  end

  // add to sorted list if the new distance is smaller
  if i < K or j != K  then
      dist_local[j] = dist_new;
      index_local[j] = i;
  end;
end

// write the index of K closest reference point back to global memory
for i from 0 to K-1 do
    clusters[i] = index_local[i];
end
```

Heap sort [51] [52] is an efficient sorting algorithm with average and worst time complexity of O (N log N). The heap sort algorithm relies on a binary tree based data structure called heap, where all levels of the tree except the lowest is complete filled. The array based implementation of heap data structure is shown in Figure 23. Here each level

of the heap structure occupies into 2^L elements in the array, where L is the depth of that particular level. In a complete heap sort, the unsorted input is first used to construct the heap data structure. After which the largest or smallest value in heap could be returned by repeatedly popping the root and then fixing the heap.

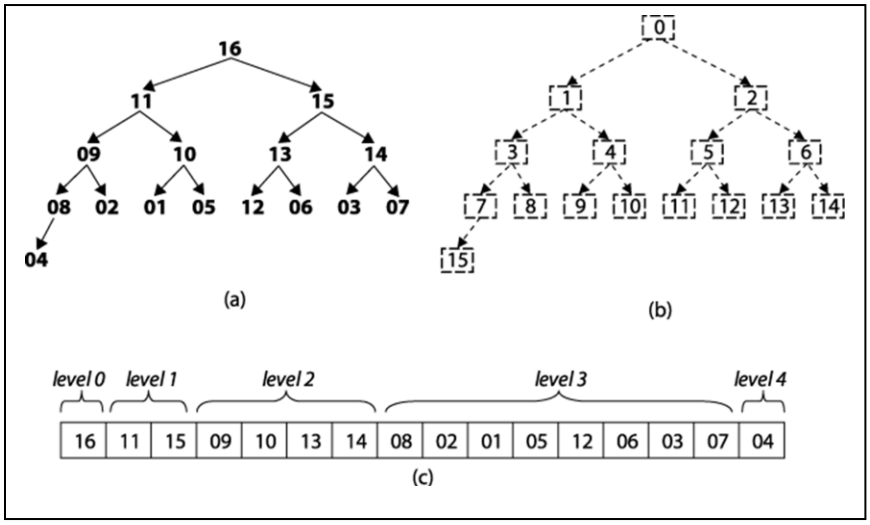

Figure 23. Visualization of Heap Data Structure Implemented Using Array [53]

A modified heap sort algorithm optimized for the kNN search is also designed in this research. A max heap is used in kNN application, where the largest value is sorted in 1^{st} index of the heap as root. Since kNN search only interests in finding k-minimal points and not interested in the order of the neighbors, the heap data structure can be stored in local memory only as large as the number of clusters. For each query point, the first k distance values are used to build the initial heap, while all remaining distance values are used to update the heap. After all distance values are processed, the indexes stored in heap array are returned as the indexes of reference points with smallest distance to the query point.

Similar to the insertion sort mentioned earlier, the building and maintenance of heap data structure is done completely locally and the entire distance array is only read once. One work-item is allocated for each query point, while distance values from multiple query points are processed concurrently. Additional optimizations are applied. For example, the heap array is padded with an extra 0^{th} element in the beginning, so that indexing of every level in heap is power of 2. This way the shift operations could be used instead of more expensive division and multiply operations to index the heap. In the actual AOCL implementation, *while* loops where used to perform heap build and fix functions, which is not optimal since the compiler could not design fully optimized pipelines with *while* loop. An alternative *for* loop implementation of the heap sort was attempted. Unfortunately while the kernels with *for* loop successfully passed emulation, the results were incorrect when compiled to hardware, so it could not be used. A simplified version of heap sort pseudo code optimized for the FPGA implementation of kNN is provided below.

ALGORITHM 5. Heap Sort Algorithm (Simplified)

```
input: Array of distance values dist[N]
output: A set of nearest reference point indexes clusters[K]
local memory: Heap data structure is stored in dist_local[K + 1] and
index_local[K + 1]

initialize local memory dist_local[K + 1] to INF
initialize local memory index_local[K + 1] to 0
for i from 0 to N-1 do
  dist_new = dist[i];
  if i < K then
      //if heap is not filled
      append dist_new into the Heap;
  else
      //if the heap is filled
      if dist_new < dist_local[1] then
          //if dist_new is smaller than the largest value in heap
          use dist_new to replace the current heap root;
          fix the heap;
      end
  end
```

```
end

//write the index of K closest reference point back to global memory
for i from 1 to K do
    clusters[i] ← index_local[i];
end
```

One of the most efficient soring algorithms on CPU is quick sort. Quick sort randomly selects pivot points, and use them to partition the data array into smaller sub-arrays where smaller and larger values in the array placed in order. However it does not map to parallel architectures very well. Quick select algorithm derived from quick sort could be used to efficiently determine the k smallest numbers from an array. However, it requires the host to control the execution of sorting and supply the pivot during each partition, and thus may not be a good fit for FPGA. Similar to quick sort it has a worst case time complexity of O (N^2), but has an average time complexity of O (N) instead of O (N log N).

The fastest sort used on GPU is non-comparative radix sort, which recursively partitions the keys based on whether the individual bits of each key is zero or one. The radix sort has a linear average and worst-case time complexity of O (k*N), where N is the total data size and k is the number of bits the each key. For single precision floating point data type k is 32. The radix needs to go through the entire array of data multiple times. In each run it requires repeated radix sum operations for index calculation as well as swapping the data around in order to achieve optimized memory access, and thus may not be a good fit for FPGA.

4.3.2 Implementation Specifics and Use of Channel Extension

In order to determine the best way to compute distance for kNN search, test kernels with various data dimension where constructed to compare the 1D and 2D kernels. After testing it was determined that the 2 dimensional version of the kernel not only give slightly better performance, but also use less FPGA reconfigurable resources. Thus the 2D version of distance kernel is used in final version of the kNN. Kernels optimized for data dimension sizes of 64, 80, and 128, and cluster sizes from 4 to 32 were selected for performance testing. The inner loop for distance kernel with those dimensions are fully unrolled. For the dimension size of 64, SIMD factor of 2~4 could be applied dependent on how large the cluster size is. While for dimension of 80 and 128, SIMD factor of 1~2 could be applied. Setting higher SIMD factor allows the compiler to design hardware that could execute more work-items in parallel, but will cost more FPGA hardware resources. Distance kernel with dimension size lower than 32 are proven to be inefficient due to insufficient parallelism and data reuse. Although untested, the kernels could easily be modified to process data with dimension larger than 128 without performance penalty.

Both insertion sort and heap sort are implemented and optimized in this research. During testing it is determined that the heap version of the sorting kernel was much faster than insertion sort in all problems sizes, and thus heap sort was used in the final version of the kNN kernels.

In order to minimize global memory access, use of Altera Channel Extension to transfer distance data from the distance kernel to the sorting kernel was attempted. However, while it passed emulation, the kernels with channel extension applied would often get stuck in execution. In some kernels compiler error relating to LLVM was

encountered during compilation. Only a few kernels with heap sort and cluster size of 4 and smaller worked with channel extension, and the speed improvement was not significant. Therefore, Channel Extension is not used in the final version of kNN kernels.

4.4 Result and Discussion

The Results from FPGA implementations synthesized using Altera SDK for OpenCL running on Stratix V A7 FPGA is compared with results from ANN library and kNN CUDA mentioned in related work. The ANN library was compiled using GCC compiler with level 3 optimization enabled and debugging disable, and the kNN CUDA source code are compiled with CUDA SDK version 6.5. The ANN code was tested on Intel Xeon E5-2637V3 CPU, which has 4 cores running at maximum frequency of 3.7GHz, 15MB of cache, 68 GB/s memory bandwidth and a TDP of 135W. The CUDA code was tested on NVIDIA K620 GPU with 384 CUDA cores running at maximum frequency of 1.124GHz and 45W TDP. This GPU has a peak single precision floating point throughput of 812.5 GFLOPS. The FPGA used in performance test was DE5-net accelerator while the FPGA used in testing the power utilization was Nallatech 385 accelerator. Both cards contain Stratix V A7 FPGA and returns similar performance, with the exception that the Nallatech accelerator has more memory to fit larger data sizes, and kernels targeting Nallatech board sometimes have higher FPGA resource utilizations.

Three sets of performance tests are conducted. The first set of tests was conducted with varying dimension size for 64 to 128. The second set of test was conducted with constant dimensions size of 128, but varying the cluster size from 4 to 32. The third set of test was conducted with constant 128 dimensions and 4 clusters while varying the query

and reference data size. The number of query and reference data point was set to be identical for simplicity. The results are summarized in Table 5, 6, and 7.

Table 5. kNN Performance with 16384 Samples, 4 Clusters and Various Dimension Sizes

Dimension Size	FPGA DE5 Time (s)	ANN CPU Time (s)	CUDA Time (s)	CUBLAS Time (s)
64	0.46477	3.13379	0.60076	0.39306
80	0.92264	6.29346	1.00081	0.3368
128	2.1163	26.47461	3.55568	0.52022

Table 6. kNN Performance with 128 Dimensions, 16384 Samples, and Various Cluster Sizes

Cluster Size	FPGA DE5 Time (s)	ANN CPU Time (s)	CUDA Time (s)	CUBLAS Time (s)
4	2.11527	26.47607	3.55568	0.52022
8	2.1555	26.49854	3.56075	0.52982
16	2.60873	26.53711	3.62862	0.58699
32	3.67524	26.54639	3.80429	0.76323

Table 7. kNN Performance with 128 Dimensions, 4 Clusters and Various Data Sizes

Cluster Size	FPGA DE5 Time (s)	ANN CPU Time (s)	CUDA Time (s)	CUBLAS Time (s)
128	0.00143	0.00526	0.16934	0.22893
256	0.00181	0.0118	0.17724	0.2301
512	0.0034	0.03316	0.18966	0.23147
1024	0.00969	0.11275	0.22775	0.24067
2048	0.03452	0.4202	0.22775	0.23147
4096	0.13382	1.66472	0.3973	0.25404
8192	0.53037	6.6235	1.02513	0.30444
16384	2.11518	26.44714	3.54027	0.517

The tables show that while the FPGA performs well with very small clusters sizes, the performance drops sharply with increasing cluster sizes. Figure 24, 25 and 26 illustrates the speedup of FPGA and GPU implementation over the ANN library running on CPU.

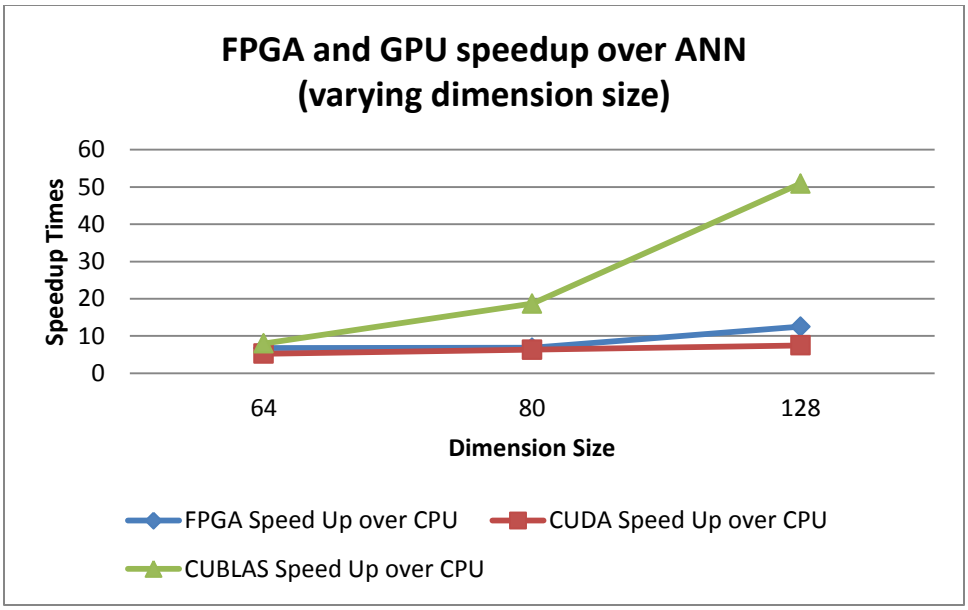

Figure 24. Speedup of FPGA and GPU over CPU with Varying Dimension Sizes

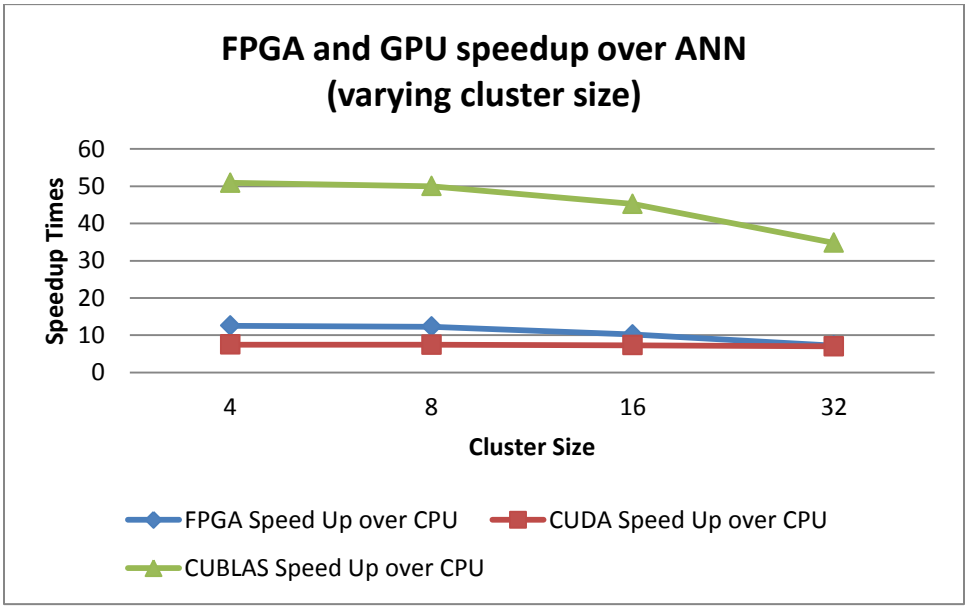

Figure 25. Speedup of FPGA and GPU over CPU with Varying Cluster Size

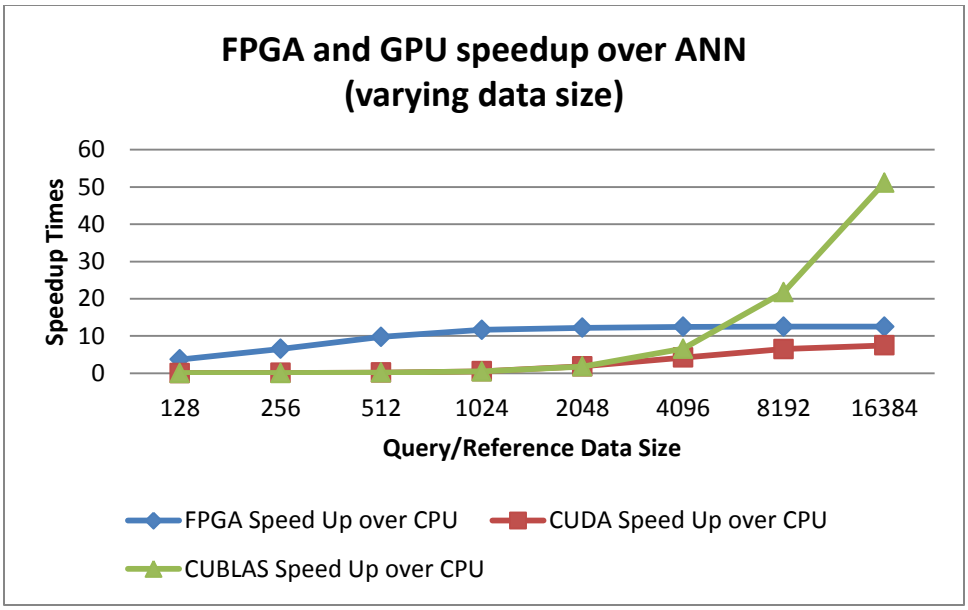

Figure 26. Speedup of FPGA and GPU over CPU with Varying Data Size

The plots indicate that the FPGA implementation of kNN design in this research outperforms ANN library running on CPU by a factor of more than 10 times. The FPGA could also outperform the GPU with tiny data sizes, but with any sufficiently large data or dimension sizes the GPU outperforms the FPGA implementation.

In addition to performance tests, the power consumption of CPU, GPU, and FPGA was tested with *Watts up? PRO* power meter. The test condition chosen for the power test is to repeat kNN search on 128 dimensions, 4 clusters and 16384 query and reference data points for 4 times. The result is summarized in Table 8. Notice that the energy is calculated by using the difference between power while running the kernel and idling multiplied by the computation time. The resource utilization of various kernels along with maximum frequencies used in the test is summarized in Table 9.

Table 8. Power Utilization of Various kNN Implementations

Power Test	Total time (s)	Idle power (W)	full power (W)	TDP Rating (W)	Total Energy (J)
FPGA (385 A7)	9.04185	72.3	82	25	87.70595
ANN (Xeon E5)	105.33008	52.6	88.2	135	3749.75085
CUDA (K620)	14.17392	60	108.5	45	687.43512
CUBLAS (K620)	1.88504	60	105	45	84.8268

Table 9. FPGA Resource Utilization and Frequency of Various AOCL kNN Kernels

Kernels (d = dimension, K= clusters)	Logic % (234K ALM total)	I/O pins % (1064 total)	DSP blocks % (256 total)	Memory bits % (52Mbit total)	RAM blocks % (2560 total)	Kernel fmax (MHz)
64d_4simd_4k	83	26	100	19	41	208.46
80d_2simd_4k	60	26	66	29	54	204.24
128d_2simd_4k	81	26	100	19	42	221.28
128d_2simd_8k	86	26	100	20	44	217.24
128d_16k	67	26	54	22	47	211.64
128d_32k	92	26	54	24	57	162.6

From the tables, we can see that the FPGA implementation is nearly 50 times more power efficient than CPU, and is on par with best GPU implementation. Increase in cluster size causes the logic utilization to increase dramatically while the maximum frequency drops sharply.

During profiling, it turned out that for 64 dimension kernels with 4 clusters, the time it takes to perform sorting is roughly five times it take to compute the distance. For 128 dimension kernels with 4 or 8 clusters the sorting takes about twice as long compared with distance calculation. This clearly indicates that the sorting algorithm implemented is not fully optimized. While bad sorting performance with large cluster size could be mediated by using more recent FPGA such as Arria 10, better sorting method should be

explored. Modified quick selected and bitonic sort algorithm could be attempted to see if better sorting performance could be achieved.

Chapter 5

Acceleration of N-body Simulation

5.1 Introduction to N-body Simulation Algorithm

N-body simulation is one of the easier algorithms to parallelize and is one of the popular benchmark to measure CPU and GPU performance. It is a physics simulation on a system of bodies that interact with each other through some form of force. Some example applications of N-body simulation include simulation of galaxies, where the interactions are caused by gravitational forces; or molecules, where the interactions are carried out by electrostatic and Van der Waals forces. The most simplistic way of implementing N-body simulations is all-pairs method. In each time step, the speed, location and acceleration of each body is updated by calculating the force of all affecting bodies. Updating the parameters of one body in one time step requires calculating the force interaction from remaining N-1 bodies, thus the computational complexity is in the order of O (N^2) for N-body simulation. When the problem size is sufficiently large, more complex algorithms can be used to obtain approximated result more efficiently. One of such algorithm is Barnes-Hut simulation [54]. It takes the advantage of the fact that objects far away from the body under update have little effect on the said body and could be neglected. Barnes-Hut simulation utilizes octree data structure to partition the bodies into 3 dimensional cells. All the objects that are more than a certain distance away from the body under evaluation are not directly used in force calculation, but instead they are treated as one object located at the centroid of their cell. As a result, the computational complexity of the algorithm is reduced to O (N log N) instead of O (N^2), at the cost of less accuracy. In our research, only the pair wise algorithm is implemented in order to

achieve maximum throughput. The application chosen for our implementation is simulation of galaxies. The acceleration of each body caused by gravitational force interactions can be calculated as follows:

$$a_i = \mathrm{G} * \sum_{j=1}^{N} \left(\frac{m_j r_{ij}}{(||r_{ij}||^2 + \varepsilon^2)^{3/2}} \right) \qquad [55]$$

Where r_{ij} is the distance between the pair of bodies and m_j is the mass of the interacting body. To avoid complex decimal exponent calculations, the denominator in the summation is calculated using three multiplications and an inverse square root (*rsqrt*) operation. When three dimensional space is considered, the total number of floating point calculation is 20 if the inverse square root operation is considered as a separate division and square root operation.

5.2 Related Works

NVIDIA published a CUDA based brute-force parallel N-body simulation [55] in GPU Gems 3 book. An updated version of this implementation was provided as an example in CUDA SDK Toolkit, which is used for GPU performance comparison in this research. Intel provided a fully optimized OpenCL implementation [56] of N-body simulation that works in a similar fashion. It is used for CPU performance comparison in this research.

A highly efficient FPGA implementation [57] of N-body simulation was introduced in 2009. This FPGA implementation used both fixed point and logarithmic number format for different parts of computations. The researchers compared performance from CPU, GPU, ASIC and FPGA and found out that while GPU has the

highest throughput, their FPGA implementation targeting Xilinx Sparta3 XC3S5000 FPGA could achieve the highest power efficiency of 49 GFLOPS per Watt, which is an order of magnitude higher than any other platform.

Segal et al. briefly discussed an AOCL implementation [38] of N-body simulation. This implementation is ported from high level Java code automatically to Altera SDK for OpenCL by using theAPARAPI Java Framework. On Stratix V A7 FPGA it was 4.8~5.3 times faster when compared with CPU implementation.

5.3 Altera SDK for OpenCL Implementation

The computation of each body is independent of other bodies and thus could be computed using different threads, and various optimizations such as SIMD vectorization could be used to increase the throughput. The computation done on individual bodies however needs to be summed in the end using reduction. Three different approaches were attempted in this research. First approach is straight forward implementation without considering data reuse. One dimensional work-groups were used, where each work-item maps to a different body. Second approach is to utilize data reuse, where multiple work-groups are used, each reads a subset of data and shares it among all the work-items in the work-group. Data reuse results in reduction of memory transfer to the global memory by local group size times. Although this application is not memory intensive, any saving is helpful. The third approach is similar to the second approach, but task parallelism is used instead of multiple threads, and the computation is implemented in a shift register fashion.

During testing it turned out that the second approach gives the best performance, thus it is used in the final version of FPGA N-body kernel. Also, loop unrolling is preferred over SIMD due to higher DSP utilization. With loop unrolling factor of 23, 98% of DSP resources available in A7 FPGA are used up.

Mixed point implementation using logarithmic number system does not fit OpenCL programing model, and could not be implemented without adding customized Verilog models, which defeats the purpose of HLS and was not implemented in this research.

5.4 Synthesis Result and Discussion

The N-body simulation was tested on Nallatech 385 accelerator which contains Stratix V A7 FPGA. The optimized CUDA and OpenCL implementation of N-body simulation was also tested on NVIDIA K620 GPU and Xeon E5-2637V3 CPU for performance comparison. The result is summarized in Table 10. From the table we can see that the while the FPGA could outperform the optimized CPU implementation, it could only compete against GPU with small data size. The GPU can easily outperform both FPGA and CPU when data size is sufficiently large.

Table 10. N-body Simulation Performance Result in Term of Throughput

# of Bodies	Performance (GFLOPS)		
	FPGA	GPU	CPU
256	1.88	0.1	2.59
512	12.40	0.3	7.48
1024	30.19	1.3	23.18
2048	53.68	5	45.07
4096	81.10	20.1	63.76
8192	94.06	80.6	73.19
16384	98.74	322.3	76.24
32768	99.82	429.7	77.31

When the kernel is implemented to use single precision floating point operations, there are 9 add/sub operations, 9 multiply operations, and one inverse square root operation for every force calculation. The total DSP utilization is 9+2 = 11 DSPs. With 23 loop unrolling, 23 force calculations can be computed concurrently, which uses 253 DSP in total. When converting the single precision floating point kernel to use double precision operation, the resource utilization dramatically increased to 9*4+9 = 45 DPs per force calculation, and the maximum possible loop unroll factor is decreased to 5. Estimated performance is at least more than 4 times slower without considering the decreased clock speed.

When compiling N-body kernel targeting the Stratix V D5 FPGA with less logic resources than A7 but significantly more DSP blocks (1590 vs. 256 DSPs), twice the number of DSP block could be utilized before the FPGA runs out of logic resources. Since Stratix V D5 has similar power consumption rating as Stratrix V A7. It should be able to achieve 200 GFLOPS with about twice the power efficiency in term of performance per watt. However, this is still not competitive with GPUs even on GFLOPS per Watt terms, because GPUs easily achieve throughput in the range of TFLOP throughput with little more than 100W of power consumption.

The last generation Stratix V FPGA requires lots of logic resources beside the DSP units to implement floating point operations. Therefore even if there are plentiful DSP resources, the logic resources such as ALMs and Registers usually run out before the DSP resources could be fully utilized. The new generation Arria 10 and Stratix 10 utilizes hard DSP unit optimized for floating point operations which requires far less supplementary logic and could be clocked at much higher speed. They may offer multiple

orders of performance increase without increasing the power consumption. It is possible that they could offer similar performance when compared to their GPU counterparts at much lower power consumption.

Chapter 6

Acceleration of Matrix Decomposition

6.1 Introduction to Matrix Decomposition Algorithms

Many important engineering and machine learning applications today rely on matrix decomposition. Factoring large matrices is computationally intensive. Appropriate hardware acceleration can dramatically speed up the application.

Gauss–Jordan elimination is one of the oldest methods for solving matrices. It involves use of repeated elementary row operations such as scalar multiplication, addition and swapping of rows to reduce the matrix to upper triangular form. However Gauss-Jordan elimination for solving matrix is not fully optimized. In most engineering and science problems, the coefficient matrix often stays the same, whereas the constant vector constantly changes. In this case a method called LU decomposition can be used to simplify computation. LU decomposition is one of the ways to factorize a non-singular matrix A into the product of a lower triangular matrix L and an upper triangular matrix U, such that [A] = [L] x [U]. Solving system of linear equations using LU decomposition has time complexity similar to Gauss Jordan elimination, which is O (N^3) for N by N matrix. However, Gauss Jordan elimination method requires constantly performing forward elimination and back substitutions, whereas the LU decomposition computes those two steps separately, and thus the factorization of matrix into upper and lower triangular matrices needs only to be done once. When LU decomposition is complete, the solving part of the computation only costs O (N^2). The LU decomposition also allows fast computation of determinant, as the determinant is simply the product of diagonal

elements. Also, because the resultant L and U are both triangular matrices, they could be stored in-place in the matrix to be solved and save significant memory space.

There are a few different flavors of LU decomposition. Two of the most common sequential algorithms for performing LU decomposition are Doolittle's method and Crout's method. The Doolittle method uses Gauss elimination to decompose the matrix A into upper triangular U and lower triangular L matrix. In every iteration one Gauss elimination is performed on an increasingly smaller sub matrix, until only one element is left. The Crout's method on the other hand factorizes one row and one column at each time and then updates the rest of the matrix in each iteration.

To increase the computation density per memory access, the Blocked LU decompositions were developed, which divide the problems into smaller fixed sized blocks. Each iteration solve one blocked column or panel using LU decomposition and uses general matrix multiply (GEMM) to update the trailing matrices. When the size of the block can be fitted into high speed cache memory, it is possible to reuse it in the computation without having to update the lower speed global memory. There are a few different variant of blocked LU decomposition, namely Left-looking, Right-looking and Crout (not to be confused with sequential version of Crout method) Blocked LU decomposition [58]. The time complexity of the different methods in general is the same but they differ in memory access and order of executions.

The optimized LU decomposition routine has been built into many numerical computing libraries. One of the most popular ways of implementing matrix factorization and solving is by using LAPACK library [59]. The library includes a set of Basic Linear

Algebra Subprograms (BLAS) that are essential in matrix operation. There are 3 levels of BLAS operations. Level 1 BLAS deal with vector to vector operation, level 2 BLAS include vector to matrix operation, and level 3 BLAS are used to perform matrix to matrix operations. Due to the fact that modern computers are mostly limited by memory latency and throughput rather than the capability of performing arithmetic operations, performance of an algorithm can be increased when the number of arithmetic operations performed per memory access is high. The level 3 BLAS allows memory reuse by loading data into fast cache memory, and reusing in later computations. The algorithm for unblocked and blocked LU decomposition is shown in Figure 27.

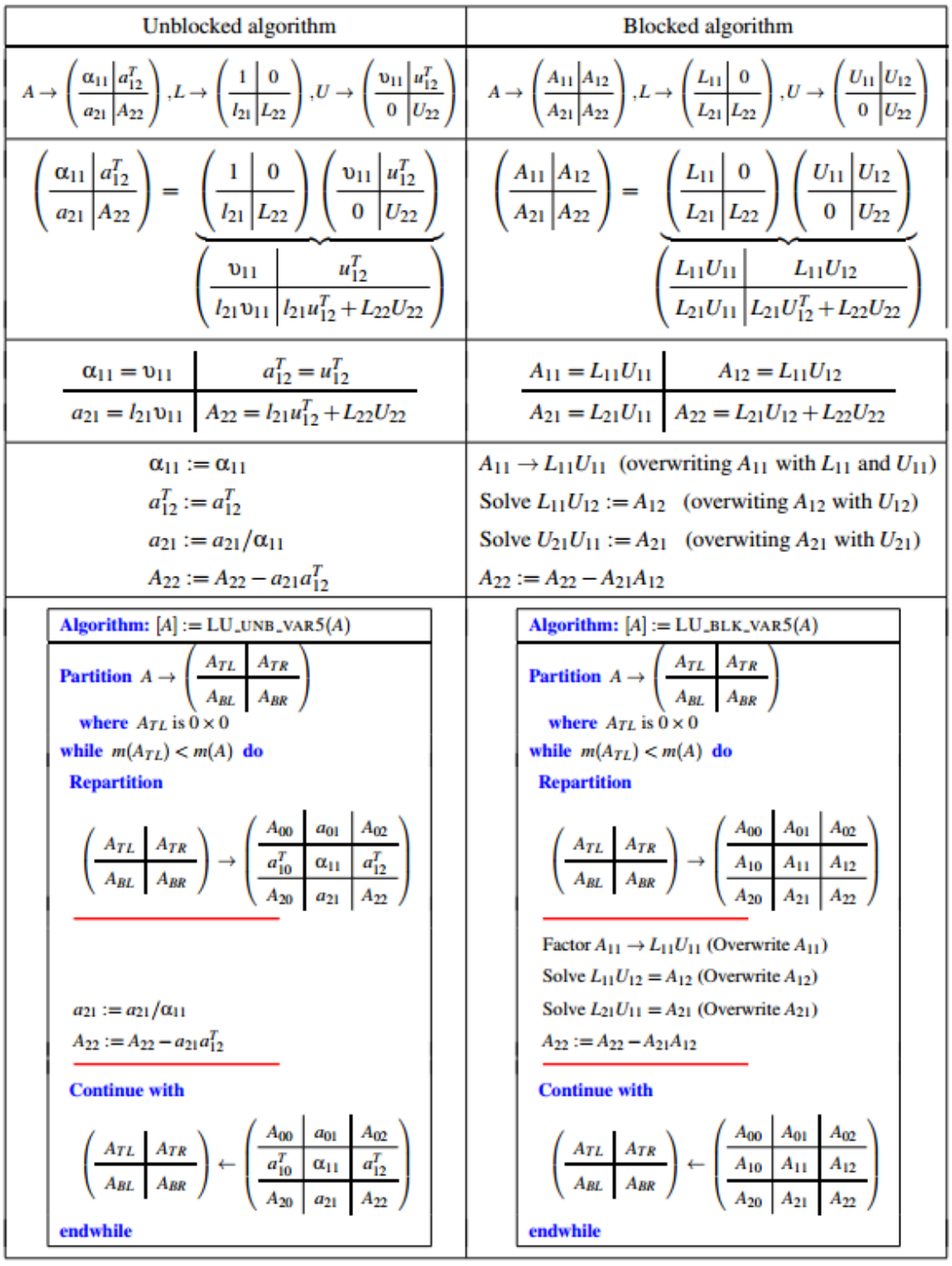

Figure 27. Unblocked and Blocked LU Decomposition Algorithm [60]

6.2 Related Works

Over the years, extensive Research has been done on accelerating LU decomposition on CPU, GPU and FPGA. Hardware vendors such as Intel, AMD, and NVIDIA have implemented those BLAS operations in their own math libraries such as Intel Math Kernel Library (MKL) [61] and NVIDIA cuBLAS library [62]. Those libraries are written in highly optimized assembly code and can achieve very high throughput on high performance GPUs and CPUs. Due to the advancements made in the field of heterogeneous computing, modern version of linear algebra libraries such as Matrix Algebra on GPU and Multicore Architectures (MAGMA) [63] are developed to utilize both the power of massively parallel GPUs and multi core CPUs to solve matrix problems. The MAGMA library utilizes both MKL and cuBLAS library and is one of the fastest numerical library available.

Various FPGA implementations of LU decomposition also exist, but they are mostly implemented before the age of GPGPU computing, so their performance was not as competitive. One good example of LU decomposition implemented on FPGA [64] was published in 2008, in which the author was able to achieve 47GFLOPS in blocked LU on an older generation Stratix III FPGA with relatively small sized matrix by using hand coded Verilog. The dual core CPU the researcher used in comparison could achieve 42GFLOPS with MKL. Since Stratix III FPGA uses 18W of power while that CPU uses 80W, it was competitive against CPU in terms of power efficiency.

6.3 Altera OpenCL Implementation and Synthesis

In this research blocked LU decomposition was implemented using AOCL. The initial implementation used three single thread kernels, one for each of the LU

decomposition operations. The performance was poor due to the fact that very little memory reuse existed and the optimized pipeline could not be instantiated by the compiler. Various one and two dimensional multi-work-group implementations were later developed and optimized. Tests show that kernel that uses 2D work-groups performs the best, especially with SIMD vectorization enabled. But they are far from 100 percent efficient and they used too much local memory as local cache to store the block of matrix under computation. As a result not all DSPs could be utilized because the local memory resources were already depleted before work-group size, vector size and loop unroll factor could be further increased.

Because the double precision floating point operations are too costly for older generation Stratix V FPGAs only single precision floating point data was used. The best blocked LU decomposition AOCL implementation designed in this research uses the right-looking blocked algorithm. It uses 1D cache for LU, left kernels and no pivoting. The work-group size used in the kernel is 64 by 64. In this kernel, the factorization of the A_{TL} (see Figure 27) matrix is done using a small 2D threaded LU kernel. The process of updating the left panel of matrix A is done separately in two different kernels for A_{TL} and A_{BL} matrix. This way the process of updating matrix A_{TR} and A_{BL} in the upper and left panels could be done concurrently. The update of trailing matrix A_{BR} is done using a scaled down version of matrix multiplication kernel. In total four kernels are used for blocked LU decomposition.

6.4 Results and Discussion

The performance of the kernels was measured in terms of GFLOPS. The throughput for LU decomposition is calculated by dividing the total number of operations

by the execution time, which is equivalently $\frac{1.5*N^2 + 0.5*N^2 - N/6.0}{1e9*\text{time in seconds}}$, assuming the size of the matrix is N by N. In order to compare performance between FPGA and CPU/GPU, the appropriate CPU and GPU routines from MAGMA library were called to solve the matrixes with the same sizes. The MAGMA library was compiled with Intel parallel studio version 11.2 and CUDA SDK version 6.5 with optimizations enabled. The CPU used in the test was Intel Xeon E5-2637 v3 while the GPU used was NVIDIA K620. Randomly generated square matrices with size from 1024 to 16384 were used in the test, and the result is summarized in Table 11. The FPGA implementation was eclipsed by CPU and GPU in all the cases. The GPU outperforms the CPU for large matrix sizes. The CPU used in our test is one of the high end server CPUs with 4 cores and 12 MB of level 3 cache, whereas the GPU is a low power workstation graphics card. The newer and higher end GPUs should be able to achieve much higher throughput in the range of TFLOPS.

Table 11. Blocked LU Decomposition Throughput Performance Results

	Performance (GFLOPS)		
Matrix Sizes N	FPGA (DE5)	MKL (Xeon E5)	MAGMA GPU (K620)
1024	6.6829	114.87	82.39
2048	14.9069	213.84	199.94
4096	25.8361	253.76	325.22
8192	35.8658	317.19	446.55
16384	42.7429	328.1	545.38

Test run with 16384 matrix size is profiled, and the visualization of kernel execution is shown in Figure 28. In each kernel launch iteration, the trailing matrix update kernel takes the most time to finish, while the LU decomposition of the top left diagonal matrix takes the least amount of time. The update of the top and left panel

matrix is processed in parallel and takes comparatively less time to compute than trailing matrix update.

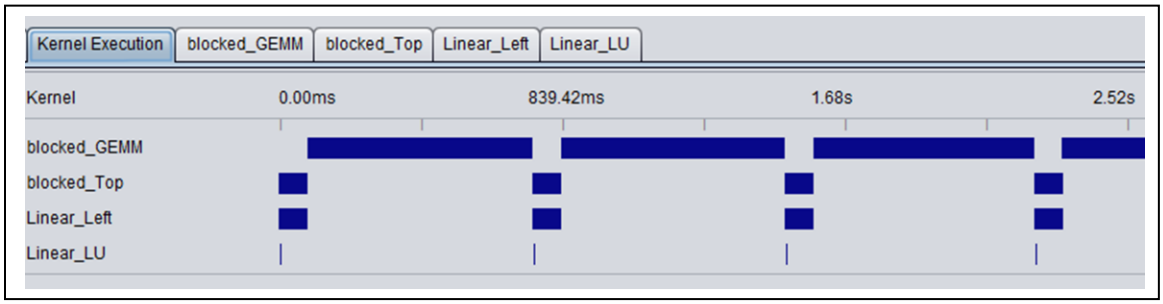

Figure 28. AOCL LU Decomposition Profile Result [60]

The FPGA resource utilization for Blocked LU decomposition kernel is shown in Table 12. The DE5-net accelerator used less I/O pins due to less memory on board, which means it could be compiled to a slightly faster frequency. The effect on performance is miniscule however. The kernel used 64x64 work-item 2D work-groups. While larger work-group sizes such as 80x80 are also possible, the memory block utilization was over 100 percent and could not fit on the Stratix V A7 FPGA.

Table 12. Resource Utilizations of Blocked LU Decomposition Kernel

	DE5-net	Nallatech 385
Logic utilization %	53	53
I/O pins %	26	58
DSP blocks %	72	72
Memory bits %	35	35
RAM blocks %	66	65
Kernel fmax (MHz)	202.47	194.7
Peak throughput (GFLOPS)	41.2	42.7

If the blocked LU decomposition kernel is compiled on newer generation FPGAs such as Arria 10 and Stratix 10, it is possible to achieve higher performance due to more

DSP and memory blocks are available. However it would still be difficult trying to compete with high end GPUs running fully optimized linear algebra libraries. Further research is needed to determine more efficient ways of implementing the matrix solvers on Altera SDK for OpenCL.

Chapter 7

Conclusion and Future Work

7.1 Summary

The AOCL implementations of k-means clustering and k-nearest neighbor algorithms synthesized for FPGA performed well against CPUs. However, they were only able to compete with GPUs in certain problem sizes. When power consumption is considered, the power efficiency of FPGA far out matched CPU implementations, and is equal or better when compared with GPU. In the case of N-body and LU decomposition, the AOCL implementation did not significantly outperform the optimized CPU implementation and is outperformed by GPU. However, it is important to note that CPU and GPU architecture has been optimized to handle those kinds of algorithms very well. Those applications may not be good representatives to illustrate of the full potential of FPGAs for hardware acceleration.

7.2 Evaluation of Altera SDK for OpenCL

The Altera SDK for OpenCL allows acceleration of algorithms on FPGA without extensive hardware knowledge. It exposes power of reconfigurable hardware to software engineers. At the same time when compared with other traditional HLS tools, it offers more streamlined development process and allows high performance heterogeneous computing across different platforms. The AOCL compiler always attempts to automatically generate the most efficient pipeline and memory structure for every kernel program, which means the applications that do not map to GPU architecture perfectly may perform better on FPGA with AOCL. Since FPGA usually has lower power profile,

when used in data centers the cooling and electricity cost could be greatly reduced. At the same time FPGAs could be packed more densely together to save space. The kernels developed on AOCL targeting FPGAs have a relative long life cycle, because it is often unnecessary to redesign the kernel for a newer generation FPGA.

The current generation Altera SDK for OpenCL also has a few drawbacks. The kernel compilation time is usually in the range of hours, which is exceptionally long comparing to CPU and GPU. The compilation also requires 32 GB of memory and a powerful CPU, which could be costly. Also, the current generation Stratix V FPGAs has lower peak floating point processing capability compared to GPUs. The Stratix V A7 FPGAs tested in this research have very limited number of DSP units for floating point processing. Also, implementation of floating point functions requires a large amount of supporting logic resources. Meanwhile the logic resources could not be shared between different kernels because AOCL compiler always ensures that the kernels can be executed concurrently, even when that is not necessary. Additionally, sometimes it is difficult to optimize for AOCL because it is less transparent compared to GPU. Since GPU has fixed architecture, developers simply needs to maximize the utilization of cores and memory bandwidth in order to achieve high performance. Whereas in the case of AOCL, the Altera offline compiler generates hardware architecture automatically based on kernel source code provided. Although it is possible to obtain the LLVM code and the Verilog source code generated by the compiler, they are very difficult to read or understand by a developer. Therefore, sometimes it is hard to determine what issue lowers performance. Lastly, the optimized kernels developed on AOCL for FPGA

usually use different methods to achieve parallelism than CPU or GPU, which means the portability of the OpenCL code is reduced.

Overall, the Altera SDK for OpenCL is a powerful high level synthesis tool. It will make FPGA a strong contender in the high performance computing arena.

7.3 Future Work

In this research, only brute-force versions of N-body simulation and k-nearest neighbors search were implemented. It would be interesting to implement tree based approximation methods for N-body simulation and k-nearest neighbors search using AOCL on FPGA and investigate what kind of performance could be achieved compared to CPU and GPU. For k-nearest neighbors search, better search algorithms could be implemented to enhance performance for larger cluster sizes. In addition, sparse matrix decomposition could be implemented using AOCL to determine if FPGA could achieve performance comparable to or higher than CPU and GPU.

Altera Corporation recently released new generation 10 FPGAs. The mid-range Arria 10 [65] and high-end Stratix 10 [66] FPGAs will be supported by Altera SDK for OpenCL. The new generation FPGAs uses hard floating point DSP units, which require far lower number of supplementary logic resources to implement floating point functions. The new generation FPGAs also supports ultra-high bandwidth Hybrid Memory Cube (HMC), which means they could have the same amount of global memory bandwidth as any high-end GPU. The new generation FPGAs are fabricated using state of the art Intel technology, which results in 2 to 5 times increase in logic density, and more than doubled the maximum frequency over current generation Stratix V. It would be interesting to

compile the kernels developed in this research to target Arria 10 and Stratix 10 and determine how much performance increase could be achieved.

References

[1] G. Moore, “Cramming more components onto integrated circuits,” Reprinted from Electronics, volume 38, number 8, April 19, 1965, pp.114 ff. in *Solid-State Circuits Society Newsletter, IEEE* , vol.11, no.5, pp.33-35, Sept. 2006

[2] A. Huang, “The Death of Moore’s Law Will Spur Innovation,” *Spectrum.ieee.org*, 2015. [Online]. Available: http://spectrum.ieee.org/semiconductors/design/the-death-of-moores-law-will-spur-innovation. [Accessed: 01- Nov- 2015].

[3] D. Patterson, “The trouble with multi-core,” in *Spectrum, IEEE* , vol.47, no.7, pp.28-32, 53, July 2010

[4] R. Dennard, V. Rideout, E. Bassous and A. LeBlanc, “Design of ion-implanted MOSFET's with very small physical dimensions,” in *Solid-State Circuits, IEEE Journal of* , vol.9, no.5, pp.256-268, Oct 1974

[5] D. Patterson, “The parallel computing landscape: a Berkeley view,” in *Low Power Electronics and Design (ISLPED), 2007 ACM/IEEE International Symposium on* , vol., no., pp.231-231, 27-29 Aug. 2007

[6] D. Kanter, “Inside Nehalem: Intel’s Future Processor and System,” 2008 [Online]. Available: http://www.realworldtech.com/nehalem/10/. [Accessed: 1- Dec- 2015]

[7] OpenMP.org, “The OpenMP API,” 2015 [Online]. Available: http://openmp.org/wp/. [Accessed: 1- Dec- 2015]

[8] B. Barney, "OpenMP," *Lawrence Livermore National Laboratory*, 2015. [Online]. Available: https://computing.llnl.gov/tutorials/openMP/. [Accessed: 01-Dec-2015].

[9] The Open MPI Project, "Open MPI: Open Source High Performance Computing," 2015 [Online]. Available: http://www.open-mpi.org/. [Accessed: 1- Dec- 2015]

[10] R. Smith, "NVIDIA GeForce GTX 680 Review: Retaking the Performance Crown," 2015. [Online]. Available: http://www.anandtech.com/print/5699/nvidia-geforce-gtx-680-review. [Accessed: 01-Dec-2015]

[11] Brown Deer Technology, "OpenCL™ Tutorial: N-Body Simulation," 2010. [Online]. Available: www.browndeertechnology.com/docs/BDT_OpenCL_Tutorial_NBody-rev3.html. [Accessed: 01-Dec-2015]

[12] Khronos Group, "The open standard for parallel programming of heterogeneous systems," 2015 [Online]. Available: https://www.khronos.org/opencl/. [Accessed: 1- Dec- 2015]

[13] Altera Corporation, "High-Performance FPGA Architecture," 2015. [Online]. Available: http://wl.altera.com/devices/fpga/stratix-fpgas/stratix-v/stxv-index.jsp. [Accessed: 01-Dec-2015]

[14] Altera Corporation, "Stratix V FPGAs: Built for Bandwidth," 2015. [Online]. Available: http://wl.altera.com/devices/fpga/stratix-fpgas/about/fpga-architecture/stx-architecture.html. [Accessed: 01-Dec-2015]

[15] Altera Corporation, "OpenCL Reference Platforms," [Online]. Available:http://www.altera.com/products/software/partners/opencl/opencl-board-partner-index.html. [Accessed: 01-Dec-2015]

[16] Altera Corporation, "OpenCL Development Kits and Cards," [Online]. Available: http://www.altera.com/products/devkits/opencl-index.jsp. [Accessed: 01-Dec-2015]

[17] Altera Corporation, "Stratix V FPGA Family Overview," [Online]. Available: http://www.altera.com/devices/fpga/stratix-fpgas/stratix-v/overview/stxv-overview.html. [Accessed: 01-Dec-2015]

[18] Terasic Technologies Inc., "DE5-Net FPGA Development Kit Specification," 2014. [Online]. Available: de5-net.terasic.com [Accessed: 01-Dec-2015]

[19] Nallatech, "Nallatech 385 – with Stratix V A7 FPGA," 2015. [Online]. Available: http://www.nallatech.com/store/fpga-accelerated-computing/pcie-accelerator-cards/385-a7/ [Accessed: 01-Dec-2015]

[20] A. Canis, "LegUp," 2015. [Online]. Available: http://legup.eecg.utoronto.ca/ [Accessed: 01-Dec-2015]

[21] Altera Corporation, "Altera SDK for OpenCL Overview," 2015. [Online]. Available: https://www.altera.com/products/design-software/embedded-software-developers/opencl/overview.html [Accessed: 10-Sept-2015]

[22] S. Settle, "High-performance Dynamic Programming on FPGAs with OpenCL," in *17th Ann. HPEC Conf.*, Waltham, MA USA, 2013. [Online]. Available:

http://ieee-hpec.org/2013/index_htm_files/29-High-Performance-Settle-2876089.pdf [Accessed: 01-Dec-2015]

[23] D. Bacon, R. Rabbah, S. Shukla, “FPGA Programming for the Masses,” 2013. [Online]. Available: http://delivery.acm.org/10.1145/2450000/2443836/p40-bacon.pdf [Accessed: 01-Dec-2015]

[24] T. S. Czajkowski et al., “OpenCL for FPGAs: Prototyping a Compiler,” in *Inter. Conf. Reconfigurable Systems and Algorithms,* 2012. [Online]. Available: http://ersaconf.org/ersa12/papers/Brown-opencl-for-fpgas.pdf [Accessed: 01-Dec-2015]

[25] Altera Corporation, “Implementing FPGA design with the OpenCL standard,” November 2013. [Online]. Available: http://www.altera.com/literature/wp/wp-01173-opencl.pdf [Accessed: 01-Dec-2015]

[26] Altera Corporation, “Altera SDK for OpenCL Programming Guide,” 2015. [Online]. Available: http://www.altera.com/literature/hb/opencl-sdk/aocl_programming_guide.pdf [Accessed: 01-Dec-2015]

[27] C. Cao, et al. "clMagma," in *Proceedings of the International Workshop on OpenCL 2013 & 2014 (IWOCL '14)*. 2014.

[28] Altera Corporation, “Altera SDK for OpenCL Best Practices Guide,” 2015. [Online]. Available: http://www.altera.com/literature/hb/opencl-sdk/aocl_optimization_guide.pdf [Accessed: 01-Dec-2015]

[29] X. Wu, V. Kumar, J. Quinlan, J. Ghosh, Q. Yang, H. Motoda, A. McLachlan, A. Ng, B. Liu, Z. Zhou, M. Steinbach, D. Hand, and D. Steinberg, "Top 10 algorithms in data mining," in *Knowledge and Information Systems*, 14, 1, 1-37, 2007.

[30] D. Arthur and S. Vassilvitskii. "k-means++: the advantages of careful seeding," in *Proceedings of the eighteenth annual ACM-SIAM symposium on Discrete algorithms (SODA '07)*. Society for Industrial and Applied Mathematics, Philadelphia, PA, USA, 1027-1035, 2007.

[31] R. Narayanan, B. Ozisikyilmaz, J. Zambreno, G. Memik, and A. Choudhary, "MineBench: A Benchmark Suite for Data Mining Workloads," in *Workload Characterization, 2006 IEEE International Symposium on*, Oct. 182-188, 25-27, 2006.

[32] S. Che, M. Boyer, J. Meng, D. Tarjan, J. Sheaffer, and K. Skadron, "A performance study of general-purpose applications on graphics processors using CUDA," in *Journal of Parallel and Distributed Computing*, 68, 10, 1370-1380, 2008.

[33] L. Jian, C. Wang, Y. Liu, S. Liang, W. Yi, and Y. Shi, "Parallel data mining techniques on Graphics Processing Unit with Compute Unified Device Architecture (CUDA)," in *J Supercomput*, 2011, 64, 3, 942-967.

[34] R. Wu, B. Zhang and M. Hsu, "Clustering billions of data points using GPUs," in *Proceedings of the combined workshops on UnConventional high performance computing workshop plus memory access workshop (UCHPC-MAW '09)*, 2009.

[35] Y. Li, K. Zhao, X. Chu and J. Liu, "Speeding up K-Means Algorithm by GPUs," in *2010 10th IEEE International Conference on Computer and Information Technology*, 2010.

[36] B. Dhanasekaran and N. Rubin, "A new method for GPU based irregular reductions and its application to k-means clustering," in *Proceedings of the Fourth Workshop on General Purpose Processing on Graphics Processing Units (GPGPU-4)*, ACM, New York, NY, USA, 2011.

[37] Y. Choi, and H. So, "Map-reduce processing of k-means algorithm with FPGA-accelerated computer cluster," in *2014 IEEE 25th International Conference on Application-Specific Systems, Architectures and Processors*, 2014.

[38] O. Segal, M. Margala, S. R. Chalamalasetti and M. Wright, "High Level Programming for Heterogeneous Architectures," in *First International Workshop on FPGAs for Software Programmers (FSP 2014)*, 2014.

[39] ThinkTank Energy Products Inc., "Watt's Up Pro Power metter specifications," [Online]. Available: https://www.wattsupmeters.com/secure/products.php?pn=0&wai=276&spec=4 [Accessed: 10-Sept-2015]

[40] S. Arya, D. Mount, N. Netanyahu, R. Silverman, and A. Wu, "An optimal algorithm for approximate nearest neighbor searching fixed dimensions," in *Journal of the ACM*, vol. 45, no. 6, pp. 891-923, 1998.

[41] D. Mount and S. Arya, "ANN: A Library for Approximate Nearest Neighbor Searching," Jan 27, 2010. [Online]. Available: https://www.cs.umd.edu/~mount/ANN/ [Accessed: 31- Dec- 2015]

[42] V. Garcia, E. Debreuve and M. Barlaud, "Fast k nearest neighbor search using GPU," in *IEEE Computer Society Conference on Computer Vision and Pattern Recognition Workshops*, 2008.

[43] V. Garcia, E. Debreuve, F. Nielsen and M. Barlaud, "K-nearest neighbor search: Fast GPU-based implementations and application to high-dimensional feature matching," in *IEEE International Conference on Image Processing*, 2010.

[44] V. Garcia, E. Debreuve and M. Barlaud, "kNN CUDA," [Online]. Available: http://vincentfpgarcia.github.io/kNN-CUDA/ [Accessed: 31- Dec- 2015]

[45] N. Sismanis, N. Pitsianis and X. Sun, "Parallel search of k-nearest neighbors with synchronous operations," in *IEEE Conference on High Performance Extreme Computing*, 2012.

[46] H. Hussain, K. Benkrid and H. Seker, "An adaptive implementation of a dynamically reconfigurable K-nearest neighbour classifier on FPGA." in *NASA/ESA Conference on Adaptive Hardware and Systems (AHS)*, 2012.

[47] I. Stamoulias and E. Manolakos, "Parallel architectures for the kNN classifier -- design of soft IP cores and FPGA implementations," in *ACM Transactions on Embedded Computing (TECS)*, vol. 13, no. 2, pp. 1-21, 2013.

[48] I. Komarov, A. Dashti and R. D'Souza, “Fast k-NNG Construction with GPU-Based Quick Multi-Select.” in *PLoS ONE*, vol. 9, no. 5, p. e92409, 2014.

[49] Altera.com, “OpenCL Design Examples,” 2016. [Online]. Available: https://www.altera.com/support/support-resources/design-examples/design-software/opencl.html. [Accessed: 3- Jan- 2016].

[50] Algolist.net, “INSERTION SORT (Java, C++) Algorithms and Data Structures,” 2016. [Online]. Available: http://www.algolist.net/Algorithms/Sorting/Insertion_sort. [Accessed: 04- Jan- 2016].

[51] Personal.kent.edu, “Heap Sort,” 2016. [Online]. Available: http://www.personal.kent.edu/~rmuhamma/Algorithms/MyAlgorithms/Sorting/heapSort.htm. [Accessed: 04- Jan- 2016].

[52] C. Horstmann and T. Budd, *Big C++*. Hoboken, N.J.: Wiley, 2009.

[53] G. Heineman, S. Selkow and G. Pollice, *Algorithms in a nutshell*. Sebastopol, Calif.: O'Reilly, 2009.

[54] J. Barnes and P. Hut, “A hierarchical O(N log N) force-calculation algorithm,” in *Nature*, vol. 324, no. 6096, pp. 446-449, 1986.

[55] developer.nvidia.com, “GPU Gems 3 - Chapter 31. Fast N-Body Simulation with CUDA,” 2016. [Online]. Available: http://http.developer.nvidia.com/GPUGems3/gpugems3_ch31.html. [Accessed: 05- Jan- 2016].

[56] Intel Developer Zone, “Cross-Device NBody Simulation Sample,” 2014. [Online]. Available: https://software.intel.com/articles/opencl-cross-devices-nbody-simulation-sample. [Accessed: 05- Jan- 2016].

[57] T. Hamada, K. Benkrid, K. Nitadori and M. Taiji, “A Comparative Study on ASIC, FPGAs, GPUs and General Purpose Processors in the O(N^2) Gravitational N-body Simulation,” in *NASA/ESA Conference on Adaptive Hardware and Systems*, 2009.

[58] J. Dongarra, *Numerical linear algebra for high-performance computers*. Philadelphia: Society for Industrial and Applied Mathematics, 1998.

[59] Netlib.org, “LAPACK — Linear Algebra PACKage,” 2016. [Online]. Available: http://www.netlib.org/lapack/. [Accessed: 04- Jan- 2016].

[60] M. Myers, P. Geijn and R. Geijin, *Linear Algebra: Foundations to Frontiers*, 2014, p. 266.

[61] Intel Corporation, “Intel® Math Kernel Library (Intel® MKL),” [Online]. Available: https://software.intel.com/en-us/intel-mkl [Accessed: 04- Jan- 2016].

[62] NVIDIA Corporation, “cuBLAS,” [Online]. Available: https://developer.nvidia.com/cuBLAS [Accessed: 04- Jan- 2016].

[63] Innovative Computing Laboratory, “Matrix Algebra on GPU and Multicore Architecture,” [Online]. Available: http://icl.utk.edu/magma/

[64] Wei Zhang, “Portable and Scalable FPGA-Based Acceleration of a Direct Linear System Solver,” MASc thesis, University of Toronto, 2008

[65] Altera.com, “Arria 10 - Overview,” 2016. [Online]. Available: https://www.altera.com/products/fpga/arria-series/arria-10/overview.html. [Accessed: 05- Jan- 2016].

[66] Altera.com, “Stratix 10 - Overview,” 2016. [Online]. Available: https://www.altera.com/products/fpga/stratix-series/stratix-10/overview.html. [Accessed: 05- Jan- 2016].

Appendices

Appendix A: AOCL Reduction Sum Kernel Source Code

```
#define RELAX_FACTOR  6
#define UNROLL_FACTOR 32
#define CU_SIZE 1
#define SIMD_SIZE 1
#define WORK_SIZE 1

#pragma OPENCL EXTENSION cl_altera_channels : enable
channel float part_sum_ch12 __attribute__((depth(256)));

__attribute((reqd_work_group_size(WORK_SIZE,1,1)))
__attribute((num_simd_work_items(SIMD_SIZE)))
__attribute((num_compute_units(CU_SIZE)))
__kernel void reduction_add( __global const float * restrict a, // input array
                             const unsigned LOOP_DEPTH // elements per thread )
{
    // total threads = N/LOOP_DEPTH
    unsigned start_id = get_global_id(0)*LOOP_DEPTH;

    float result_tmp = 0.0f;
    #pragma unroll UNROLL_FACTOR
    for (unsigned i=0; i<LOOP_DEPTH; i++){
        result_tmp += a[start_id+i];
    }

    write_channel_altera(part_sum_ch12, result_tmp);
}

// Final reduction stage using task based kernel
__kernel void reduction_final( const unsigned n,
                               global float * restrict result )
{
    float local_result = 0.0f;
    float copies[RELAX_FACTOR];

    // Initiate the replicated memory
    for(unsigned i=0; i<RELAX_FACTOR; i++){
        copies[i] = 0.0f;
    }

    // Relaxed summation
    for (unsigned i=0; i<n; i++) {
        float cur = copies[RELAX_FACTOR-1] + read_channel_altera(part_sum_ch12);
        #pragma unroll
        for (unsigned j = RELAX_FACTOR-1; j>0; j--){
            copies[j] = copies[j-1];
        }
        copies[0] = cur;
    }

    // Final reduction
    for (unsigned i=0; i<RELAX_FACTOR; i++) {
        local_result += copies[i];
    }

    *result = local_result;
}
```

Appendix B: AOCL K-Means Kernel Source Code (64 Features version)

```
#ifndef MAX_NUM_CLUSTERS
#define MAX_NUM_CLUSTERS 512
#endif

#ifndef NUM_FEATURES
#define NUM_FEATURES 64
#endif

#define BLOCK_SIZE MAX_NUM_CLUSTERS*NUM_FEATURES
#define THRESHOLD 0.001f

#pragma OPENCL EXTENSION cl_altera_channels : enable

// The Channel allows both kernels to be executed concurrently.
channel unsigned short members_ch12 __attribute__((depth(16)));
channel bool change_ch12 __attribute__((depth(16)));

// Kernel with channels cannot use multi SIMD or Compute Units.
// Allows flexible number of clusters,
// but cluster size could not exceed MAX_NUM_CLUSTERS.
__attribute__((max_work_group_size(BLOCK_SIZE)))
__kernel void kmeans_assign(__global const float * restrict objects_g,
            __global float * restrict clusters_g,
            __global unsigned short * restrict membership_g,
            const unsigned num_clusters )
{
    float objects_l[NUM_FEATURES];

    // Full Local Cache of the cluster array
    __local float clusters_l[MAX_NUM_CLUSTERS*NUM_FEATURES];

    unsigned gid = get_global_id(0);
    unsigned lid = get_local_id(0);

    // Load clusters (once per-work-group)
    clusters_l[lid] = clusters_g[lid];

    // Make sure cluster is loaded before rest of the computation.
    barrier(CLK_LOCAL_MEM_FENCE);

    // Load one object per thread.
    #pragma unroll 4
    for (unsigned j=0; j<NUM_FEATURES; j++) {
        objects_l[j] = objects_g[gid*NUM_FEATURES + j];
    }

    unsigned short index;
    float min_dist = INFINITY;
    #pragma unroll 3 //7
    for (unsigned k=0; k<num_clusters; k++) {
        float dist = 0.0f;
        #pragma unroll NUM_FEATURES
        for (unsigned j=0; j<NUM_FEATURES; j++){
```

```
            dist += (objects_l[j]-clusters_l[k*NUM_FEATURES+j])
                    *(objects_l[j]-clusters_l[k*NUM_FEATURES+j]);
        }
        if (*(unsigned*)&dist < *(unsigned*)&min_dist) {
            min_dist = dist;
            index    = k;
        }
    }

    bool changed = 0;
    if (membership_g[gid]!=index){
        changed = 1;
    }

    write_channel_altera(members_ch12, index);
    write_channel_altera(change_ch12, changed);

}

__kernel
void kmeans_reduce( __global const float * restrict objects_g,
                    const unsigned num_objects,
                    __global unsigned short * restrict membership_g
                    __global float * restrict clusters_g,
                    __global float * restrict delta_g,
                    const unsigned num_clusters )
{
    unsigned cluster_size[MAX_NUM_CLUSTERS];
    float clusters_l[MAX_NUM_CLUSTERS][NUM_FEATURES];

    // Pre-load zeros.
    #pragma unroll 1 // Prevent automatic unroll to same resources
    for (unsigned i=0; i<num_clusters; i++){
        cluster_size[i] = 0;
        #pragma unroll 1
        for (unsigned j=0; j<NUM_FEATURES; j++){
            clusters_l[i][j] = 0.0f;
        }
    }

    // Sum objects to local cluster array.
    float delta = 0.0f;
    for (unsigned i=0; i<num_objects; i++) {

        // Load membership from channel.
        unsigned short index = read_channel_altera(members_ch12);
        delta += read_channel_altera(change_ch12);

        // Make sure index is loaded before write to global memory.
        mem_fence(CLK_CHANNEL_MEM_FENCE);

        // Write to global memory here instead of in the first kernel.
        // This relieves some of the global memory access latencies.
        membership_g[i] = index;
        cluster_size[index] += 1;
```

```
        #pragma unroll NUM_FEATURES
        for (unsigned j=0; j<NUM_FEATURES; j++) {
            clusters_l[index][j] += objects_g[i*NUM_FEATURES + j];
        }
    }

    *delta_g = delta;

    // Write back to global memory.
    #pragma unroll 1
    for (unsigned i=0; i< num_clusters; i++){
        unsigned cluster_sz_tmp = cluster_size[i];
        #pragma unroll 1
        for (unsigned j=0; j< NUM_FEATURES; j++){

            // Only move a centroid if it has members.
            if (cluster_sz_tmp > 0){

                // Write back to global memory.
                clusters_g[i*NUM_FEATURES + j] = clusters_l[i][j]/cluster_sz_tmp;

            }
        }
    }
}
```

Appendix C: AOCL K-Nearest Neighbor Kernel Source Code (Heap Sort Version)

```
#ifndef MAX_NUM_CLUSTERS
#define MAX_NUM_CLUSTERS 4
#endif

#ifndef BLOCK_SIZE_DIST
#define BLOCK_SIZE_DIST 128
#endif

#ifndef BLOCK_SIZE_SORT
#define BLOCK_SIZE_SORT 128
#endif

#ifndef SIMD_WORK_ITEMS
#define SIMD_WORK_ITEMS_DIST 2
#endif

#ifndef SIMD_WORK_ITEMS
#define SIMD_WORK_ITEMS_SORT 1
#endif

__attribute__((reqd_work_group_size(BLOCK_SIZE_DIST,BLOCK_SIZE_DIST,1)))
__attribute((num_simd_work_items(SIMD_WORK_ITEMS_DIST)))
__kernel void knn_dist( __global const float * restrict query_g,
                        __global float * restrict reference_g,
                        __global float * restrict dist_g,
                        const unsigned num_clusters,
                        const unsigned num_reference )
{
    // Cache 1 block of query and reference points per work-group.
    __local float query_l[BLOCK_SIZE_DIST][BLOCK_SIZE_DIST];
    __local float reference_l[BLOCK_SIZE_DIST][BLOCK_SIZE_DIST];

    unsigned gid_x = get_global_id(0);
    unsigned gid_y = get_global_id(1);

    unsigned lid_x = get_local_id(0);
    unsigned lid_y = get_local_id(1);

    unsigned group_id_x = get_group_id(0);
    unsigned group_id_y = get_group_id(1);

    query_l[lid_y][lid_x] = query_g[BLOCK_SIZE_DIST*BLOCK_SIZE_DIST*group_id_y
                                    + BLOCK_SIZE_DIST*lid_y + lid_x];
    reference_l[lid_y][lid_x] =
    reference_g[BLOCK_SIZE_DIST*BLOCK_SIZE_DIST*group_id_x
                + BLOCK_SIZE_DIST*lid_y + lid_x];

    barrier(CLK_LOCAL_MEM_FENCE);

    float dist = 0.0f;
    #pragma unroll BLOCK_SIZE_DIST
    for (unsigned j=0; j<BLOCK_SIZE_DIST; j++){
        dist += (query_l[lid_y][j]-reference_l[lid_x][j])
```

```
                *(query_l[lid_y][j]-reference_l[lid_x][j]);
    }

    // Global memory write with optimized memory efficiency.
    dist_g[gid_y*get_global_size(0) + gid_x] = dist;

}

__kernel
__attribute__((reqd_work_group_size(BLOCK_SIZE_SORT,1,1)))
__attribute((num_simd_work_items(SIMD_WORK_ITEMS_SORT)))
void knn_sort(  __global float * restrict dist_g,
                __global unsigned * restrict index_g,
                const unsigned num_clusters,
                const unsigned num_reference )
{
    __local float dist_block[BLOCK_SIZE_SORT][BLOCK_SIZE_SORT];
    float k_dist_l[MAX_NUM_CLUSTERS+1];
    unsigned k_index_l[MAX_NUM_CLUSTERS+1];

    unsigned gid = get_global_id(0);
    unsigned lid = get_local_id(0);
    unsigned group_id = get_group_id(0);

    #pragma unroll 1
    for (unsigned i=0; i<num_reference; i+=BLOCK_SIZE_SORT){

        barrier(CLK_LOCAL_MEM_FENCE);

        #pragma unroll 4
        for (unsigned j=0; j<BLOCK_SIZE_SORT; j++){
            dist_block[lid][j] = dist_g[BLOCK_SIZE_SORT*num_reference*group_id
                                        + i + j*num_reference + lid];
        }

        barrier(CLK_LOCAL_MEM_FENCE);

        for (unsigned j=0; j<BLOCK_SIZE_SORT; j++){
            float dist_new = dist_block[j][lid];
            unsigned idx = i+j;

            /* Modified Version of Heap Sort */
            // Use first k entries to build max heap.
            if (idx < MAX_NUM_CLUSTERS){
                unsigned index = idx+1;
                while (index > 1 ){
                    unsigned parent_idx = index>>1;
                    float parent = k_dist_l[index>>1];
                    unsigned parent_dist_idx = k_index_l[index>>1];
                    if(parent >= dist_new){ break; }
                    k_dist_l[index] = parent;
                    k_index_l[index] = parent_dist_idx;
                    index = parent_idx;
                }
                // Write new element to vacant spot.
```

```
                k_dist_l[index] = dist_new;
                k_index_l[index] = idx;
            }
            // Insert the dist_new as root if it is smaller than current root.
            else if (dist_new < k_dist_l[1]){
                unsigned index = 1;
                while (index <= MAX_NUM_CLUSTERS){
                    unsigned child_idx_l = index<<1;
                    unsigned child_idx;        // store temp child local index
                    float child_val;           // store temp child value
                    unsigned child_dist_idx;   // store temp child global index
                    if (child_idx_l <= MAX_NUM_CLUSTERS){
                        // Find the larger child.
                        unsigned child_idx_r = child_idx_l + 1;
                        if (child_idx_r <= MAX_NUM_CLUSTERS &&
                            k_dist_l[child_idx_r] > k_dist_l[child_idx_l]){
                                child_idx = child_idx_r;
                                child_val = k_dist_l[child_idx_r];
                                child_dist_idx = k_index_l[child_idx_r];
                        }else{
                           // Load left child if there is only one child left.
                            child_idx = child_idx_l;
                            child_val = k_dist_l[child_idx_l];
                            child_dist_idx = k_index_l[child_idx_l];
                        }
                        if(child_val > dist_new){
                            // Swap if larger child is larger than root.
                            k_dist_l[index] = child_val;
                            k_index_l[index] = child_dist_idx;
                            index = child_idx;
                        }else{
                            break; // Stop when no child is left.
                        }
                    }else{
                        break;
                    }
                }
                // Write new element to vacant spot.
                k_dist_l[index] = dist_new;
                k_index_l[index] = idx;
            }
        }
    }

    // Write clusters back to global memory.
    #pragma unroll 4
    for (unsigned i=0; i<MAX_NUM_CLUSTERS; i++){
        index_g[ gid*MAX_NUM_CLUSTERS + i ] = k_index_l[i+1];
    }
}
```

Appendix D: AOCL N-Body Kernel Source Code

```
// Used %99 of DSPs on Stratix V A7, but only around 50% of logic.
#define BLOCK_SIZE 23
#define UNROLL_FACTOR 23
#define NUM_SIMD    1
#define NUM_CU      1

// Use union to save some DSP units in position update.
typedef union array4_t {
    float4 vect;
    float array[4];
} array4;

__attribute((reqd_work_group_size(BLOCK_SIZE,1,1)))
__attribute((num_simd_work_items(NUM_SIMD)))
__attribute((num_compute_units(NUM_CU)))
__kernel void
NBody(  __global const float4 * restrict data_pos,
        __global const float4 * restrict data_v,
        __global float4 * restrict out_pos,
        __global float4 * restrict out_v,
        const unsigned int num_bodies,
        const float t_delta,
        const float half_t_delta_sqr,
        const float eps_sqr  )
{
    __local float4 pos_buffer[BLOCK_SIZE];

    int global_x = get_global_id(0);
    int local_x = get_local_id(0);

    array4 acc_i;
    acc_i.vect = (float4)0.0f;

    array4 body_i;
    body_i.vect  = data_pos[global_x];

    #pragma unroll 1 //Unrolling of this loop is not efficient.
    for ( unsigned tile_offset = 0; tile_offset < num_bodies;
          tile_offset+=BLOCK_SIZE ){

        // Cache 1 block of data to local memory.
        pos_buffer[local_x] = data_pos[tile_offset + local_x];

        barrier(CLK_LOCAL_MEM_FENCE);

        // Perform calculations on the Block.
        #pragma unroll UNROLL_FACTOR
        for (unsigned i = 0; i < BLOCK_SIZE; ++i ){

            float4 body_j = pos_buffer[i];
            float4 dist;

            // Maintain similar structures for vector type
```

```
            // as suggested by AOCL best practice guide.
            dist.x = body_j.x - body_i.vect.x;
            dist.y = body_j.y - body_i.vect.y;
            dist.z = body_j.z - body_i.vect.z;
            dist.w = 0.0f;

            float sqr_dist = dist.x*dist.x + dist.y*dist.y + dist.z*dist.z;

            float inv_dist = rsqrt(sqr_dist+eps_sqr);

            // Store mass in body.w.
            float s = (body_j.w * inv_dist) * (inv_dist * inv_dist);

            acc_i.vect.x += dist.x*s;
            acc_i.vect.y += dist.y*s;
            acc_i.vect.z += dist.z*s;
            acc_i.vect.w = 0.0f;
        }

        barrier(CLK_LOCAL_MEM_FENCE);

    }

    // Velocity and position update, tied to use as little hardware as possible.
    array4 v_i;
    v_i.vect = data_v[global_x];
    #pragma unroll 1 //Unrolling of this loop wastes FPGA area
    for (unsigned i=0; i<4; i++){
        v_i.array[i] += acc_i.array[i] * t_delta;
        body_i.array[i] += v_i.array[i] * t_delta
                         + acc_i.array[i] * half_t_delta_sqr;
    }
    out_v[global_x] = v_i.vect;
    out_pos[global_x] = body_i.vect;

}
```

Appendix E: AOCL Blocked LU decomposition Kernel Source Code

```
// Requires (N/BLOCK_SIZE) kernel launches to solve one matrix.
// Matrix Size = N x N; must be divisible by BLOCK_SIZE.
#define BLOCK_SIZE 64
#define SIMD_WORK_ITEMS 2

// Kernel 1: Linear BLAS 1/2 version of LU decomposition
__attribute((reqd_work_group_size(BLOCK_SIZE,BLOCK_SIZE,1)))
__kernel void
Linear_LU( __global float * restrict A, int N, int iter ){
    __local float col_buffer[BLOCK_SIZE];
    __local float row_buffer[BLOCK_SIZE];
    int local_x = get_local_id(0);
    int local_y = get_local_id(1);
    const int offset = (N + 1)*iter*BLOCK_SIZE;

    __local float pivot;
    float sum = A[local_x + local_y*N + offset];
    barrier(CLK_LOCAL_MEM_FENCE);

    for(int k=0; k<BLOCK_SIZE-1; ++k){
        if (local_x == k && local_y == k){
            pivot = sum;
        }
        if (local_x > k && local_y == k){
            row_buffer[local_x] = sum;
        }
        if (local_x == k && local_y > k){
            sum /= pivot;
            col_buffer[local_y] = sum;
        }
        if (local_x > k && local_y > k){
            sum -=  row_buffer[local_x] * col_buffer[local_y];
        }
        barrier(CLK_LOCAL_MEM_FENCE);
    }

    A[local_x + local_y*N + offset] = sum;
}

// Kernel 2: Linear BLAS 1/2 version of Left panel update
__attribute((reqd_work_group_size(BLOCK_SIZE,BLOCK_SIZE,1)))
__attribute((num_simd_work_items(SIMD_WORK_ITEMS*2)))
__kernel void Linear_Left( __global float * restrict A, int N, int iter ){
    // iter = current iteration number starts from zero
    __local float U_Block [BLOCK_SIZE][BLOCK_SIZE];
    __local float A_Block [BLOCK_SIZE];
    int local_x = get_local_id(0);
    int local_y = get_local_id(1);
    const int offset_U = (N + 1)*iter*BLOCK_SIZE;
    int offset_A = offset_U + N*BLOCK_SIZE*(get_group_id(1)+1);

    U_Block[local_y][local_x] = A[local_x + local_y*N + offset_U];
    barrier(CLK_LOCAL_MEM_FENCE);
```

```
    float sum = A[local_x + local_y*N + offset_A];
    barrier(CLK_LOCAL_MEM_FENCE);

    for(int k=0; k<BLOCK_SIZE; ++k){
        if(local_x == k){
            sum /= U_Block[k][k];
            A_Block[local_y] = sum;
        }
        if (local_x > k){
            sum -= A_Block[local_y] * U_Block[k][local_x];
        }
        barrier(CLK_LOCAL_MEM_FENCE);
    }

    A[local_x + local_y*N + offset_A] = sum;
}

// Kernel 3: BLAS 3 version top panel update
__attribute((reqd_work_group_size(BLOCK_SIZE,BLOCK_SIZE,1)))
__attribute((num_simd_work_items(SIMD_WORK_ITEMS*2)))
__kernel void blocked_Top( __global float * restrict A, int N, int iter ){

    __local float L_Block [BLOCK_SIZE][BLOCK_SIZE];
    __local float A_Block [BLOCK_SIZE][BLOCK_SIZE];

    int local_x = get_local_id(0);
    int local_y = get_local_id(1);

    int const offset_L = (N+1)*(iter)*BLOCK_SIZE;
    int offset_A = offset_L + (get_group_id(0)+1)*BLOCK_SIZE;

    L_Block[local_y][local_x] = A[offset_L + N * local_y + local_x];
    barrier(CLK_LOCAL_MEM_FENCE);

    A_Block[local_y][local_x] = A[offset_A + N * local_y + local_x];
    barrier(CLK_LOCAL_MEM_FENCE);

    for(int k=0; k<BLOCK_SIZE; ++k){
        if (local_y>k){
            A_Block[local_y][local_x] -= L_Block[local_y][k] *
                                         A_Block[k][local_x];
        }
    }
    barrier(CLK_LOCAL_MEM_FENCE);

    A[offset_A + N * local_y + local_x] = A_Block[local_y][local_x];

}

// Kernel 4: BLAS 3 Trailing matrix update (GEMM)
__attribute((reqd_work_group_size(BLOCK_SIZE,BLOCK_SIZE,1)))
__attribute((num_simd_work_items(SIMD_WORK_ITEMS)))
__kernel void blocked_GEMM( __global float * restrict A, int N, int iter ){
```

```
    __local float L_Block [BLOCK_SIZE][BLOCK_SIZE];
    __local float U_Block [BLOCK_SIZE][BLOCK_SIZE];

    int local_x = get_local_id(0);
    int local_y = get_local_id(1);

    int const offset = (N+1)*(iter)*BLOCK_SIZE;
    int offset_L = offset + (get_group_id(1)+1)*BLOCK_SIZE*N;
    int offset_U = offset + (get_group_id(0)+1)*BLOCK_SIZE;
    int offset_A = offset + (get_group_id(1)+1)*BLOCK_SIZE*N
                   + (get_group_id(0)+1)*BLOCK_SIZE;

    float sum = A[offset_A + N * local_y + local_x];
    barrier(CLK_LOCAL_MEM_FENCE);

    L_Block[local_y][local_x] = A[offset_L + N * local_y + local_x];
    barrier(CLK_LOCAL_MEM_FENCE);

    U_Block[local_x][local_y] = A[offset_U + N * local_y + local_x];
    barrier(CLK_LOCAL_MEM_FENCE);

    #pragma unroll
    for(int k=0; k<BLOCK_SIZE; ++k){
        sum -= L_Block[local_y][k] * U_Block[local_x][k];
    }
    barrier(CLK_LOCAL_MEM_FENCE);

    A[offset_A + N * local_y + local_x] = sum;
}
```

Vita Auctoris

NAME:	Qing Yun Tang
PLACE OF BIRTH:	Beijing, China
YEAR OF BIRTH:	1989
EDUCATION:	Vincent Massey Secondary School, Windsor, ON, 2009
	University of Windsor, B.Sc., Windsor, ON, 2013
	University of Windsor, M.Sc., Windsor, ON, 2015