University of Windsor
Scholarship at UWindsor

Electronic Theses and Dissertations Theses, Dissertations, and Major Papers

10-5-2017

FPGA-Based Acceleration of Expectation Maximization Algorithm using High Level Synthesis

Mohammad Abdul Momen
University of Windsor

Follow this and additional works at: https://scholar.uwindsor.ca/etd

Recommended Citation

Momen, Mohammad Abdul, "FPGA-Based Acceleration of Expectation Maximization Algorithm using High Level Synthesis" (2017). *Electronic Theses and Dissertations*. 7279.
https://scholar.uwindsor.ca/etd/7279

This online database contains the full-text of PhD dissertations and Masters' theses of University of Windsor students from 1954 forward. These documents are made available for personal study and research purposes only, in accordance with the Canadian Copyright Act and the Creative Commons license—CC BY-NC-ND (Attribution, Non-Commercial, No Derivative Works). Under this license, works must always be attributed to the copyright holder (original author), cannot be used for any commercial purposes, and may not be altered. Any other use would require the permission of the copyright holder. Students may inquire about withdrawing their dissertation and/or thesis from this database. For additional inquiries, please contact the repository administrator via email (scholarship@uwindsor.ca) or by telephone at 519-253-3000ext. 3208.

FPGA-Based Acceleration of Expectation Maximization Algorithm using High Level Synthesis

by

Mohammad Abdul Momen

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

2017

© 2017, Mohammad Abdul Momen

FPGA-Based Acceleration of Expectation Maximization Algorithm using HLS Tool

by

Mohammad Abdul Momen

APPROVED BY:

T. Bolisetti

Department of Civil and Environmental Engineering

M. Abdelkhalek

Department of Electrical and Computer Engineering

M. Khalid, Advisor

Department of Electrical and Computer Engineering

August 1st, 2017

Author's Declaration of Originality

I hereby certify that I am the sole author of this thesis and that no part of this thesis has been published or submitted for publication.

I certify 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 and have included copies of such copyright clearances to my appendix.

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 of any other University or Institution.

Abstract

Expectation Maximization (EM) is a soft clustering algorithm which partitions data iteratively into M clusters. It is one of the most popular data mining algorithms that uses Gaussian Mixture Models (GMM) for probability density modeling and is widely used in applications such as signal processing and Machine Learning (ML). EM requires high computation time and large amount of memory when dealing with large data sets. Conventionally, the HDL-based design methodology is used to program FPGAs for accelerating computationally intensive algorithms. In many real world applications, FPGA provide great speedup along with lower power consumption compared to multi-core CPUs and GPUs. Intel FPGA SDK for OpenCL enables developers with no hardware knowledge to program the FPGAs with short development time. This thesis presents an optimized implementation of EM algorithm on Stratix V and Arria 10 FPGAs using Intel FPGA SDK for OpenCL. Comparison of performance and power consumption between CPU, GPU and FPGA is presented for various dimension and cluster sizes. Compared to an Intel(R) Xeon(R) CPU E5-2637 our fully optimized OpenCL model for EM targeting Arria 10 FPGA achieved up to 1000X speedup in terms of throughput (${}^{T}{}_{speedup}$) and 5395X speedup in terms of throughput per unit of power consumed (${}^{T/P}{}_{speedup}$). Compared to previous research on EM-GMM implementation on GPUs, Arria 10 FPGA obtained up to 64.74X ${}^{T}{}_{speedup}$ and 486.78X ${}^{T/P}{}_{speedup}$.

Acknowledgements

Firstly, I would like to express my deepest gratitude to my supervisor Dr. Khalid for giving me the opportunity to work under his supervision. I am very grateful for his advice, guidance, support and encouragement during my study and research.

I would like to thank Dr. Maher AbdelKhalek and Dr. Tirupati Bolisetti for being part of my thesis committee. They provided insightful suggestions to improve my research.

I am also grateful to Dr. Roberto Muscedere for his help in maintaining the workstations in our research lab.

I am dedicating my research to my parents. Without their continued support and care, I could not finish this research.

Table of Contents

Author's Declaration of Originality iii

Abstract iv

Acknowledgements v

List of Tables viii

List of Figures ix

List of Acronyms x

Chapter 1 Introduction 1

1.1. Motivation 1

1.2. Objectives 2

1.3. Thesis Outline 2

Chapter 2 High Level Synthesis for FPGAs 4

2.1. FPGA Architecture 4

2.2. High-Level Synthesis 5

2.3. Overview of OpenCL 6

2.3.1. Platform Model 7

2.3.2. Execution Model 8

2.3.3. Memory Model 10

2.3.4. Programming Model 11

2.4. Intel FPGA SDK for OpenCL 12

2.4.1. Overview 12

2.4.2. Design Flow 13

2.4.3. Optimization Strategies in Intel FPGS SDK for OpenCL 15

2.5. Summary 20

Chapter 3 Expectation Maximization 21

3.1. Background 21

3.2. Previous Work on Accelerating EM-GMM 24

3.2.1. Acceleration on GPUs 24

3.2.2. Acceleration on FPGAs 24

3.3. Summary 26

Chapter 4 Optimized EM-GMM OpenCL FPGA Implementation 27
4.1. EM-GMM OpenCL Model for FPGA 27
4.1.1. Initialization 29
4.1.2. M-Step 29
4.1.3. Pre-Calculation E-Step 31
4.1.4. E-Step 32
4.2. Optimization for different Problem Sizes 35
4.3. Summary 36
Chapter 5 Experimental Results 37
5.1. Experimental Setup 37
5.2. Dataset 38
5.3. Power Measurement 39
5.4. Performance analysis 40
5.4.1. Performance Results 40
5.4.2. Speedup 45
5.4.3. FPGA Resource Utilization 50
5.4.4. Performance Comparison Between CPU and FPGA in Relation to FPGA Resource Utilization 54
5.4.5. Comparison with Previous EM Research 57
5.5. Verification of Results 59
5.6. Summary 59
Chapter 6 Conclusion 60
6.1. Future Work 61
References 62
Appendix A: EM-GMM Opencl Kernel Code 67
Vita Auctoris 83

List of Tables

Table 1 Overview of Currently Available HLS CAD Tools [12]. 6
Table 2 Previous Work on Accelerating EM-GMM. 25
Table 3 Dimension and Cluster achived on Stratix V A7 FPGA and Arria 10 FPGA. 35
Table 4 Device list and Description. 37
Table 5 FPGA Device Specification. 38
Table 6 Power Consumption of CPUs and FPGAs. 39
Table 7 Performance comparison of FPGAs over CPUs and FPGA Resource Usage for highest cluster achieved by each dimension. 55
Table 8 Performance comparison of FPGAs and FPGA Resource Usage for highest cluster achieved by each dimension. 56
Table 9 IPS*and speedup gained by our implementation over other implementations. 57
Table 10 IPS*/power consumption and speedup gained by our implementation over other implementations. 58

List of Figures

Figure 1 Intel Arria 10 Architecture. 5
Figure 2 Platform Model, from [14]. 7
Figure 3 Execution Model (2D Range), from [19]....... 9
Figure 4 Memory Model, from [14]....... 10
Figure 5 Memory model of Intel FPGA SDK for OpenCL....... 12
Figure 6 Intel FPGA SDK for OpenCL Design Flow [19]. 14
Figure 7 Difference between Data Parallelism and Pipeline Parallelism, taken from [16]. 17
Figure 8 EM-GMM operational flow OpenCL FPGA. 28
Figure 9 FPGA Execution Time for EM. 42
Figure 10 CPU Execution Time. 42
Figure 11 FPGA Throughput for EM. 43
Figure 12 CPU Throughput for EM. 43
Figure 13 FPGA Throughput/Power. 44
Figure 14 CPU Throughput/Power. 44
Figure 15 CPU1 Speedup (T) over CPU2. 45
Figure 16 Stratix V FPGA Speedup (T) over CPUs. 46
Figure 17 Arria 10 FPGA Speedup (T) over CPUs....... 46
Figure 18 Arria 10 FPGA Speedup (T) overs Stratix V FPGA 47
Figure 19 CPU 1 Speedup (T/P) over CPU 2 48
Figure 20 Stratix V FPGA Speedup (T/P) over CPUs. 48
Figure 21 Arria 10 FPGA Speedup (T/P) over CPUs. 49
Figure 22 Arria 10 FPGA Speedup (T/P) over Stratix V FPGA. 49
Figure 23 FPGA Logic Element Utilization. 51
Figure 24 FPGA ALUT Utilization. 51
Figure 25 FPGA Register Utilization....... 52
Figure 26 FPGA Memory Block Utilization. 52
Figure 27 FPGA DSP Block Utilization. 53
Figure 28 FPGA Operational Frequency. 53

List of Acronyms

EM	Expectation Maximization
GMM	Gaussian Mixture Model
FPGA	Field Programmable Array
ML	Machine Learning
HDL	Hardware Descriptive Language
HLS	High-Level Synthesis
CPU	Central Processing Unit
ALM	Adaptive Logic Module
AOCL	Altera SDK for OpenCL
AOC	Altera Offline Compiler
API	Application Programming Interface
ASIC	Application Specific Integrated Circuits
CUDA	Compute Unified Device Architecture
GPU	Graphics Processing Unit
HPC	High-Performance Computing
LAB	Logic Array Block
LE	Logic Element
LUT	Look Up Table
OpenCL	Open Computing Language
SIMD	Single Instruction Multiple Data
SPMD	Single Program Multiple Data

MLE	Maximum Likelihood Estimation
HLL	High Level Language

Chapter 1 Introduction

1.1. Motivation

Currently, Machine Learning algorithms are highly used to solve complex computationally intensive problems. Maximum Likelihood Estimation (MLE) carries a lot of standing in parametric estimation. MLE is used to calculate and fit a statistical model from sample dataset. EM computes Maximum Likelihood (ML) iteratively where the dataset is incomplete or some data is missing. EM contains two steps: the E-step or Expectation step which computers log-likelihood from the dataset and assigns each sample to clusters consequently and M-step or Maximization step maximizes the log-likelihood provided by E-step. Both steps of EM are repeated until log-likelihood reaches convergence [1]. EM is a computationally complex problem and consumes more memory and hardware resources compared to other clustering algorithms. As a result, hardware accelerators such as Graphic Processing Units (GPU) [2][4] and Field Programmable Gate Arrays (FPGA) [3][5][6] have been utilized to increase the throughput of the EM.

Traditionally, GPUs are used to accelerate ML algorithms because of high throughput and better memory bandwidth. However, GPU has a huge drawback. GPU power consumption is very high. On the other hand, FPGA-based accelerators provide high throughput with low power consumption.

There are three [3][5][6] FPGA-based EM implementations currently available in the literature. Two of these [5][6] designs were implemented at the Register Transfer Level (RTL) level. This is a time-consuming design methodology. Intel FPGA SDK for

OpenCL [23] is a High-level synthesis (HLS) tool that provides the opportunity to program FPGA in a high-level language, specifically Open Computing Language (OpenCL) to accelerate the design process. HLS helps developers to program FPGA with little FPGA knowledge and to achieve peak performance by utilizing optimized OpenCL specification. The Altera Offline Compiler (AOC) automatically converts OpenCL code to optimized Verilog model and compiles into FPGA hardware binaries. Since developers need less hardware knowledge, time to market and cost for HLS is much lower than RTL-based design methodology.

1.2. Objectives

The main objectives of this thesis are to answer following questions:

- Can FPGA gain better throughput compared to other HPC platforms using Intel FPGA SDK for OpenCL?
- How much speedup will we gain in Arria 10 FPGA compared to Stratix V FPGA?
- What dimension and cluster sizes can we fit in Stratix V and Arria 10 FPGAs?

1.3. Thesis Outline

The thesis is organized as follows:

Chapter 2 discusses background information about FPGA, HLS, OpenCL and Intel FPGA SDK for OpenCL.

In Chapter 3, we first provide a brief discussion about Expectation Maximization and previous EM implementation done by different FPGA and GPU based HPC systems.

Chapter 4 provides explanation on how we designed the kernels for Expectation Maximization algorithm to get a better speedup.

In Chapter 5 we present HLS synthesis results for EM and their analysis. We present a comparison of EM implementation results using state of the art CPUs, GPUs and FPGAs.

We conclude in Chapter 6 will a summary this research and suggestions for future work.

Chapter 2 High Level Synthesis for FPGAs

Computation demand in HPC has increased rapidly in recent years and this trend will continue for the foreseeable future. Traditionally multi-core CPUs were for HPC. Because of ease of programming, data and instruction parallelism and high throughput general purpose GPU is widely used for HPC. But that comes at the cost of high power consumption. To obtain high throughput with less power consumption architecture, things like memory organization and interconnect topology needs to comply with algorithmic requirements [7]. In [8], is has been assessed that for any HPC platform half of its lifetime cost is power consumption. Since FPGAs has reprogrammable, reconfigurable resource with high throughput and less power consumption, FPGA is more suitable for HPC. In the following section, we will discuss FPGA Architecture, HLS, OpenCL and Intel FPGA SDK for OpenCL.

2.1. FPGA Architecture

Field Programmable Gate Array (FPGA) is a reprogrammable and reconfigurable large Integrated Circuit (IC) that consists of a large number of of Look-Up Tables (LUT) and flip flops. These can be used to create custom hardware functionality and execute any algorithm as a digital circuit. Development in FPGA is less costly and time-consuming, though Application Specific Integrated Circuit (ASIC) has more throughput and less power consumption compared to FPGA [9]. Modern FPGAs consists of both fine-grained and coarse-grained programmable blocks. Nearly 70% of the FPGA market is controlled

by Xilinx and Intel [10]. The overall architecture of a state-of-the-art FPGA is shown in Figure 1, which is an Intel Arria 10 FPGA [11].

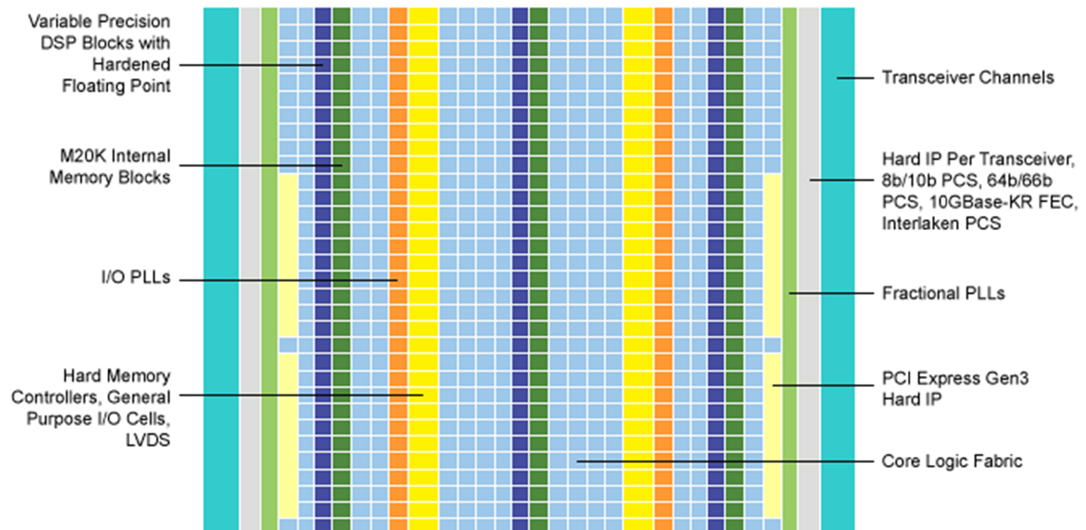

Figure 1 Intel Arria 10 Architecture.

Traditionally, Hardware Descriptive Language (HDL) such as Verilog and VHDL is used to design hardware at the Register Transfer Level (RTL) or Gate level for implementation on FPGAs. An RTL level synthesis Computer Aided Design (CAD) tool first reads the HDL model and then it synthesizes, places and routes the design on targeted FPGA. This requires extensive hardware knowledge. People with little or no hardware knowledge cannot program FPGA and because of this most companies are using CPUs and GPUs for HPC rather than FPGAs.

2.2. High-Level Synthesis

High-level Synthesis (HLS) is a method where a developer can use a High-level programming language such as C/C++ or System C to specify any algorithm which is be

synthesized in an optimized manner to hardware. With HLS, even software a developer can target any supported FPGA and synthesize optimized hardware. Currently, there are several HLS CAD tools available in both academia and industry. Table 2 shows some of these HLS CAD tools. For our research, we used Intel FPAG SDK for OpenCL.

Table 1 Overview of Currently Available HLS CAD Tools [12].

Owner	Compiler	License	Input	Output
Intel	Intel FPGA SDK for OpenCL	Commercial	C with OpenCL	Verilog
Xilinx	Vivado HLS	Commercial	C/C++ System C	VHDL/Verilog System C
Cadence	CtoS	Commercial	SystemC TLM/C++	Verilog System C
Mentor Graphics	DK Design Suite	Commercial	Handel-C	VHDL/Verilog
Maxeler	MaxCompiler	Commercial	MaxJ	RTL
Synopsys	Synphony C	Commercial	C/C++	VHDL/Verilog System C
LegUp	U.Toronto	Academic	C	Verilog

2.3. Overview of OpenCL

Open Computing Language (OpenCL) is the first industry standard framework for heterogeneous computing with the compatibility of HLS. Normally, CPUs, GPUs, DSPs are included in the heterogeneous platform. Because of HLS, FPGAs are added heterogeneous platform list. OpenCL consists of C99 based programming standard and also Application Program Interface (API). Using OpenCL, a developer can program multiple devices where using Compute Unified Device Architecture (CUDA), a developer can only program Nvidia GPUs. OpenCL is open source programming language. It is maintained and updated by Khronos Group and various companies like

Creative Technology, ARM Holdings, ZiiLABS, IBM, Samsung, Imagination Technologies, Qualcomm, Apple, Nvidia, Vivante, AMD, Xilinx, and Intel support and uses it [13].

The OpenCL framework is divided into four models.

2.3.1. Platform Model

Every platform consists of 2 units. First one is host unit which is usually a CPU and second unit is device unit which is one of more combination of devices Figure 2. Host unit controls device runtime. In device unit, any device controlled by the OpenCL platform contains one or more Compute Units (CUs) and each CU consists of multiple Processing Units (PUs). The Processing Elements (PEs) or work-items does the actual computation.

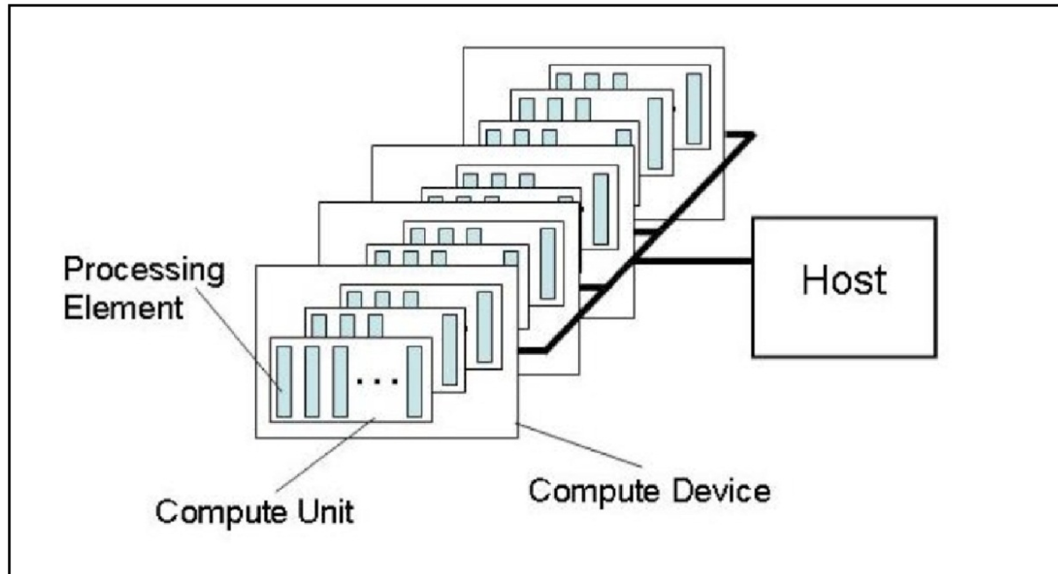

Figure 2 Platform Model, from [14].

2.3.2. Execution Model

An OpenCL program needs host code and kernel code to run and execute. The developer writes Host code in C/C++ with API to manage memory objects, program objects and command queues for the kernel. Kernel code consists of the main computationally intensive part of the algorithm which is executed on the device or devices.

- **Context:**

  The contest consists of all crucial information regarding the targeted one or more devices and it is created for one or more devices.

- **Program Objects:**

  During runtime, the program provides a dynamic library for multiple kernels and also includes kernel/kernels binary implementation.

- **Memory Object:**

  Memory objects are used to transfer back and forth between the host and one or more devices. A memory object is used as kernel data input and output. We will discuss more memory object on next sections.

- **Command Queue:**

  Using command queue host manages the execution of commands. Command queue contains three commands: For Memory command to transfer data within memory, Kernel command to launch one or kernels and Synchronization command creates a point to manually synchronize the host code.

- **Work Groups and Work Items:**

  Kernels consist of multiple threads and each thread is called work-item. Work-item can be multiple dimensions distributed in multi-dimensional space. Each work-item has a unique ID called Global-ID in the index space. Each work-item executes the same operation with different data. A collection of Work-items in a dimension or collection of work-items is called a workgroup. Each workgroup has its own Group-ID and work-item within has its own Local-ID.

  Mapping of work-items in a 2D Range space is shown in figure 3. We are giving an example with four workgroups and total of 10x10 work-items are equally divided into all four workgroups (5x5 work-items in each). Inside a work-group, all work-items are work-items are executed concurrently. Conversely, work-items of different work-groups may or may not run concurrently.

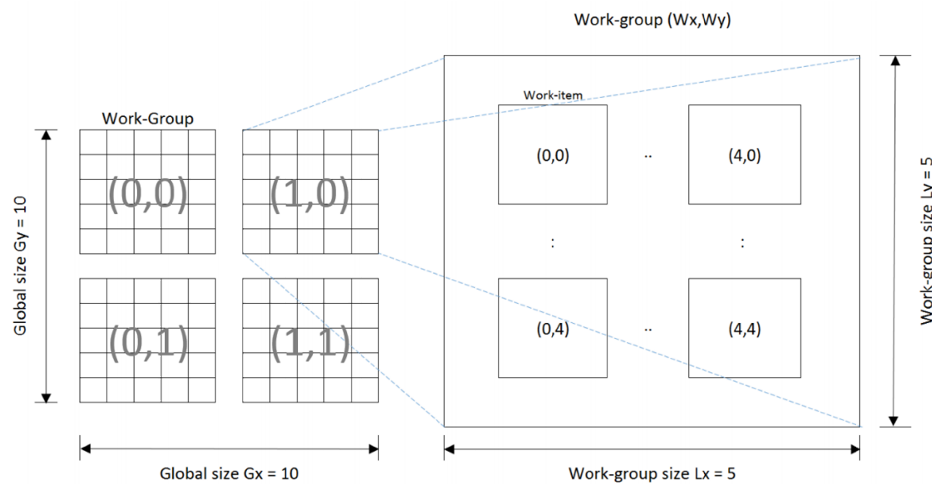

Figure 3 Execution Model (2D Range), from [19].

2.3.3.Memory Model

Four OpenCL memory types of are shown in figure 4 and described below:

- **Global Memory:**

  Global Memory is visible and accessible to both host and one or more devices by all work-items for reading/writing data.

- **Constant Memory:**

  Stores constant data in global memory during kernel execution. It is a read-only memory. It is much faster than global memory because data is copied to on-chip memory right before the kernel is executed.

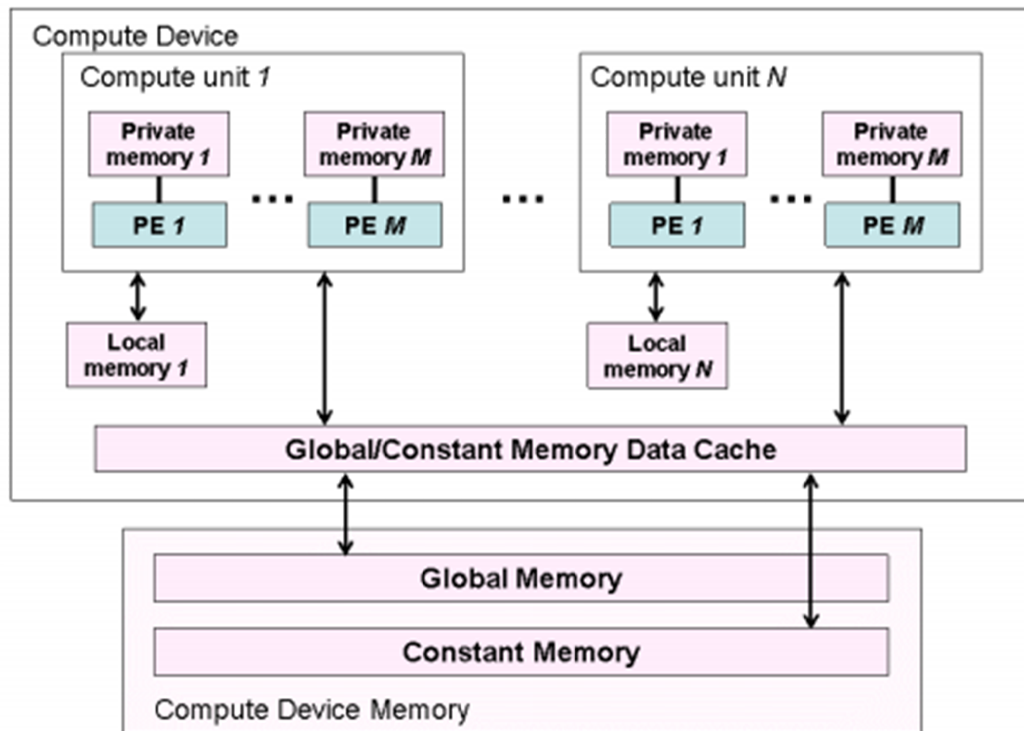

Figure 4 Memory Model, from [14].

- **Local Memory:**

  All work items within a workgroup can access local memory. For this, each work-item can collect the data as fast as possible and do the calculation.

- **Private memory:**

  Private memory is accessible for a single work-item.

2.3.4.Programming Model

OpenCL consists of two types of data parallelism: task parallelism and data parallelism. In task parallelism, multiple large kernels that contain single work-item execute concurrently at the same time. In data parallelism, kernels contain multiple work-items and each work-item within a workgroup calculates the same operation with different data concurrently based on their Global-ID. Data parallelism falls under the category of Single Program Multiple Data (SPMD) and Single Instruction Multiple Data (SIMD) stream. Because of architecture, data parallelism is suitable for GPUs. However, in FPGAs both parallelisms are suitable and it also supports Single Instruction Single Data (SISD), Multiple Instruction Single Data (MISD), and Multiple Instruction Multiple Data (MIMD).

2.4. Intel FPGA SDK for OpenCL

2.4.1. Overview

Intel FPGA SDK for OpenCL is one of the HLS tools that enable a developer to program an FPGA using a High-Level Language (HLL) specification. For parallel programming, both CPU and GPU use SIMD and/or SPMD model. During program execution, first the instruction is fetched, decoded and then data is transferred back and forth between register file and memory system. This is inefficient because processing unit waits for the data. Intel FPGA SDK for OpenCL can generate a highly customizable architecture that can support all four models of computation (SIMD, SISD, MISD, and MIMD) individually and or in combination. Unlike GPU, FPGA can support a different kind of algorithm and make a fully customized power-efficient high throughput hardware compared to Von-Neumann processors [15].

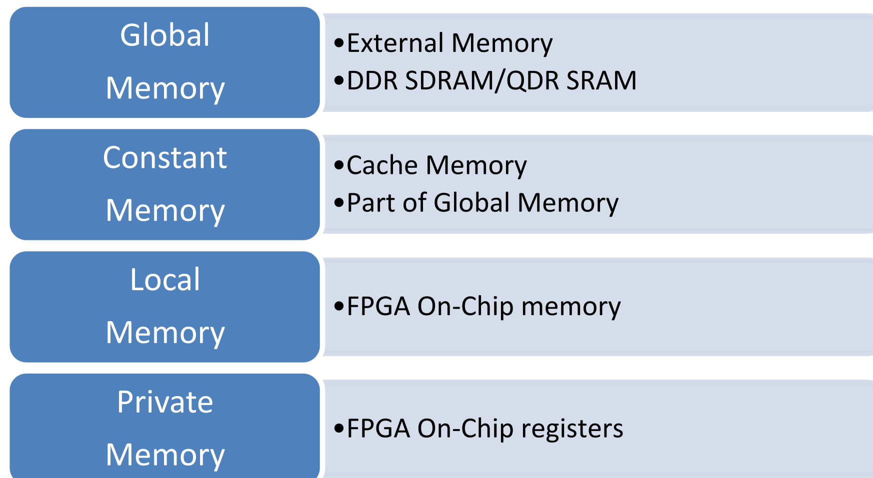

Figure 5 Memory model of Intel FPGA SDK for OpenCL.

Like any other OpenCL platforms, Intel FPGA SDK for OpenCL supports all four kinds of the memory systems. Global memory is the external memory of the FPGA

system which might be Double Data Rate (DDR) Synchronous Dynamic Random Access Memory (SDRAM) DDR3 SDRAM, DDR4 SDRAM, DDR5 SDRAM and/or Quad Data Rate (QDR) Static Random Access Memory (SRAM) [16] with large capacity and long latency. Constant memory is a special type of global memory which is loaded in cache during runtime. Local memory is stored in on-chip FPGA memory and this one has low capacity, high bandwidth, and less latency compared to global memory. Finally, compared to DDR memory private memory is assigned to FPGA on-chip registers which have the lowest latency, highest bandwidth and lowest capacity. The memory model used in Intel FPGA SDK for OpenCL is summarized in Figure 5.

Intel FPGA SDK for OpenCL supports OpenCL 1.0, parts of OpenCL 1.2 and OpenCL 2.0. For Example, OpenCL started using Channels/pipes from OpenCL 2.0 which gave FPGA better data transfer speed compared to GPUs. Channel/pipe is a First-In-First-Out (FIFO) buffer with a channel ID and depth that enables to transfer data back and forth between work-items in the same kernel and/or different kernels which require no additional synchronization and host interaction [17]. Furthermore, channels can be used to synchronize work items and/or kernels because of work-item and/or kernel stalls if they try to read an empty buffer and/or write to a channel that is full [18].

2.4.2. Design Flow

Initially, Intel FPGS SDK for OpenCL creates an emulated label by compiling kernel program (.cl) in its Altera Offline Compiler (AOC). During the emulation stage, AOC checks for errors (syntax, functional, logical, etc.) and also checks for stalls. Furthermore, in this stage AOC provides optimization report regarding memory transaction, pipeline execution to get better throughput and reduce wait time for each

stage in a pipeline. Because of this information, a developer can design an optimized hardware in a short period of time. Next step is to fully compile and synthesize the kernel program with AOC to directly generate Verilog RTL design from OpenCL code.

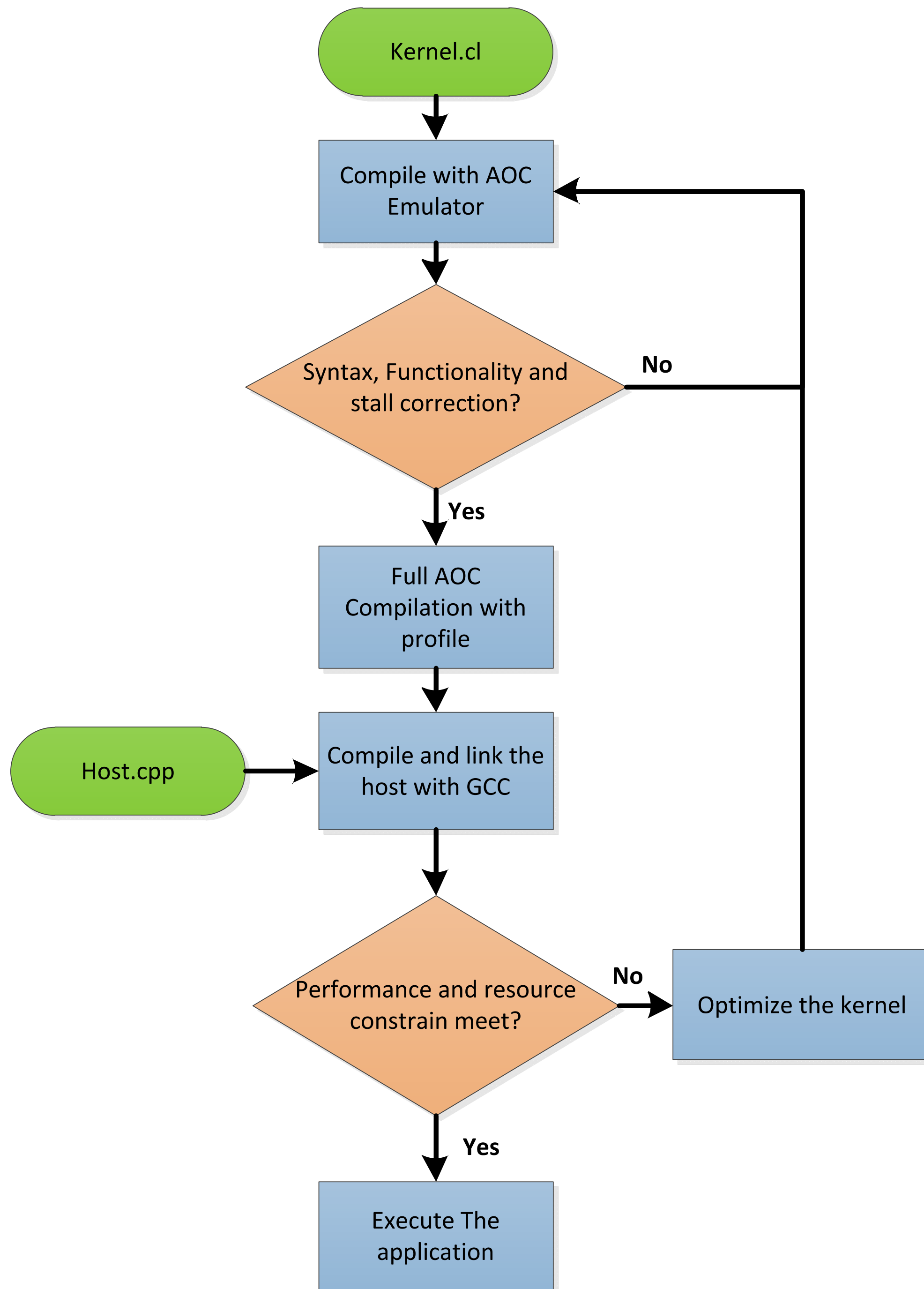

Figure 6 Intel FPGA SDK for OpenCL Design Flow [19].

AOC automatically generates pipelines (if instruction is provided) and memory interaction between kernels and different memory regions. Depending on FPGAs and applications a full compilation takes 4-8 hours. In this stage, AOC will provide a report that will enable the developer to check if his/her design will fit in the FPGA or not. He/she might need to change his/her design if the design fails to meet the resource requirement. Finally, GNU Compiler Collection (GCC) is used to compile the host program along with FPGA executable and then run on that same system. Figure 6 shows the design flow of Intel FPGS SDK for OpenCL.

2.4.3.Optimization Strategies in Intel FPGS SDK for OpenCL

One of the core rules in HPC is to increase speed in computation part and reduce communication time or number of global memory access and host memory access because of communication bottleneck between global memory and FPGA and host memory and global memory, respectively. In this section, we will discuss strategies to increase computation speed and reduce communication time.

2.4.3.1. Parallelism Strategies

Intel FPGA SDK for OpenCL supports Task Parallelism, Data Parallelism, and Pipeline/Loop Parallelism.

- **Data Parallelism:**

  In data parallelism, the kernel uses SPMD/SIMD model to access ND (N: number of dimensions) grid work-items. Like GPU, a work-group is a combination

of work-items and each work-item executes the same operation with different data. Workgroup size is equal to a number of work-items in a workgroup. A number of work items in a work-group needs to be managed to ensure optimal hardware resource usage and maintain parallelism within workgroups. In data parallelism, loops with no dependencies will get the highest throughput.

- **Loop/Pipeline Parallelism:**

Pipeline/Loop parallelism is known as *Task* in Intel FPGA SDK for OpenCL and the kernel is a single thread work-item. In GPUs, single thread work-item is used in data dependent sections and is inefficient. While processing single thread work-item, other processing units remain idle which is a waste of resources. Conversely, FPGA makes pipeline architecture by breaking down loop into multiple stages to resolves loop carried dependencies. Compiler pipelines each stage of a loop and launches next iterations as soon as loop carried dependencies have been resolved. The developer has no control over pipeline structure and scheduling. The only thing a developer can do is to reduce, remove or simplify loop carried dependencies.

Figure 7 shows a visual comparison between data parallelism and pipeline parallelism. In this example, there is a kernel of 6 work-items with six stages (A-E). Data parallelism takes ten clock cycles to finish (output data is ready on 10th clock cycle) and at a time executes three work-items. In pipeline parallelism, all six stages are launched in stages in a pipeline manner within a kernel. Though it took the same amount of time to finish the work, the data output of the first loop is ready for 6th clock cycle wherein data parallelism same data output was ready in 10th clock cycle. Pipeline parallelism has higher throughput than data parallelism.

SIMD Parallelism (GPU)	1 A	1 B	1 C	1 D	1 E	4 A	4 B	4 C	4 D	4 E
	2 A	2 B	2 C	2 D	2 E	5 A	5 B	5 C	5 D	5 E
	3 A	3 B	3 C	3 D	3 E	6 A	6 B	6 C	6 D	6 E
Clock Cycle	1	2	3	4	5	6	7	8	9	10
Pipeline Parallelism (FPGA)	1 A	2 A	3 A	4 A	5 A	6 A				
		1 B	2 B	3 B	4 B	5 B	6 B			
			1 C	2 C	3 C	4 C	5 C	6 C		
				1 D	2 D	3 D	4 D	5 D	6 D	
					1 E	2 E	3 E	4 E	5 E	6 E

Figure 7 Difference between Data Parallelism and Pipeline Parallelism, taken from [16].

- **Task Parallelism:**

  Intel FPGA SDK for OpenCL executes task parallelism by running kernels in a pipeline manner using command queue. Using multiple asynchronous command queues AOC executes kernels concurrently. However, for task parallelism an explicit synchronization point (*clfinish* and *channel*) is required. To get the highest throughput from task parallelism, the application needs to be divided into multiple kernels.

2.4.3.2. Throughput-based Optimizations

There are three methods to get the highest throughput from Intel FPAG SDK for OpenCL.

- **Vectoring work-items:**

  By vectoring work-items, reading/writing data and doing arithmetic/logic operations can be done in SIMD fashion. The compiler will create kernel data path based on a number of vectors and this will reduce the number of memory accesses. Vectoring work-items increase efficiency in memory read/write.

- **Loop Unrolling:**

  In every application, there are lots of loops. Unrolling loops fully or partially will increase throughput linearly. However, based on unrolling factor, hardware resource usage will increase.

- **Multiple Compute Units (CU):**

  In Intel FPGA SDK for OpenCL, multiple copies of same kernel hardware can be created in addition to kernel vectoring and loop-unrolling. There will be a communication bottleneck because multiple CUs will share same global memory. So, speedup might not be linear. Multiple CUs will consume more resource and it will decrease operational frequency.

2.4.3.3. Optimizing Data Access/Transfer

One of the main barriers of HPC is memory bandwidth and proficient memory access to reduce communication bottleneck. We are provided with some techniques to optimize data access and/or reduce communication bottleneck [20].

- **Aligning Memory:**

  On the Host side, memory alignment has to be 64-bytes aligned. This drastically increases data transfer efficiency. On Linux, *posix_memalign* function and on Windows, *malloc* function is used on host code to align memory.

- **Caching Local Memory:**

  We discussed earlier that Local memory or FPGA On-Chip Memory has high bandwidth and low latency compared to global memory. Because global memory has low bandwidth, we repeatedly grab data block by block from global memory and store on local memory before computation so that work-items can access data in short time. All work-items in a workgroup can access local memory and use local memory for increased throughput.

- **Memory Coalescing:**

  In all HPC platforms including FPGA, memory coalescing improves memory efficiency by reducing a number of memory accesses and/or reading data serially. This is crucial when reading/writing data from global memory.

- **Channels:**

  In GPUs, to transfer data between kernels, the data first needs to be stored in global memory and then another kernel will read the data to process it. Because global memory has high latency and low bandwidth getting speedup is hard. Because FPGA architecture is customizable, Intel came up with a FIFO based bus called channel. Using channels, consumer kernel can launch as soon as data is available in producer kernel. Since FIFO based bus stores data in On-chip memory temporarily, bandwidth

is high and so throughput is high. However, one of the drawbacks of channels is we cannot vectorize work-items and create multiple CUs.

More tips and tricks to better optimize kernels in Intel FPGA SDK for OpenCL are presented in [20].

2.5. Summary

This chapter first explained FPGA architecture, HLS and gave an overview of OpenCL. Then we described Intel FPGA SDK for OpenCL and the OpenCL coding strategies to optimize OpenCL based FPGA designs.

Chapter 3 Expectation Maximization

3.1. Background

This thesis focuses on implementing and accelerating Expectation Maximization (EM) for Gaussian Mixture Models (GMMs). In the following section, we describe GMMs and EM for GMMs.

- **Gaussian Mixture Models:**

  GMM is a probabilistic model that contains multiple Gaussian distributions in a linear combination. A GMM with *D* dimensions, *M* Clusters (Gaussian components) and *N* number of points that can be represented by:

$$p(x_n|\Theta) = \sum_{m=1}^{M} \Upsilon(x_n|\mu_m, \Theta_m)\, \omega_m \qquad \textbf{(1)}$$

Where,

- $x_n = (x_{1d}, x_{2d}, \ldots \ldots, x_{Nd})$ is number of points with *D* dimensions. In vector representation, sample data set is a (*N*, *D*) matrix.
- $\Upsilon(x_n|\mu_m, \Theta_m)$ is Gaussian probability density function of *M* number of clusters/ Gaussian components. It is calculated using mean $\mu_m = (\mu_1, \mu_2, \ldots \ldots, \mu_M)$, sample dataset, inverse covariance Θ_m^{-1} and log determinant $|\Theta|_m^{1/2}$ of covariance Θ_m. Probability density function can be represented by,

$$\Upsilon(x_n|\mu_m, \Theta_m) = \frac{\exp\{-\frac{1}{2}(x_n - \mu_m)^T \Theta_m^{-1} (x_n - \mu_m)\}}{(2\pi)^{D/2} |\Theta|_m^{1/2}} \qquad \textbf{(2)}$$

- $\omega_m = (\omega_1, \omega_2, \ldots \ldots, \omega_M)$ is mixture coefficient/weight of *M* number of clusters .

In short, mean, sample dataset, weight and covariance matrix is required to calculate GMM.

- **Expectation Maximization (EM) for GMMs:**

  EM for GMM (EM-GMM) is a probabilistic method to calculate likelihood with incomplete dataset [21] of GMM. EM-GMM calculates likelihood iteratively. First we generate random dataset and initialize all of the parameters. Then we update the parameters by alternating between two following steps until point of convergence is reached.

  - **Expectation (E) Step:**

    Membership or log likelihood value φ_{nm} is calculated for each data instance x_n with respect to each cluster m. membership value φ_{nm} can be calculated by following equation:

$$\varphi_{nm} = \frac{\Upsilon(x_n|\mu_m, \Theta_m)\, \omega_m}{\sum_{m=1}^{M} \Upsilon(x_n|\mu_m, \Theta_m)\, \omega_m} \quad \textbf{(3)}$$

  - **Maximization (M) Step:**

    Estimate new values of mean μ_m, weight ω_m and covariance Θ_m using the membership value obtained from the E-Step and replace new value with old ones. For each cluster m, update parameters:

$$N_m = \sum_{n=1}^{N} \varphi_{nm} \quad \textbf{(4)}$$

$$\omega_m = \frac{N_m}{N} \quad \textbf{(5)}$$

$$\mu_m = \frac{1}{N_m} \sum_{n=1}^{N} x_n \varphi_{nm} \quad \textbf{(6)}$$

$$\Theta_m = \frac{1}{N_m} \sum_{n=1}^{N} \varphi_{nm}(x_n - \mu_m)(x_n - \mu_m)^T \quad \textbf{(7)}$$

Algorithm 1 represents the pseudo code for EM-GMM which shows number of memory access and computational complexity. In *Algorithm 1*, line 4 to line 10 represents E-Step and line 12 to line 19 represents M-step. As described in *Algorithm 1*, both E-Step and M-Step execute iteratively until convergence of log likelihood. Computational complexity for each iteration of EM algorithm is O (ND^2K) [22].

Algorithm 1 EM-GMM
INPUT: sample dataset x_{nd}
OUTPUT: mean μ_{md} , weight ω_{md} and covariance Θ_{mdd}
1. **For all** n ∈ 1 to N **do**
2. initialize parameters ($\mu_m, \omega_m, \Theta_m$)
3. **while** not convergence **do**
4. **for all** m ∈ 1 to M **,** n ∈ 1 to N **do**
5. $sum_m = 0$
6. **for all** d_1 ∈ 1 to D, d_2 ∈ 1 to D **do**
7. $h_{mn} = h_{mn} + \omega_m \frac{\exp\{-\frac{1}{2}(x_{nd_1} - \mu_{md_1})(x_{nd_2} - \mu_{md_2})^T \Theta_{md_1 d_2}^{-1}\}}{(2\pi)^{D/2} \lvert\Theta\rvert_m^{1/2}}$
8. $sum_m = sum_m + h_{mn}$
9. **for all** n ∈ 1 to N, m ∈ 1 to M **do**
10. $\varphi_{nm} = \frac{h_{mn}}{sum_m}$
11. $N_m = 0, \omega_m = 0, \mu_m = 0, \Theta_m = 0$
12. **for all** n ∈ 1 to N, m ∈ 1 to M **do**
13. $N_m = N_m + \varphi_{nm}$
14. **for all** m ∈ 1 to M **do**
15. $\omega_m = \frac{N_m}{N}$
16. **for all** n ∈ 1 to N, m ∈ 1 to M, d ∈ 1 to D **do**
17. $\mu_{md} = \mu_{md} + \frac{1}{N_m} \sum_{n=1}^{N} x_{nd} \varphi_{nm}$
18. **for all** m ∈ 1 to M, n ∈ 1 to N, d_1 ∈ 1 to D, d_2 ∈ 1 to D **do**
19. $\Theta_{md_1 d_2} = \frac{1}{N_m} \sum_{n=1}^{N} \varphi_{nm}(x_{nd_1} - \mu_{md_1})(x_{nd_2} - \mu_{md_2})^T$

3.2. Previous Work on Accelerating EM-GMM

This section provides detailed description of previous EM-GMM implementation research.

3.2.1. Acceleration on GPUs

Pongbar from Rochester Institute of Technology presented a CUDA implementation of EM-GMM algorithm [22]. This research closely matches with our research because they worked with non-diagonal covariance matrix, which are suitable for massive dataset and they compared their work against multiple GPUs and reference implementations. In [22], they achieved maximum of 73.5X speedup on Nvidia GTX260 and 120x speedup on Nvidia C1060x2 against Intel Xeon 2.5 GHz Quad Core E5420 CPU.

In [2], they implemented EM-GMM on Geforce 8800 ULTRA and Quadro FX 5800 using CUDA with diagonal-only covariance matrix. They got maximum of 164x speedup against Dual Core 3.0 GHz Pentium IV CPU on Quadro FX 5800.

Altinigneli [4] used an asynchronous approach for executing EM-GMM in contrast to the traditional synchronous approach. They enabled the parallel threads to asynchronously exchange local information/data. Using asynchronous EM-GMM approach, they accelerated convergence and reduced overhead caused by sequential algorithm and limited memory bandwidth. They achieved 720X speedup on NVidia GTX480 against Intel i7-920 2.66 GHz CPU.

3.2.2. Acceleration on FPGAs

A fully pipelined EM-GMM is implemented in [3] using Maxeler MAX3 acceleration card with a Xilinx Virtex-6 FPGA. For their implementation, they used diagonal-only

covariance matrix and fixed point arithmetic dataset. They achieved 517X speedup against Intel Core i3 CPU.

In [5][6], EM is implemented on 4 Xilinx Virtex-6 LX760 FPGAs for 3D computed tomography (CT) Reconstruction. [5], accomplished 85X speedup compared to single thread Intel Xeon 5138 CPU and [6] accomplished 26.9X speedup compared to 16-thread multi-core Intel Xeon E5-2420 CPU.

Ref. No.	Device	N*	C*	D*	CM*	Speedup
[22]	**GPU:** Nvidia GTX260 **CPU:** Intel Xeon 2.5 GHz Quad Core E5420	10^2-10^6 (float)	100	24	N-D*	58x-84x and 73.5x (avg.). vs. CPU
[22]	**GPU:** Nvidia C1060x2 **CPU:** Intel Xeon 2.5 GHz Quad Core E5420	10^2-10^6 (float)	100	24	N-D*	93x-145x and 120x (avg.) vs. CPU
[2]	**GPU:** Geforce 8800 ULTRA **CPU:** DualCore 3.0 GHz Pentium IV	48.6K-153.6K (float)	8-32	8-32	D-O*	20x-119.3x vs. CPU
[2]	**GPU:** Quadro FX 5800 **CPU:** Dual Core 3.0 GHz Pentium IV	48.6K-153.6K (float)	8-32	8-32	D-O*	20x-164.0x vs. CPU
[4]	**GPU:** NVidia GTX480 **CPU:** Intel i7-920 2.66 GHz	2^{20} (float)	10	8	D-O*	720x vs. CPU
[3]	**FPGA:** Maxeler MAX3 acceleration card with a Xilinx Virtex-6 FPGA **CPU:** Intel Core i3 CPU	10^6 (fixed)	2,4,6	3,6	D-O*	517x vs. CPU
[5]	**FPGA:** 4 Xilinx Virtex-6 LX760 **CPU:** Intel Xeon 5138	-	-	3		85X vs. CPU (single thread)
[6]	**FPGA:** 4 Xilinx Virtex-6 LX760 **CPU:** Intel Xeon E5-2420	-	-	3		26.3x vs CPU (Multi Thread)

Table 2 Previous Work on Accelerating EM-GMM.

N* = Number of Points, C* = Number of Clusters, D* = Number of Dimensions,
CM* = Covariance Matrix, D-O* = Diagonal Only, N-D* = Non-Diagonal

Table 2 provides illustrates a comparison of previous work on accelerating EM-GMM on GPUs and FPGAs against CPUs (section 3.2.1 and 3.2.2). For each implementation it shows how much speedup they gained against CPU using different number of points, cluster size and dimension size. It also shows which implementation uses diagonal-only or full/non-diagonal covariance matrix. The reason for choosing diagonal-only over full/non-diagonal covariance matrix is that it reduces computation [2-6]. Note that full/non-diagonal covariance matrix requires a lot of computation [22].

3.3. Summary

In this chapter we first explained how the EM-GMM algorithm works. Then we discussed related research on EM implementation using FPGAs and GPUs.

Chapter 4 Optimized EM-GMM OpenCL FPGA Implementation

4.1. EM-GMM OpenCL Model for FPGA

Data dependencies in original EM-GMM algorithm (*Algorithm 1*) make it impossible to create a fully optimized FPGA design with high throughput. For our research, we implemented a fully-pipelined EM-GMM OpenCL FPGA architecture using same operations flow as [22]. Operational flow of a fully-pipelined EM-GMM OpenCL model for FPGA is shown in Figure 8.

The EM-GMM computation on Inter FPGA SDK for OpenCL can be broken down into five kernels. Full description of kernel 1, 2, 3, 4, 5 is shown in algorithm 2, 3, 4, 5, 6, respectively. Dataflow between host and kernels (through global memory) and between kernels (through channel extension) are shows in Figure 8. During execution sample datasets (x_{nd} and x_{dn}) and membership data is cached to FPGA On-chip memory as they are used repeatedly. From our comprehensive analysis we found that we get better throughput if we implement all five kernels in fully pipelined single thread work item manner rather than *NDRange* Kernel. AOC can only pipeline kernels with single thread work item. Multi-thread work item pipelining is not supported on AOC. Intel FPGA SDK for OpenCL FIFO based Channel Extension is used to directly transfer data within kernels. The Channel extension helped to synchronize the kernels without host involvement. Latency for each iteration reduced because channel extension helps to execute kernels concurrently. Channel helped to gain speed up depending on different

dimension and cluster sizes. The depth of the channel affects FPGA on-chip memory usage.

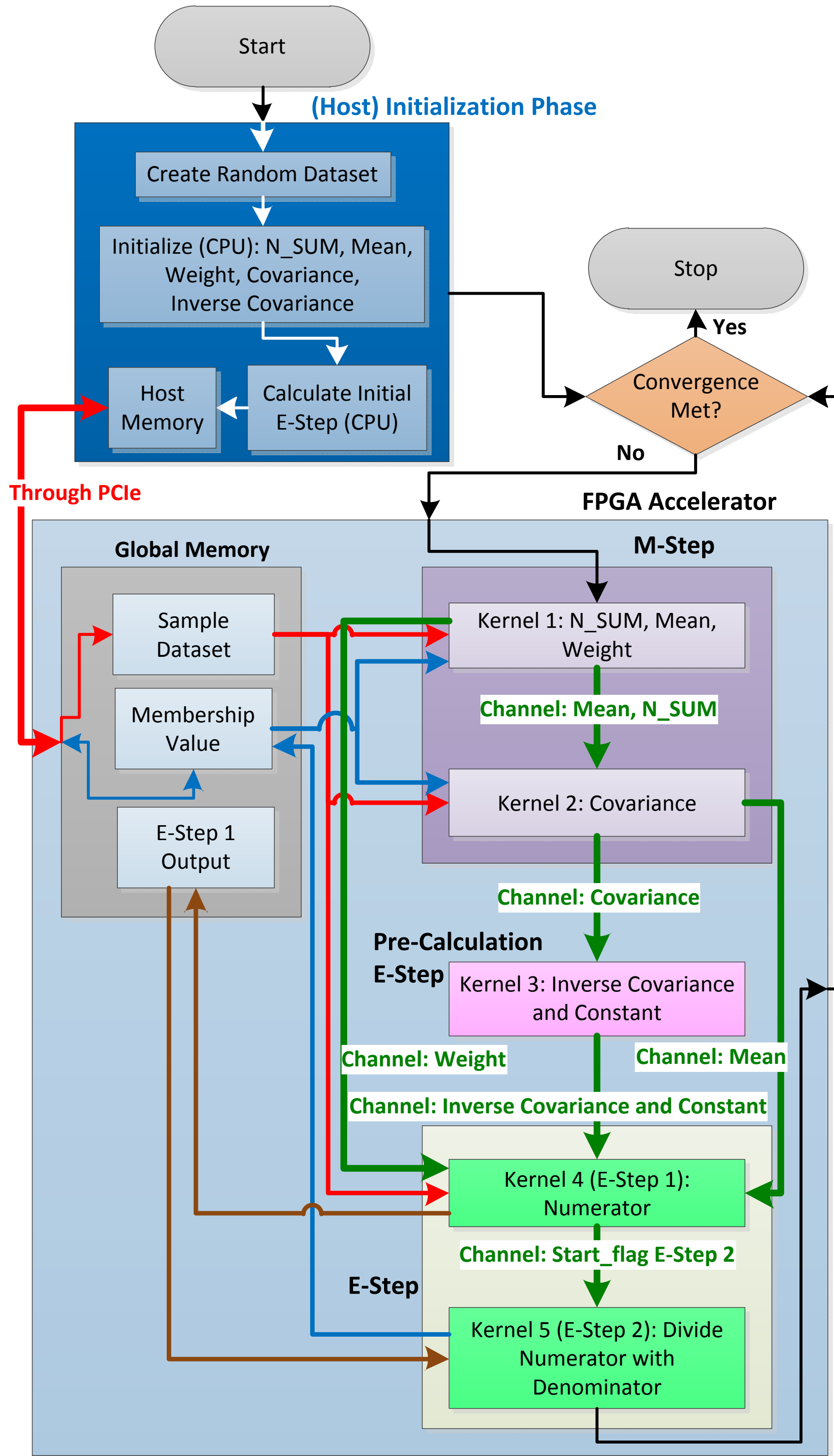

Figure 8 EM-GMM operational flow OpenCL FPGA.

4.1.1.Initialization

For our implementation we first generated random float dataset x_{dn} and transpose of that same dataset x_{nd}. Kernel 1 requires x_{dn} and kernels 2 and 4 require x_{nd} for coalesced memory access from global memory. Global memory is a very large memory, so we do not need to worry about memory usage. Then we calculated initial value of mean μ_{md}, weight ω_m, sum of all membership values in a cluster N_SUM_m, covariance Θ_{mdd}, inverse covariance Θ_{mdd}^{-1} and constant $const_m$ on the host side. Afterwards, we also calculated a full E-Step calculation on the host end to generate initial membership value φ_{mn}. Then we send the random float dataset x_{dn} , transpose of that same dataset x_{nd} and initial membership value φ_{mn} to FPGA global memory for calculation.

Inside FPGA, calculation starts with M-Step and ends with E-Step for each iteration. During acceleration, choosing M-Step first before E-step helps to reduce communication overhead and number of memory access between kernel and global memory. We cannot use Channel extension from E-Step to M-Step because of complicated memory access pattern between these steps and membership value is too big to store in FPGA memory. However, memory access pattern from M-Step to E-Step is not complicated. Connecting M-Step to E-Step with Channel extension reduced global memory access and because of less memory load and store units, hardware resource utilization also decreased.

4.1.2.M-Step

First kernel (*Algorithm 2*) reads random float sample dataset and membership value form global memory to calculate sum of all membership value for a given cluster, mean

and weight and sends sum of all membership value for a given cluster, mean data to kernel 2 and weight data to kernel 4 through channel.

Algorithm 2 (Kernel 1) Calculates: N_SUM_m, μ_{md}, ω_m
G* I*: sample dataset x_{dn} , membership value φ_{mn}. **C* I*:** none. **G* O*:** none. **C* O*:** mean μ_{md} , weight ω_m and N_SUM_m. 1. **for all** m ∈ 1 to M **do** 2. **for all** n ∈ 1 to N **do** 3. $N_SUM_m = N_SUM_m + \varphi_{mn}$ 4. **write_channel:** N_SUM_m 5. **for all** n ∈ 1 to N, d ∈ 1 to D **do** 6. $\mu_{md} = \mu_{md} + \frac{1}{N_SUM_m}\sum_{n=1}^{N} x_{dn}\varphi_{mn}$ 7. **write_channel:** μ_{md} 8. $\omega_m = \frac{N_SUM_m}{N}$ 9. **write_channel:** ω_m

G* = GLOBAL, I* = INPUT, C* = CHANNEL, O* = OUTPUT

Algorithm 3 (Kernel 2) Calculates: covariance Θ_{mdd}
G* I*: sample dataset x_{nd} , membership value φ_{mn}. **C* I*:** mean μ_{md} and N_SUM_m.. **G* O*:** none. **C* O*:** covariance Θ_{mdd}, mean μ_{md}. 1. **for all** m ∈ 1 to M **do** 2. **for all** d ∈ 1 to D **do** 3. **read_channel:** μ_{md} 4. **read_channel:** N_SUM_m 5. **for all** n ∈ 1 to N **do** 6. **for all** d_1 ∈ 1 to D, d_2 ∈ 1 to D **do** 7. $\Theta_{md_1d_2} = \Theta_{md_1d_2} + \frac{1}{N_SUM_m}\sum_{n=1}^{N} \varphi_{nm}(x_{nd_1} - \mu_{md_1})(x_{nd_2} - \mu_{md_2})^T$ 8. **write_channel:** covariance Θ_{mdd} 9. **for all** d ∈ 1 to D **do** 10. **write_channel:** μ_{md}

G* = GLOBAL, I* = INPUT, C* = CHANNEL, O* = OUTPUT

Second kernel (*Algorithm 3*) reads random float sample dataset and membership value form global memory and sum of all membership value for a given cluster, mean from channel (kernel 1) to calculate covariance of dimension ($d_1$and d_2) for each cluster. We are calculating full or non-diagonal covariance matrix. By doing so, we have to do a more computation O (NMD^2) compared to non-diagonal covariance matrix O(NMD) [2-6] which will consume more FPGA hardware resource usage. However, this will not compromise the shape of Gaussians [22] which leads to better accuracy in results. Kernel 2 sends covariance to kernel 3 and same mean value (for a given cluster) to kernel 4 through channel.

Both Kernel 2 and 4 requires mean value to calculate covariance and E-Step numerator, respectively. So, we need to write mean value to two channels. Logic Element (LE) utilization is same if we write mean value in two channels either on kernel 1 or one channel on kernel 1 and another channel on kernel 2. However, if we write mean value in two channels on kernel 1, FPGA on-chip memory usage increases. To reduce FPGA on-chip memory usage, we send mean values to kernel 2 using one channel. Kernel 2 reads mean value from channel for a given cluster, calculates covariance and sends the mean value to kernel 4 through another channel for that cluster. Data is read from one channel and that same data is written to another channel after computation. By doing so, FPGA on-chip memory usage reduces because channel is FIFO based system.

4.1.3.Pre-Calculation E-Step

Kernel 3 (*Algorithm 4*) reads covariance data from kernel 2 using channel and calculates inverse covariance using LU decomposition and log determinant to get

constant (see *Algorithm 4* for more detail) value. After calculation, Kernel 3 sends inverse covariance and constant value to kernel 4 using channel.

Algorithm 4 (Kernel 3) Calculates: inverse covariance Θ^{-1}_{mdd} and constant $const_m$
G* I*: none. **C* I*:** covariance Θ_{mdd}. **G* O*:** none. **C* O*:** inverse covariance Θ^{-1}_{mdd} and constant $const_m$. 1. **for all** m ∈ 1 to M 2. **for all** d_1 ∈ 1 to D, d_2 ∈ 1 to D **do** 3. **read_channel:** $\Theta_{md_1d_2}$. 4. calculate Θ^{-1}_{mdd} using LU Decomposition. 5. $\|\Theta\|_m$ = calculate log determinant. 6. $const_m = \log(\frac{1}{(2\pi)^{D/2}\|\Theta\|_m^{1/2}})$ 7. **for all** d_1 ∈ 1 to D, d_2 ∈ 1 to D **do** 8. **write_channel:** $\Theta^{-1}_{md_1d_2}$. 9. **write_channel:** $const_m$

G* = GLOBAL, I* = INPUT, C* = CHANNEL, O* = OUTPUT

4.1.4. E-Step

Detail of E-Step calculation is provided in section 3.1. It takes 2 kernels to execute E-Step. Kernel 4 (Algorithm 5) reads sample dataset from global memory and means, weight, constant and inverse covariance from channel to compute the log likelihood of sample dataset for each cluster. Since the membership value is too big to store in FPGA on-chip memory and memory access and computation pattern between kernel 4 (E-Step 1) and kernel 5 (E-Step 2) is complicated we have to store output of log likelihood to global memory. We cannot use channel extension to transfer data from kernel 4 (E-Step 1) to kernel 5 (E-Step 2). However, we used channel to create a starting point of kernel 5 (E-Step 2) by using a start flag. Kernel 4 computes the log likelihood of sample dataset

for each cluster using equation (9) given below. We are not using equation (8) because of numerical instability caused by *exponent* function. Exponent function causes overflow in a 32-bit floating point number with small input. For example e^{90} will easily overflow 32-bit floating point.

$$A = \omega_m; B = \frac{1}{(2\pi)^{D/2}|\Theta|_m^{1/2}} ;$$

$$C = (x_{nd_1} - \mu_{md_1})(x_{nd_2} - \mu_{md_2})^T \Theta_{md_1d_2}^{-1}$$

$$A \times B \times C \quad \textbf{(8)}$$

$$\log(A) + \log(B) - \frac{1}{2}(C) \quad \textbf{(9)}$$

Algorithm 5 (Kernel 4) Calculates: E-Step 1 Numerator calculation
G* I*: none. **C* I*:** inverse covariance Θ_{mdd}^{-1} and constant $const_m$, mean μ_{md} , weight ω_m. **G* O*:** none. **C* O*:** E-Step 1 Numerator calculation $\varphi_{mn_{E_{Step-1}}}$ 1. **for all** m ∈ 1 to M **do** 2.   **for all** d_1 ∈ 1 to D, d_2 ∈ 1 to D **do** 3.   **read_channel:** $\Theta_{md_1d_2}^{-1}$. 4.   **for all** d ∈ 1 to D **do** 5.   **read_channel:** μ_{md} 6.   **read_channel:** $const_m$ 7.   **read_channel:** ω_m 8.   **for all** n ∈ 1 to N **do** 9.   **for all** d_1 ∈ 1 to D, d_2 ∈ 1 to D **do** 10. $h_m = h_m + (x_{nd_1} - \mu_{md_1})(x_{nd_2} - \mu_{md_2})^T \Theta_{md_1d_2}^{-1}$ 11. $\varphi_{mn_{E_{Step-1}}} = -\frac{1}{2} h_m + \log(\omega_m) + const_m$ 12.   **if** (m = M && n = N) **do** 13.   **write_channel:** start_flag_E-Step_2

G* = GLOBAL, I* = INPUT, C* = CHANNEL, O* = OUTPUT

Kernel 5 first waits of start flag from E-Step 1. When start flag is enabled (read from channel), Kernel 5 starts implementation. For member in each cluster, kernel 5 (*Algorithm 4*) reads weighted likelihood (output of Kernel 4) from global memory and converts into fuzzy probability by using equation (11) given below. Since kernel 4 calculates log-likelihood, denominator has to use a large sum of exponents. Equation 11 also helps to avoid overflows. After calculation Kernel 5 writes membership value into global memory.

$$log \sum_{n} \exp(x_n) \equiv \max(x) + log \sum_{n} \exp(x_n - \max(x_n)) \quad \textbf{(10)}$$

Algorithm 6 (Kernel 5) Calculate: E-Step 2 Numerator/ Denominator
G* I*: E-Step 1 Numerator $\varphi_{mn_{E_{Step-1}}}$ **C* I*:** none. **G* O*:** Membership value φ_{nm}. **C* O*:** none 1. **read_channel:** start_flag_E-Step_2 2. **if** (start_flag_E-Step_2) **do** 3. **for all** n ∈ 1 to N **do** 4. $max_m = 0$ 5. **for all** m ∈ 1 to M **do** 6. $max_m = fmax\ (\varphi_{mn_{E_{Step-1}}})$ 7. **for all** m ∈ 1 to M **do** 8. $denom_{sum} = \exp(\varphi_{mn_{E_{Step-1}}},\ max_m)$ 9. $denom_{sum} = max_m + \log(denom_{sum})$ 10. **for all** m ∈ 1 to M **do** 11. $\varphi_{nm} = \exp(\varphi_{mn_{E_{Step-1}}} - denom_{sum})$

G* = GLOBAL, I* = INPUT, C* = CHANNEL, O* = OUTPUT

4.2. Optimization for different Problem Sizes

Based on previous CPU and GPU based implementations, highest dimension achieved is 32 with 32 clusters [2] and highest cluster achieved is 100 with 24 dimensions [22]. For FPGA based implementation [3] highest dimension achieved is 6 and highest cluster achieved is 6. The maximum dimension we could fit on Stratix V A7 FPGA is 4 with 2 clusters and on Arria 10 FPGA, 8 with 2 clusters. The maximum clusters we could fit on Stratix V A7 FPGA are 8 with 2 dimensions and on Arria 10 FPGA, 32 with 2 dimensions. Even with reduced unrolling factor for Stratix V FPGA and Arria 10 FPGA, we cannot fit the design on FPGA after crossing highest number of dimension and/or cluster given above. After crossing the highest number of dimension and/or cluster, AOC fails to fit the design on FPGA because either LE utilization or Memory block utilization or both overflow. Table 3 shows highest number of clusters we could fit in each dimension for Stratix V A7 FPGA and Arria 10 FPGA.

Dimension	Cluster	Stratix V FPGA	Arria 10 FPGA
2	2	✓	✓
2	4	✓	✓
2	8	✓	✓
2	16	X	✓
2	32	X	✓
3	2	✓	✓
3	4	X	✓
3	8	X	✓
3	16	X	✓
4	2	✓	✓
4	4	X	✓
8	2	X	✓

Table 3 Dimension and Cluster achived on Stratix V A7 FPGA and Arria 10 FPGA.

4.3. Summary

In this chapter, we first described the operational flow for EM-GMM implementation using Intel FPGA SDK for OpenCL. We explained computation of each kernel and how each kernel is connected to others through channel extension. We concluded this chapter by describing how many dimensions and/or clusters we can fit for Stratix V FPGA and Arria 10 FPGA.

Chapter 5 Experimental Results

This chapter describes evaluation of proposed design on Intel FPGA SDK for OpenCL. We start by explaining experimental setup and dataset used in this experiment. Then we analyze and compare EM execution time, throughput, throughput/power consumption, accuracy and comparison between results obtained for FPGAs, CPUs and GPUs.

5.1. Experimental Setup

For our experiments we are using Intel FPGA SDK for OpenCL 16.0 [23] as HLS CAD tool. The FPGA board used in this research is Nallatech 385 (Stratix V GX A7) [24] and Nallatech 385A (Arria 10 GX 10AX115) [25]. Nallatech 385 board is connected with Intel(R) Xeon(R) CPU E5-2637 V3 @ 3.50GHz (4 cores) CPU and Nallatech 385A board is connected with two Intel Xeon Processor E5-2620 V4 @ 2.10GHz (8 cores) CPU. Table 4 shows the list of CPUs and FPGAs used in this research and also shows the device code we will be using throughout this chapter. Table 5 represents and compares Nallatech 385 (Stratix V GX A7) and Nallatech 385A (Arria 10 GX 10AX115) board and FPGA specification.

Device Code	Device Description
FA	Nallatech 385A with Arria 10 GX 10AX115 [25]
FS	Nallatech 385 with Stratix V GX A7 [24]
C1	Intel(R) Xeon(R) CPU E5-2637 V3 @ 3.50GHz (4 cores) [26], 32GB DDR3 RAM
C2	2 x Intel Xeon Processor E5-2620 V4 @ 2.10GHz (8 cores) [27] 128GB DDR3 RAM

Table 4 Device list and Description.

Board Name and FPGA	**Nallatech 385 Stratix V GX A7 [24]**	**Nallatech 385A Arria 10 GX 10AX115 [25]**
DDR3 RAM	8GB	8GB
Logical Elements	622K	1150K
Registers	939K	1708K
FPGA memory Block	50Mbits	53Mbits
DSP Block	256	1000
PCIe Bandwidth	25.6 GB/s	8TB/s
Power	≤ 25W	≤ 25W

Table 5 FPGA Device Specification.

5.2. Dataset

For our research we conducted multiple tests with different cluster and dimension sizes. For this research, we generated random floating point data from -1000 to +1000. Each dataset consists of 2^{20} data instances with different dimensions. In Stratix V we could fit a design containing 2^{20} instances with 2 (2, 4, 8 clusters), 3 (2 clusters) and 4 (2 clusters) dimensions. However, on Arria 10 we could fit a design containing 2^{20} instances with 2 (2, 4, 8, 16, 32 clusters), 3 (2, 4, 8, 16 clusters), 4 (2, 4 clusters) and 8 (2 clusters) dimensions. We could not fit more dimensions and clusters due to lack of FPGA hardware resources.

5.3. Power Measurement

We used *Watts up? PRO* to power meter to calculate power consumption [28]. The device provides accurate power consumption of the whole system. Table 7 shows power consumption during idle, program execution and accelerator power consumption for two CPUs and FPGAs. Before installing FPGA board on the systems, idle CPU only power consumption for C1 and C2 was 105.1 W and 112.2 W, respectively. During execution of EM on CPU, average power consumption increased to of C1 and C2 140.3 W and 148.7 W respectively.

When both of the boards are installed in the system, idle power consumption of Nallatech 385A with Arria 10 FPGA (With C2 CPU) and Nallatech 385 with Stratix V FPGA (With C1 CPU) was 133.3 W and 127.6 W, respectively. During execution of EM on of Nallatech 385A with Arria 10 FPGA (With C2 CPU) and Nallatech 385 with Stratix V FPGA (With C1 CPU) average power consumption of Heterogeneous system increased to 137.2 W and 130.2 W, respectively. So, average power consumption of Nallatech 385A with Arria 10 FPGA (no CPU) and Nallatech 385 with Stratix V FPGA (no CPU) for running EM is (137.2-112.3) = 24.9 W and (130.2-105.1) = 25.1 W, respectively.

System	**CPU (C1) Only**	**CPU1 with Stratix V Board**	**CPU (C2) Only**	**CPU2 with Arria 10 Board**
Idle power (Watts)	105.1	127.6	112.3	133.3
Execution power (Watts)	140.3	130.2	148.7	137.2
Accelerator Power (Watts)		25.1		24.9

Table 6 Power Consumption of CPUs and FPGAs.

5.4. Performance analysis

For all EM implementations, number of iterations varies depending on different sample dataset to meet the convergence point. For different EM implementation, different researchers use different sample dataset. So, to compare their results, researchers compared execution time, throughput and number of data instances calculated [3] in each iteration [2-4] [22]. For our research we will do the same. However, based on our dataset, we found that it takes us on average of 100 iterations to reach convergence point. The performance is measured by execution time in milliseconds (ms), throughput in Giga-Floating Point Operations per Second (GFLOPs), power in Watts (W) and throughput per power in GFLOPs/W. To test dataset with different dimensions and cluster sizes on both Arria 10 and Stratix V, we launched host program with dataset using different dimensions and cluster sizes. Automatic testing scripts were used for this purpose.

5.4.1. Performance Results

Figure 9, Figure 11 and Figure 13 show execution time, throughput and throughput/power consumption of computing EM on Arria 10 FPGA and Stratix V FPGA with different dimension and cluster sizes respectively. We only included the computation time. Data transfer time between host and device is ignored because it is negligible at 0.5 ms. Figure 10, Figure 12 and Figure 14 show execution time, throughput and throughput/power consumption of computing EM on Intel(R) Xeon(R) CPU E5-2637 CPU and Intel Xeon Processor E5-2620 CPU with different dimension and cluster sizes, respectively.

In Figure 9-14, different colored lines correspond to different FPGAs (Figure 9, Figure 11 and Figure 13) and CPUs (Figure 10, Figure 12 and Figure 14) with different kernels using different dimensions and horizontal axis corresponds to cluster sizes. For Figure 9 and Figure 10, vertical axis represents execution time spent during calculation. For Figure 11 and Figure 12, vertical axis represents throughput. Lastly, in Figure 13 and Figure 14 vertical axis represents throughput/power consumption.

From Figure 9, we can clearly see that for same calculation FS (Stratix V FPGA) consumes more time to compute than FA (Arria 10 FPGA). However, for CPU implementation, Figure 10 shows for same calculation C1 (Intel(R) Xeon(R) CPU E5-2637 CPU) consumes less time to compute than C2 (Intel Xeon Processor E5-2620 CPU).

Figure 11 and Figure 12 show that for EM calculation FA and C1 has higher throughput compared to FS and C2 for same calculation. Figure 13 shows FA has higher throughput/power consumption compared to FS for same calculation. Note that FA and FS power consumption is almost same. Figure 14 shows that C1 has higher throughput/power consumption compared to C2 for same calculation though C2 has higher power consumption than C1.

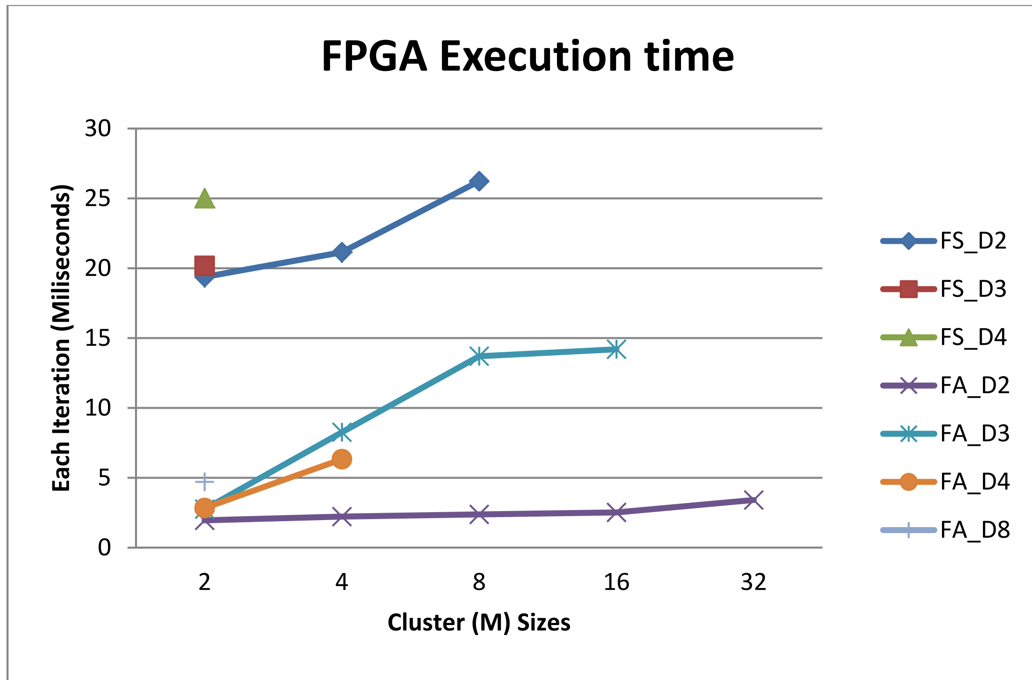

Figure 9 FPGA Execution Time for EM.

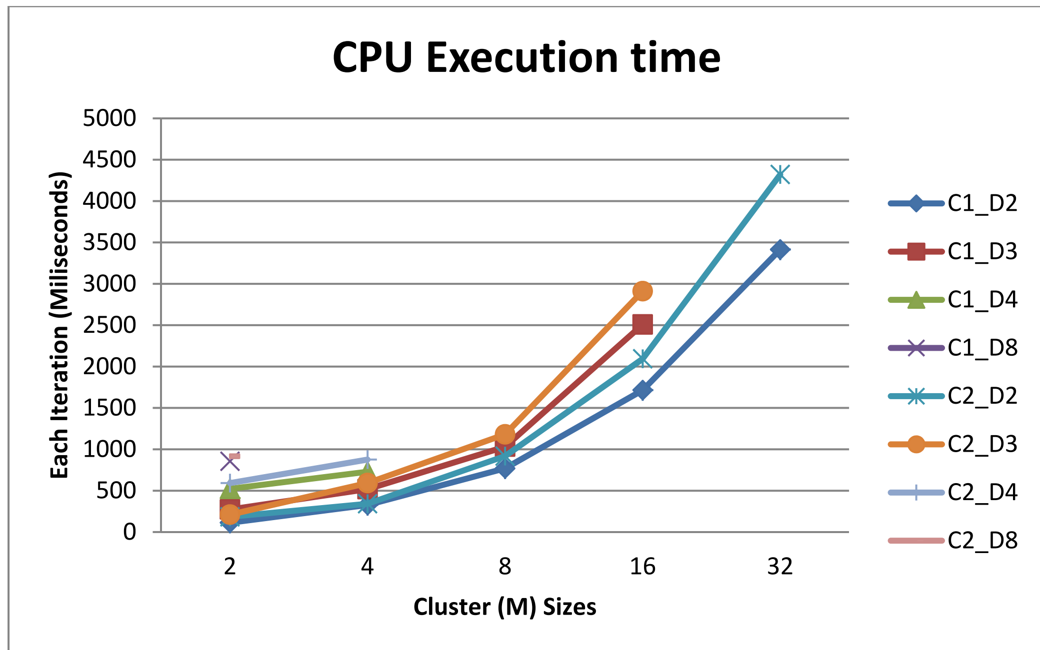

Figure 10 CPU Execution Time.

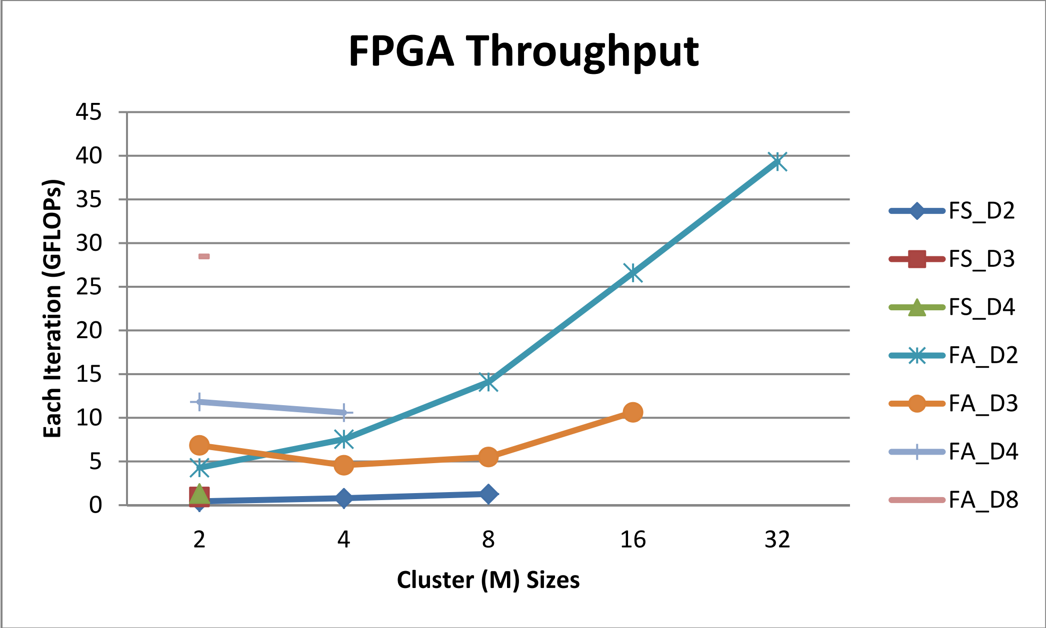

Figure 11 FPGA Throughput for EM.

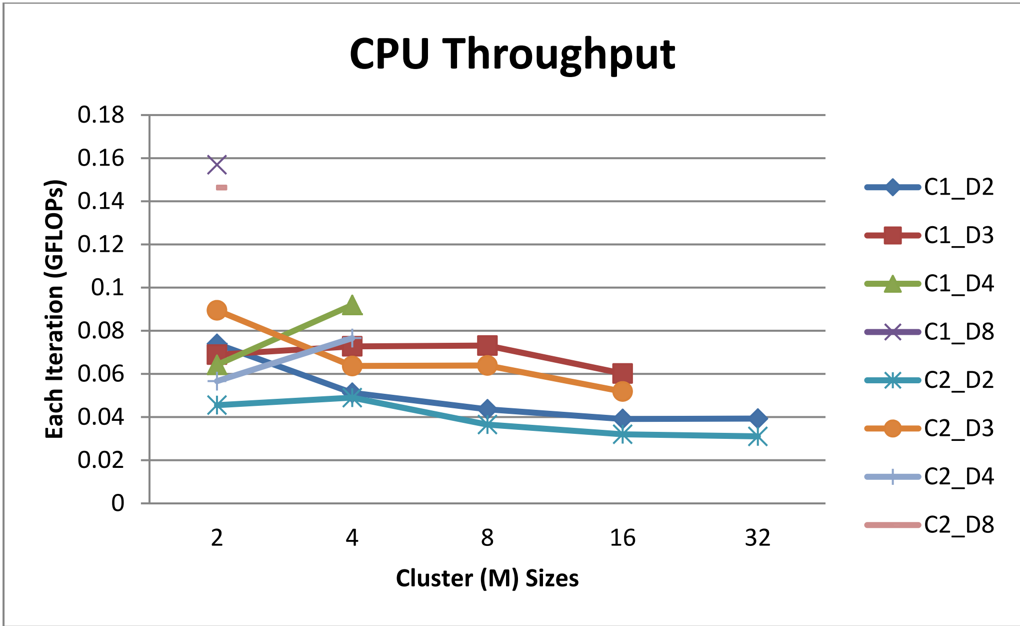

Figure 12 CPU Throughput for EM.

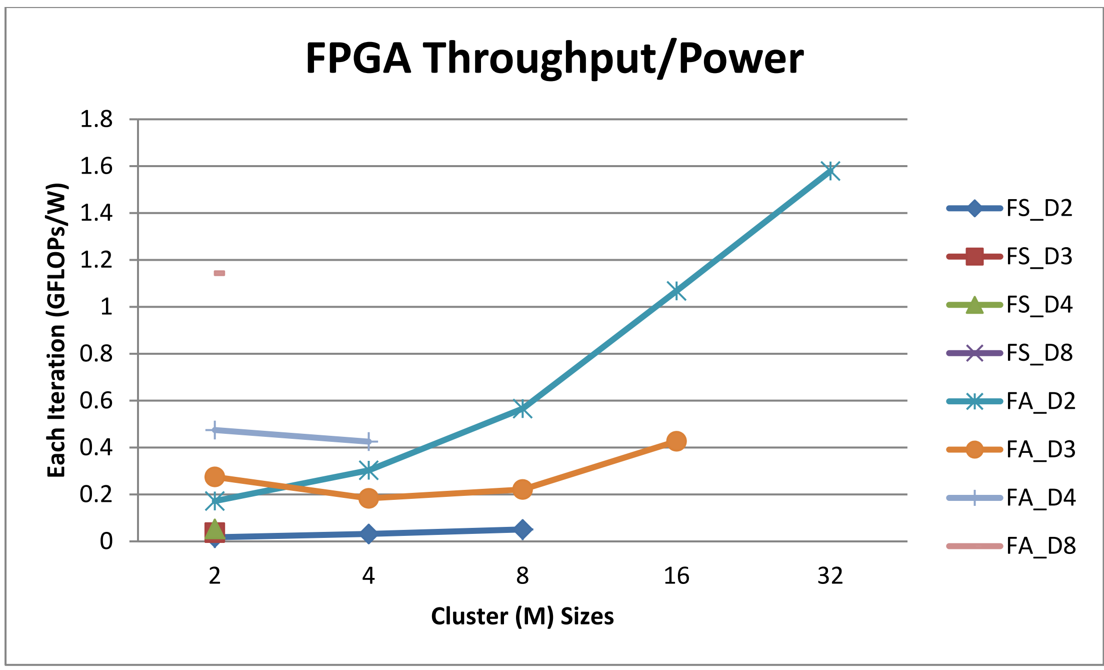

Figure 13 FPGA Throughput/Power.

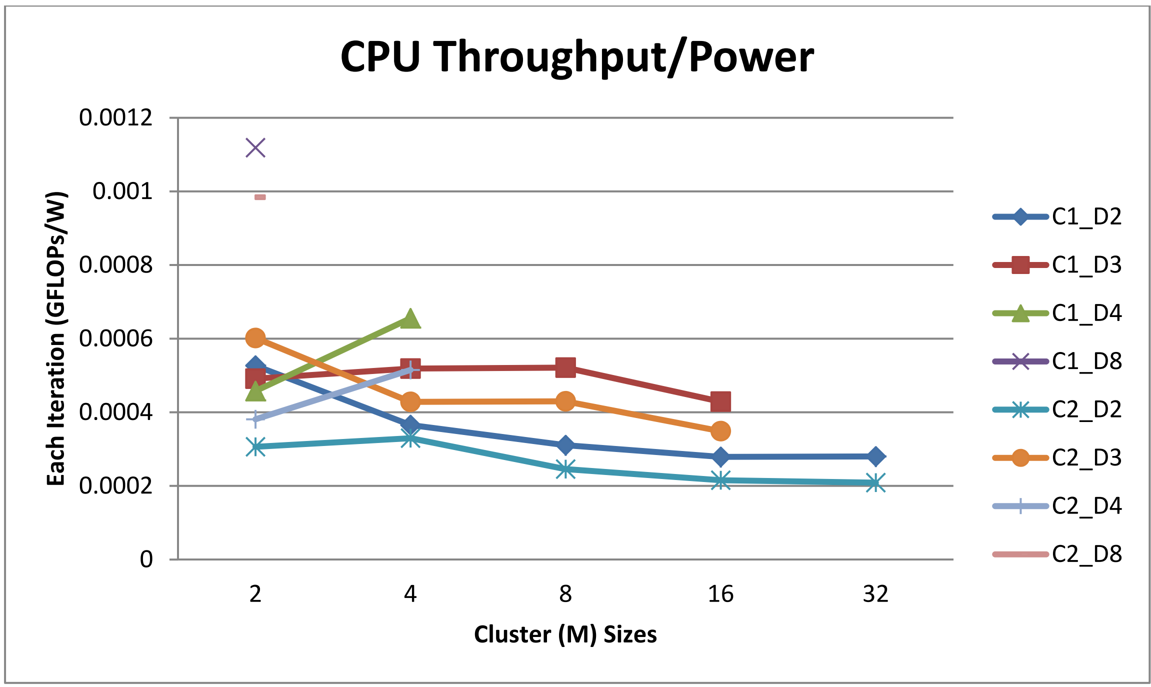

Figure 14 CPU Throughput/Power.

5.4.2. Speedup

In this section we present speedup obtained by each device compared to others. In the first section, we show speedup gained in terms of throughput and in the second section we show speedup gained in terms of throughput/power consumption.

5.4.2.1. Throughput

In Figure 15-18, different colored lines corresponds to different kernels using different dimensions and horizontal axis corresponds different cluster sizes and vertical axis corresponds to speedup gained during calculation in terms of throughput.

Figure 15 shows speedup gained by CPU1 against CPU2. Figure 16 and Figure 17 show speedup gained by Stratix V FPGA and Arria 10 FPGA against CPUs, respectively. Figure 18 shows speedup gained by Arria 10 FPGA against Stratix V FPGA.

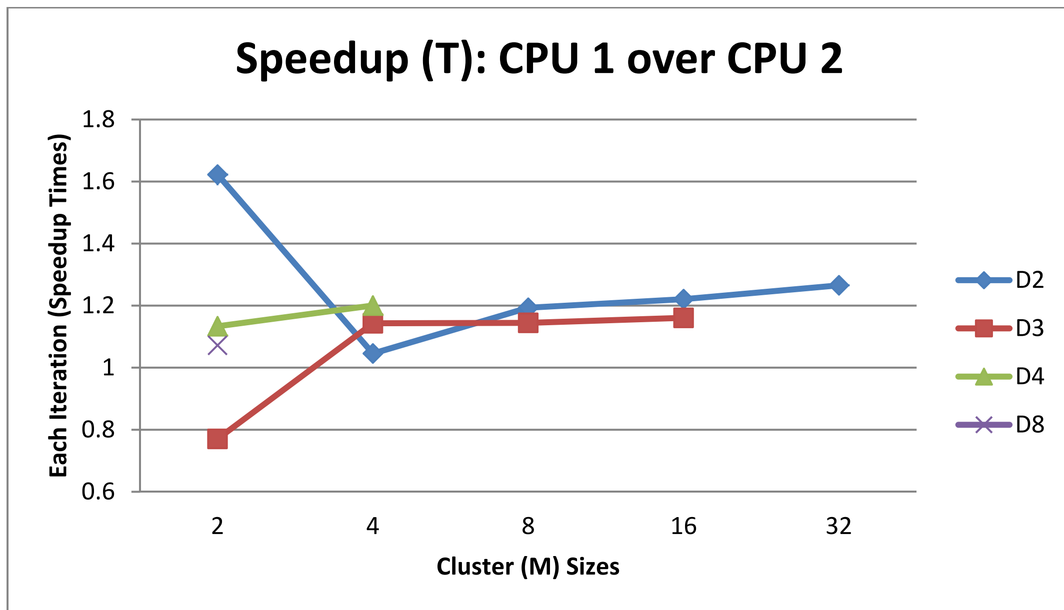

Figure 15 CPU1 Speedup (T) over CPU2.

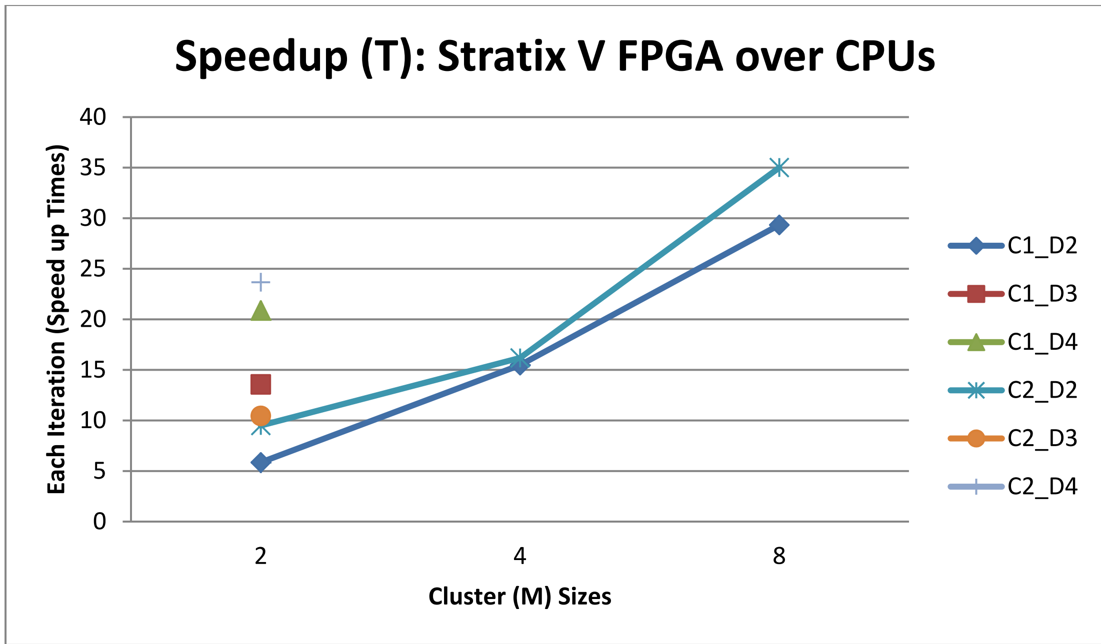

Figure 16 Stratix V FPGA Speedup (T) over CPUs.

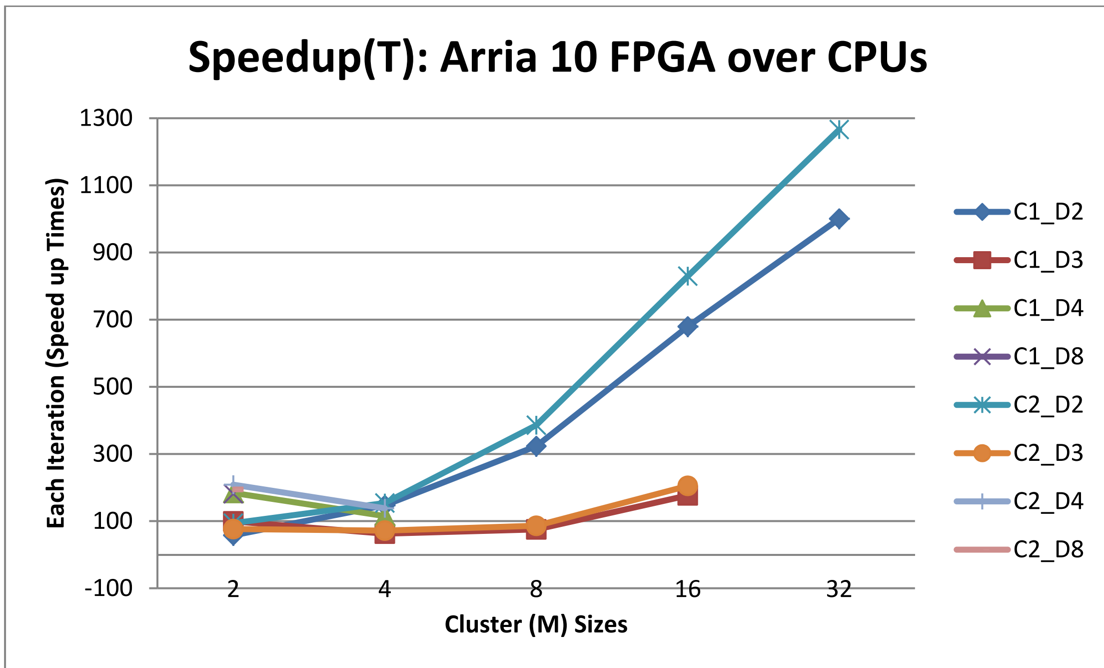

Figure 17 Arria 10 FPGA Speedup (T) over CPUs.

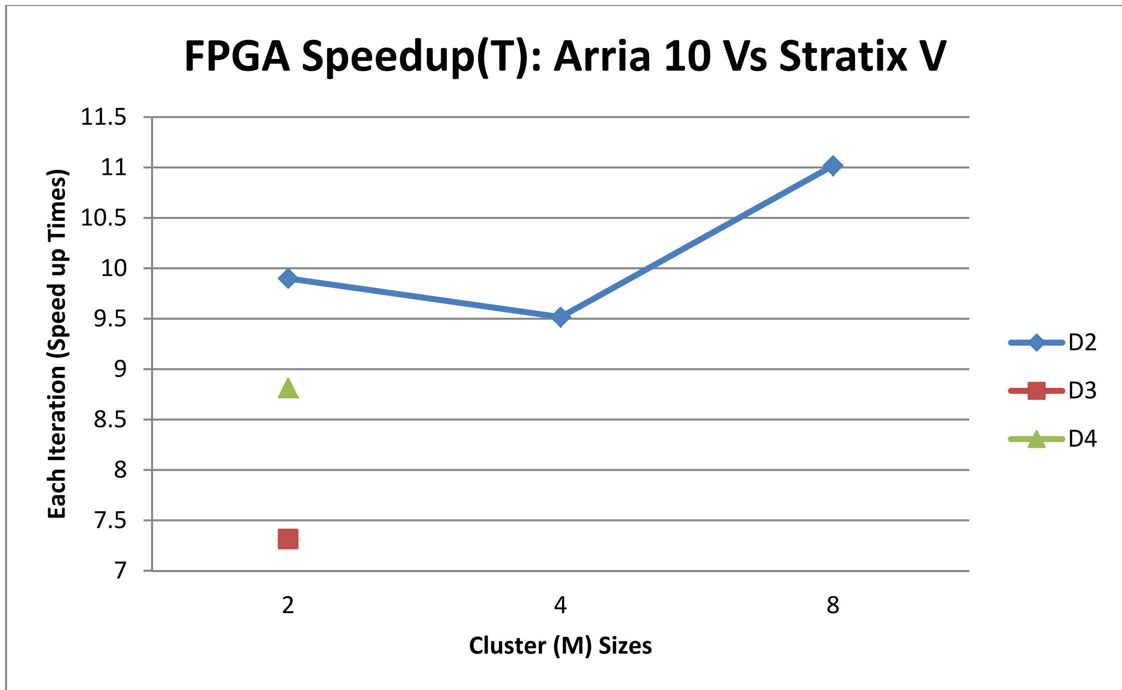

Figure 18 Arria 10 FPGA Speedup (T) overs Stratix V FPGA

5.4.2.2. Throughput/Power

In Figure 19-22, the colored lines correspond to different kernels using different dimensions and horizontal axis represents different cluster sizes and vertical axis represents speedup in terms of throughput/power consumption.

Figure 19 shows speedup gained by CPU1 against CPU2. Figure 20 and Figure 21 show speedup gained by Stratix V FPGA and Arria 10 FPGA against CPUs, respectively. Figure 22 shows speedup gained by Arria 10 FPGA against Stratix V FPGA.

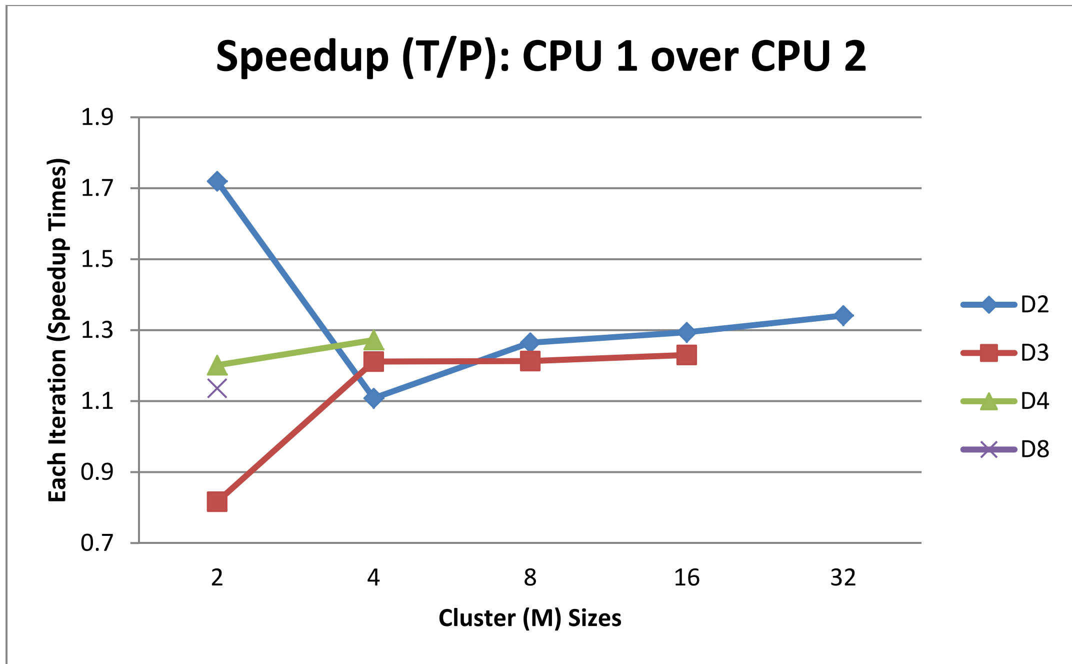

Figure 19 CPU 1 Speedup (T/P) over CPU 2

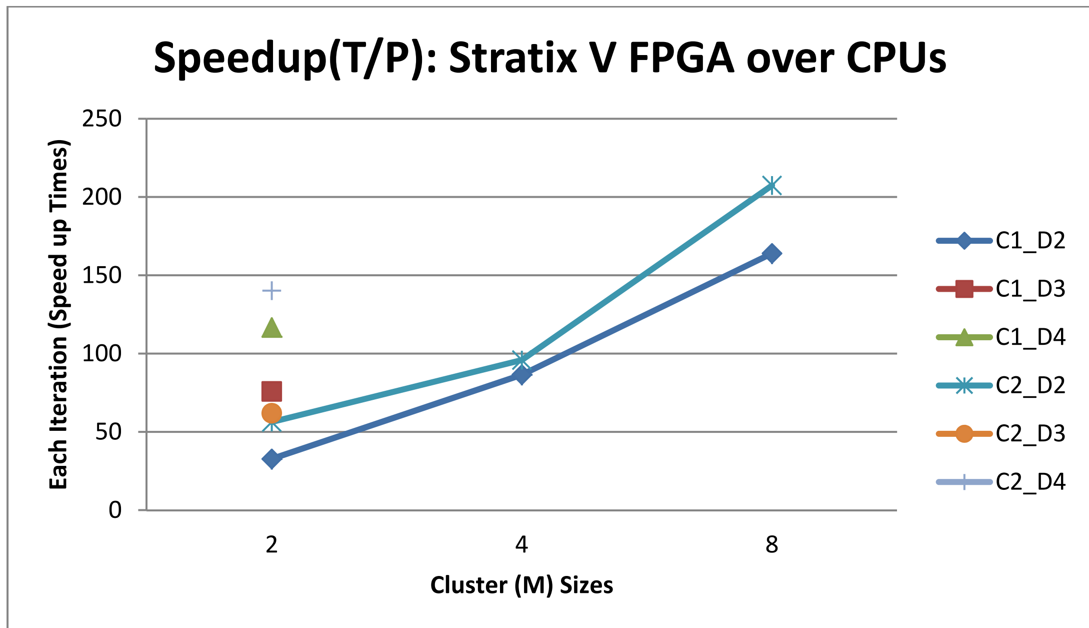

Figure 20 Stratix V FPGA Speedup (T/P) over CPUs.

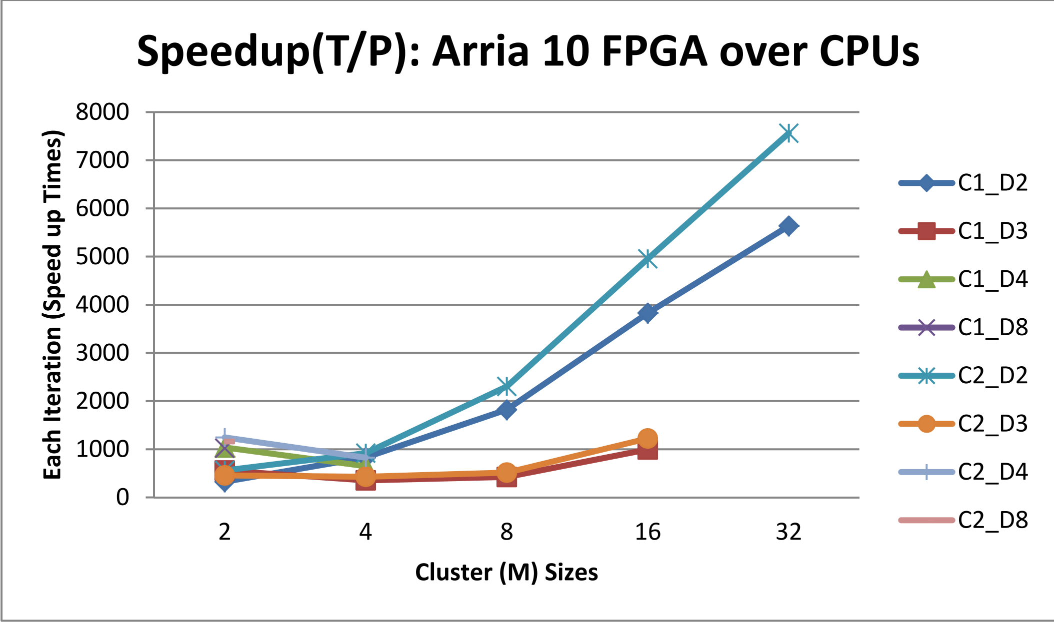

Figure 21 Arria 10 FPGA Speedup (T/P) over CPUs.

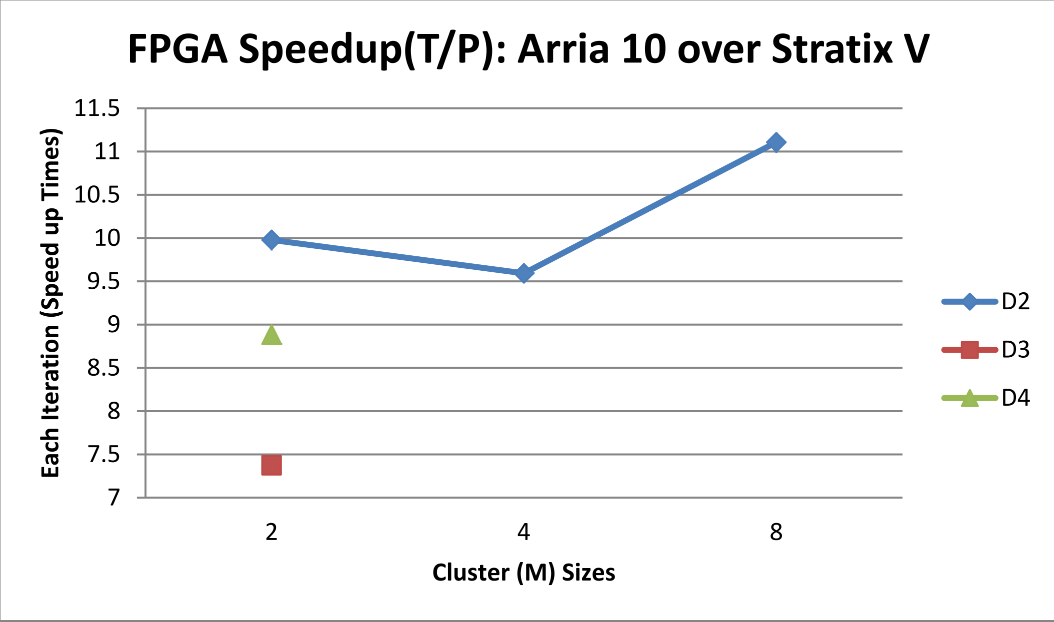

Figure 22 Arria 10 FPGA Speedup (T/P) over Stratix V FPGA.

5.4.3.FPGA Resource Utilization

This section shows FPGA resource utilization (Logic Element, Register, Memory Block, DSP and Adaptive Look-Up Tables -ALUT) and operational frequency of each FPGA for each kernel with different dimensions and clusters. For each implementation on FPGA, we got resource usage and operational frequency from synthesis report. To fit the design on FPGA for each implementation with higher dimensions and/or clusters, we had to decrease loop unroll factor or remove loop unroll for some and/or all loop inside each kernel. For implementations with higher dimensions and/or clusters we also had to reduce the memory block size to cache data into FPGA on-chip memory from global memory (DDR3 memory). For this reason, for some implementations, FPGA resource usage and operational frequency drops down.

Figure 23, Figure 24, Figure 25, Figure 26 and Figure 27 show FPGA Logic Element, ALUT, Register, Memory Block, DSP block utilization, respectively. Figure 28 shows FPGA operational frequency.

In Figure 23-27, different colored lines correspond to different kernels using different dimensions and horizontal axis represents different cluster sizes. From Figure 23 to Figure 27, vertical axis represents resource usage of each implementation for each FPGA, as percentage. In Figure 28, vertical axis represents operational frequency of each implementation for each FPGA in Megahertz (MHz). From Figure 23 to Figure 27 we can see that for each implementation Stratix V FPGA consumes more percentage resources than Arria 10 FPGA, because of smaller logic capacity of Stratix V.

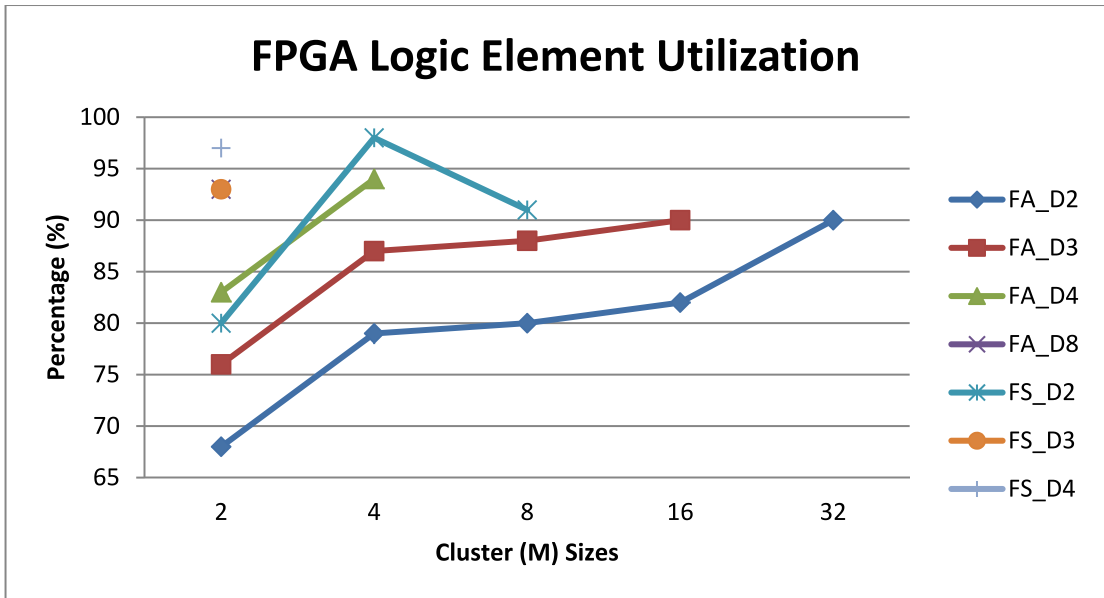

Figure 23 FPGA Logic Element Utilization.

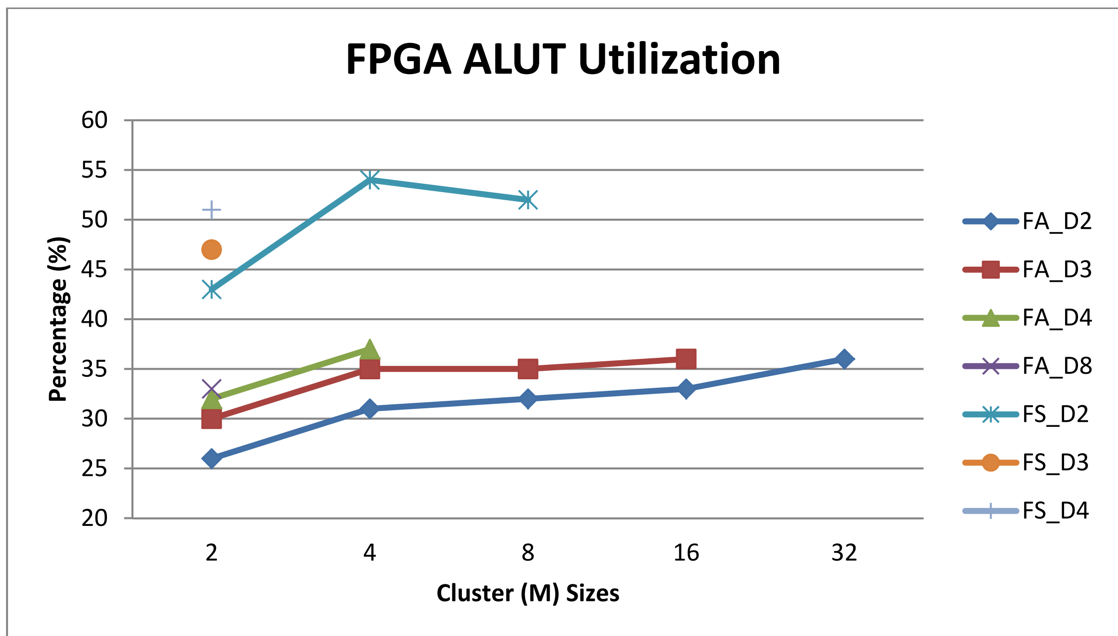

Figure 24 FPGA ALUT Utilization.

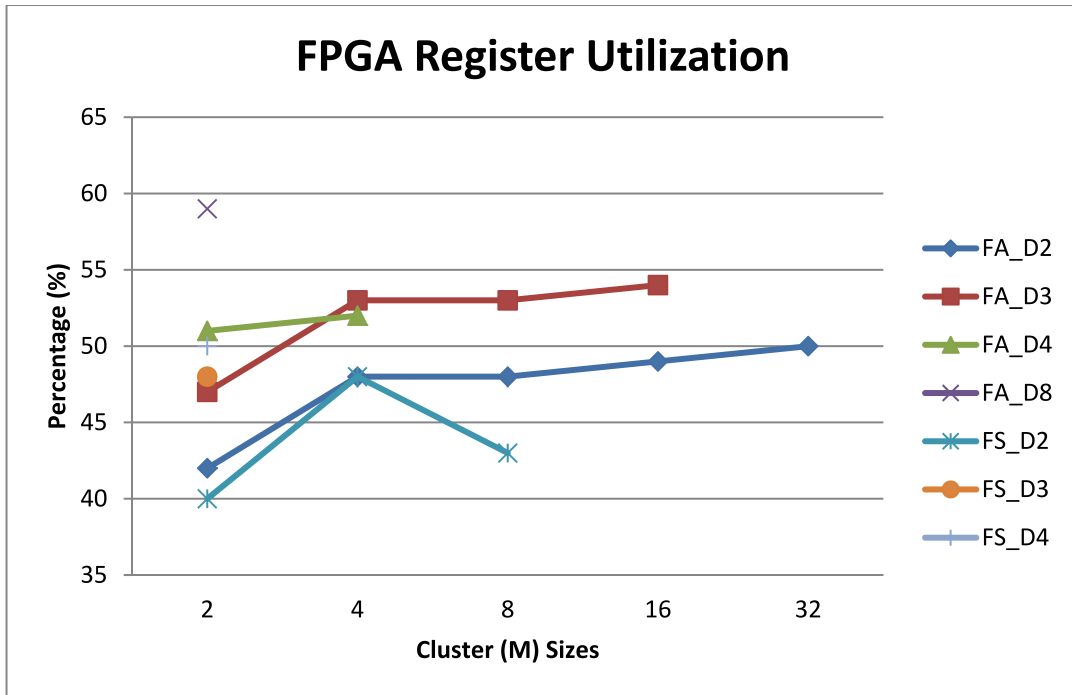

Figure 25 FPGA Register Utilization.

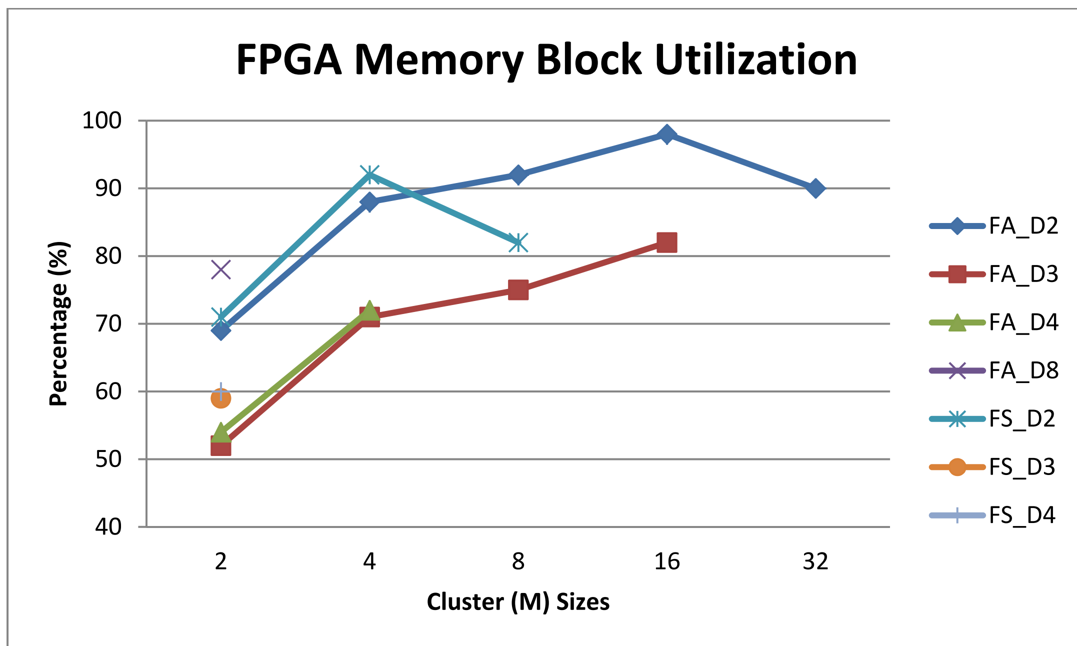

Figure 26 FPGA Memory Block Utilization.

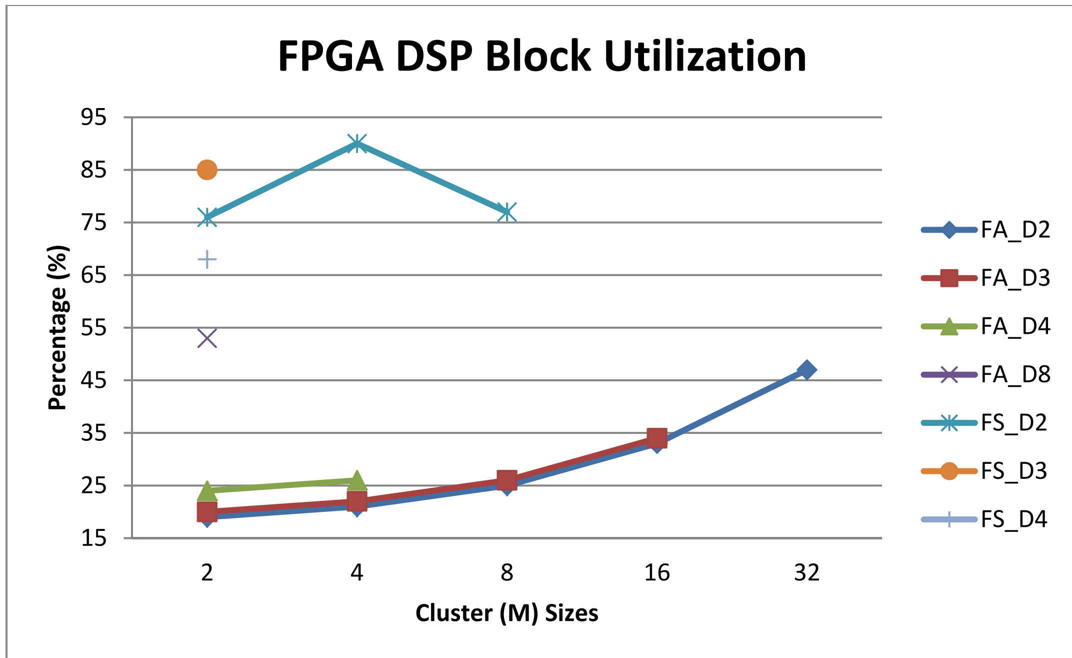

Figure 27 FPGA DSP Block Utilization.

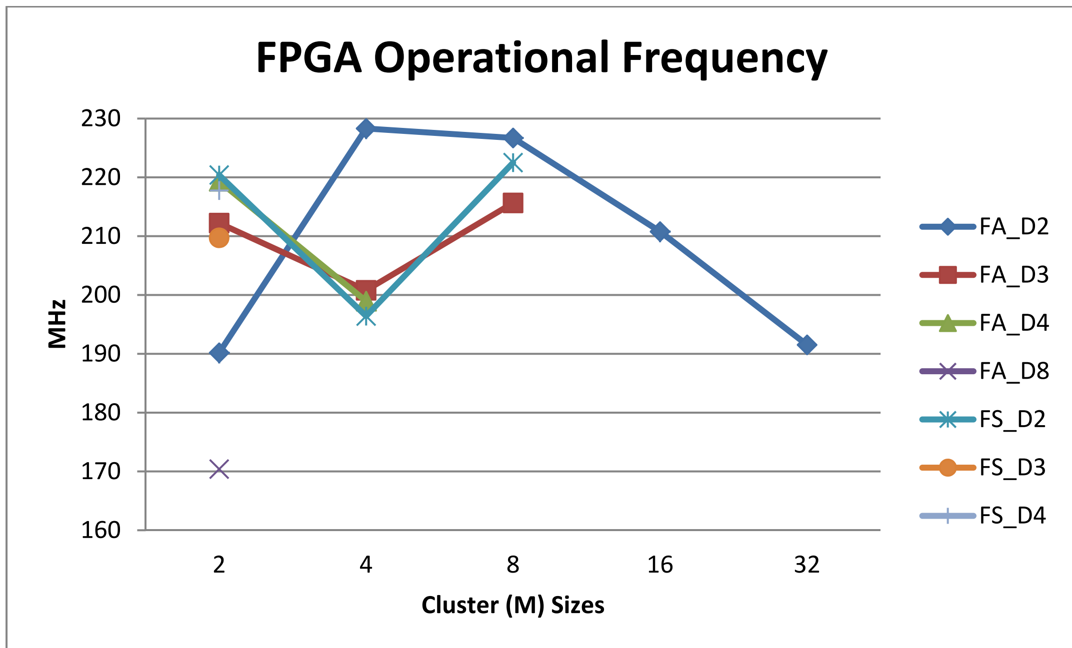

Figure 28 FPGA Operational Frequency.

5.4.4. Performance Comparison Between CPU and FPGA in Relation to FPGA Resource Utilization

Table 7 shows performance comparison between FPGAs and CPUs in relation to FPGA resource usage for highest number of cluster achieved in each dimension by FPGAs. Yellow cells shows highest number of clusters we could fit in each dimension. Pink and green cells represent throughput and throughput/power consumption of each device, respectively. Blue and purple cells show speedup in terms of throughput and speedup in terms of throughput/power consumption gained by FPGAs over CPUs, respectively. Light orange cells shows FPGA resource utilization for highest number of cluster we could fit in each dimension. Lastly, dark orange cells shows FPGA operational frequency for highest number of cluster we could fit in each dimension. This table clearly shows that Arria 10 FPGA achieved much better speedup than Stratix V FPGA, when compared to CPU performance. Both FPGAs consumed most of their hardware resources for highest number of clusters achieved by each dimension.

Table 9 shows performance comparison between FPGAs and FPGA resource usage for highest number of clusters achieved in each dimension by Stratix V FPGA. Leftmost column shows the parameters we are using to compare two FPGAs. Green and red cells represent performance, resource utilization and operational frequency of Arria 10 FPGA and Stratix V FPGA, respectively. Orange cells represent speedup gained by Arria 10 FPGA over Stratix V FPGA for highest number of cluster achieved in each dimension by Stratix V FPGA. Table 8 is similar to Table 7 except that it shows $T/P_{speedup}$ instead of $T_{speedup}$.

FPGAs	Arria 10				Stratix V		
Dimensions	2	3	4	8	2	3	4
Max_m^*	32	16	4	2	8	2	2
CPU1 T*	0.04	0.06	0.09	0.16	0.04	0.07	0.06
CPU2 T*	0.03	0.05	0.08	0.15	0.04	0.06	0.06
FPGA T*	39.33	10.63	10.59	28.48	1.28	0.94	1.34
$SU_{F/C1}$ T*	1000.68x	176.64x	115.19x	181.48x	29.34x	13.57x	20.88x
$SU_{F/C2}$ T*	1266.37x	205.00x	138.28x	194.57x	35.01x	10.46x	23.67x
CPU1 T/P*	2.91E-4	4.36E-4	6.54E-4	1.16E-3	2.91E-4	5.09E-4	4.36E-4
CPU2 T/P*	2.02E-4	3.36E-4	5.38E-4	1.01E-3	2.69E-4	4.03E-4	4.17E-4
FPGA T/P* [AO*]	1.57	0.42	0.42	1.13	0.05	0.04	0.05
$SU_{F/C1}$ T/P* [AO*]	5395.19x	963.30x	642.20x	974.14x	171.82x	78.57x	114.68
$SU_{F/C2}$ T/P* [AO*]	7772.28x	1250x	780.67x	1118.81x	185.87x	99.26x	119.90x
Logic U*	90%	90%	94%	93%	91%	93%	97%
ALUTs U*	36%	36%	37%	33%	52%	47%	51%
Register U*	50%	54%	52%	59%	43%	48%	50%
MB* U*	90%	82%	72%	78%	82%	59%	60%
DSP U*	47%	34%	26%	53%	77%	85%	68%
Frequency M*	191.53	215.65	198.96	170.41	222.51	290.73	217.77

Table 7 Performance comparison of FPGAs over CPUs and FPGA Resource Usage for highest cluster achieved by each dimension.

T* = Throughput (GLOPs), T/P*= Throughput/Power Consumption (GFLOPs/W), AO* = Accelerator only, U* = Utilization, M*= (Mhz), MB* = Memory Block, Max_m^* = Maximum size of cluster.

Dimension	2		4		8	
Max$_m$ FS*	8		2		2	
FPGAs	FA*	FS*	FA*	FS*	FA*	FS*
Throughput (GFLOPs)	14.09	1.28	6.84	0.94	11.82	1.344
T* Speed Up$_{FA/FS}$ [Arria 10]	11.02x		7.32x		8.81x	
[AO*] T/P* (GFLOPs/W)	0.56	0.05	0.27	0.04	0.47	0.05
T/P* Speed Up$_{FA/FS}$ [Arria 10]	10.92x		7.21x		8.72x	
Logic U*	80%	91%	76%	93%	83%	97%
ALUTs U*	32%	52%	30%	47%	32%	51%
Register U*	48%	43%	47%	48%	51%	50%
Memory Block U*	92%	82%	52%	59%	54%	60%
DSP U*	25%	77%	20%	85%	24%	68%
Frequency (Mhz)	226.70	222.51	212.22	290.73	219.34	217.77

Table 8 Performance comparison of FPGAs and FPGA Resource Usage for highest cluster achieved by each dimension.

FA* = FPGA Arria 10, FS* = FPGA Stratix V, T = Throughput, T/P*= Throughput/Power Consumption, AO* = Accelerator only, T* = Throughput (GLOPs), U* = Utilization

5.4.5. Comparison with Previous EM Research

We compare our performance with previous EM implementations by using the time it takes to compute all data instances in a single iteration (in seconds). This method was also used by [3] to compare their performance with previous research. Table 9 shows instructions per second (IPS) processed by previous research (green cells), Arria 10 FPGA (light blue cells), Stratix V FPGA (light red cells) and speedup gained by Arria 10 FPGA (dark blue cells) and Stratix V FPGA (dark red cells) compared with previous research.

Previous Research			**Our Research (Fl*)**			
Hardware (Accelerator)	**DF***	**IPS***	**FA* IPS***	**FS* IPS***	**SU_{FA}**	**SU_{FS}**
Xilinx Virtex-6 [3]	Fi*	1.493E+8	1.141E+9	1.559E+8	7.64x	1.04x
		1.492E+8	3.808E+8	-	2.55x	-
Geforce 8800 ULTRA [2]	Fl*	2.008E+7	1.141E+9	1.559E+8	56.82x	7.76x
		2.903E+7	1.478E+9	1.676E+8	50.91x	5.77x
Quadro FX 5800 [2]		2.887E+7	1.141E+9	1.559E+8	39.62x	5.41x
		2.283E+7	1.478E+9	1.676E+8	64.74x	7.34x
		2.783E+7	1.780E+9	-	63.96x	-
NVidia GTX480 [4]		2.796E+8	1.780E+9	-	6.67x	-
I* NVidia GTX480 [4]		1.367E+8	1.780E+9	-	13.03x	-

Table 9 IPS*and speedup gained by our implementation over other implementations.

IPS*= Instances Per Second, P*= Power Consumption, I* = Implementation of [22] code on NVidia GTX480 [4], Fl* = Floating Point, Fi* = Fixed Point, DF* = Dataset Format, FA* = Arria 10 FPGA, FS* = Stratix V FPGA, SU = Speed up

Since we are also considering power consumption as an evaluation metric for each device in Table 11 we show IPS/power consumption of each device used in previous

research and our research. And also speedup gained by our implementation over other implementations in terms of IPS/power consumption. Table 10 is same as Table 9 with the difference of IPS/power consumption and orange cells which represents power consumption of each accelerator used in previous research. We found power consumption of all GPU based accelerators. We did not find any power consumption report for Maxeler Max3 with Xilinx Virtex-6 FPGA [3]. From our implementation and implementation done in different FPGA based accelerators for different implementations [19][32] we found that average power consumption of FPGA based accelerators is 25 W. So, it is safe to assume power consumption of Maxeler Max3 with Xilinx Virtex-6 FPGA [3] is 25 W.

PR*= Previous Research				Our Research (Fl*)			
Hardware (Accelerator)	P* (W)	DF*	IPS*/P*	FA* IPS*/P*	FS* IPS*/P*	SU_{FA}	SU_{FS}
Xilinx Virtex-6 [3]	25	Fi*	5.975E+6	4.58E+7	6.21e+6	7.66x	1.04x
			4.768E+6	1.52E+7	-	3.18x	-
Geforce 8800 ULTRA [2]	172.2 [29]	Fl*	1.17E+05	4.55E+07	6.26E+06	388.89x	53.50x
			1.69E+05	5.89E+07	6.73E+06	348.52x	39.82x
Quadro FX 5800 [2]	189 [30]		1.53E+05	4.55E+07	6.26E+06	297.39x	40.92x
			1.21E+05	5.89E+07	6.73E+06	486.78x	55.62x
			1.47E+05	7.09E+07	-	482.31x	-
NVidia GTX480 [4]	223 [31]		1.25E+06	7.09E+07	-	283.6x	-
I*NVidia GTX480 [4]			6.10E+05	7.09E+07	-	116.23x	-

Table 10 IPS*/power consumption and speedup gained by our implementation over other implementations.

IPS*= Instances Per Second, P*= Power Consumption, I* = Implementation of [22] code on NVidia GTX480 [4], Fl* = Floating Point, Fi* = Fixed Point, DF* = Dataset Format, FA* = Arria 10 FPGA, FS* = Stratix V FPGA, SU = Speed up

5.5. Verification of Results

A sequential version of EM-GMM algorithm was implemented in CPU to ensure the accuracy of EM-GMM algorithm running on FPGA based accelerators. This implementation was done after FPGA implementation to generate reference results. After FPGA implementation, output membership data of FPGA is copied from global memory to host memory to compare against CPU reference results. We used Mean square error (MSE) [33] to estimate the difference between FPGA output and CPU reference output just like [22]. We found that both Stratix V FPGA and Arria 10 FPGA implementation had the same MSE rate for implementations with same dimension and cluster. For both Stratix V FPGA and Arria 10 FPGA lowest error rate was 4.78E-7 and highest error rate was 9.46E-5. This shows that the error range is quite acceptable.

5.6. Summary

In this chapter we first explained out experimental setup and the dataset used in our research. Then we presented our experimental results and their analysis. First, we showed that both the FPGAs used in our research consume almost the same amount of power. Though both FPGAs achieved significant speedups compared to CPUs, Arria 10 FPGA obtained much better speedup compared to Stratix V FPGA. When comparing the performance of FPGAs and GPUs for implementing EM, both FPGAs obtained significant speedup. Lastly, experimental results show that FPGA implementation of EM has better accuracy when compared with CPU implementation.

Chapter 6 Conclusion

Our research focused on optimized FPGA based implementation of EM-GMM using Intel FPGA SDK for OpenCL. We had to restructure the operational flow of EM to properly take advantage of channel extension inside Intel FPGA SDK for OpenCL. By using the channel extension to transfer data between kernels we reduced communication bottleneck caused by global memory. We also restructured covariance matrix calculation algorithm which is one of the biggest computational part of EM-GMM algorithm. All of these necessary changes helped up to gain better throughput and throughput/power consumption compared to different CPUs and different accelerators (GPUs and FPGAs). Due to limited LEs and FPGA on-chip memory we could only fit lower dimensions and/or clusters. To fit EM with higher dimensions and/or clusters, we need accelerators with multiple FPGAs and/or FPGAs with larger number of LEs embedded on-chip memories.

Compared to an Intel(R) Xeon(R) CPU E5-2637 our fully optimized OpenCL model for EM targeting Arria 10 FPGA achieved up to 1000X speedup in terms of throughput (${}^{T}{}_{speedup}$) and 5395X speedup in terms of throughput per unit of power consumed (${}^{T/P}{}_{speedup}$). Compared to previous research on EM-GMM implementation on GPUs, Arria 10 FPGA obtained up to 64.74X ${}^{T}{}_{speedup}$ and 486.78X ${}^{T/P}{}_{speedup}$.

6.1. Future Work

In this research, we used single chip FPGA based accelerators. Stratix V FPGA had limited resources compared to Arria 10 FPGA. It would be interesting to see how many dimensions and/or clusters we can fit on Stratix 10 FPGA based accelerator [34] since Stratix 10 FPGA has more hardware resources than Arria 10 FPGA. Nallatech released an FPGA based accelerator with dual Arria 10 FPGA chips [35]. Firstly, it would be interesting to see how to program dual Arria 10 FPGA chip using OpenCL model and then how many dimensions and/or clusters we can fit for our particular design. Since, EM has two steps and if we could put each step on each Arria 10 FPGA, how many dimensions and/or clusters we could fit. It would be also interesting to see how dual Arria 10 FPGA affects execution time, throughput, power consumption, throughput/power consumption and accuracy of the design compared to single Arria 10. Lastly, exploring multi-FPGA systems as targets for Intel SDK for OpenCL would be a very interesting research project.

References

[1] Geoffrey J. McLachlan, Thriyambakam Krishnan, The EM Algorithm, and Extensions, Wiley Series in Probability and Statistics, John Wiley and Sons, 1997.

[2] N. Kumar, S. Satoor, and I. Buck, “Fast parallel expectation maximization for Gaussian mixture models on GPUs using CUDA,” in IEEE International Conference on High-Performance Computing and Communications. IEEE, 2009, pp. 103–109.

[3] Guo, C., Fu, H., & Luk, W. (05 June 2017). A Fully-Pipelined Hardware Design for Gaussian Mixture Models. *IEEE Transactions on Computers (Volume: PP, Issue: 99)*

[4] Can Altinigneli, M., Plant, C., & Böhm, C. (2013). Massively parallel expectation maximization using graphics processing units. *ACM SIGKDD international conference on Knowledge discovery and data mining*, 838-846.

[5] Choi, Y., Cong, J., & Wu, D. (May 2014). FPGA Implementation of EM Algorithm for 3D CT Reconstruction. *Field-Programmable Custom Computing Machines (FCCM)*.

[6] Choi, Y., & Cong, J. (JUNE 2016). Acceleration of EM-Based 3D CT Reconstruction Using FPGA. *IEEE Transactions on Biomedical Circuits and Systems*, VOL. 10, NO. 3.

[7] F. A. Escobar, X. Chang, and C. Valderrama, \Suitability Analysis of FPGAs for Heterogeneous Platforms in HPC," IEEE Transactions on Parallel and Distributed Systems, vol. 27, no. 2, pp. 600{612, Feb 2016.

[8] R. Inta, D. J. Bowman, and S. M. Scott, \The "Chimera": An Off-the-shelf CPU/GPGPU/FPGA Hybrid Computing Platform," Int. J. Recong. Comput., vol. 2012, pp. 2:2{2:2, Jan. 2012.

[9] I. Kuon and J. Rose, "Measuring the gap between FPGAs and ASICs," IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems, vol. 26, no. 2, pp. 203{215, Feb 2007.

[10] "Top FPGA companies for 2013," http://sourcetech411.com/2013/04/top-fpga-companies-for-2013/, Apr 2013, [Online; accessed Jun 28, 2017].

[11] "Arria 10 architecture," https://www.altera.com/products/fpga/arria-series/arria-10/features.html, 2016, [Online; accessed Jun 28, 2017]

[12] R. Nane, V. M. Sima, C. Pilato, J. Choi, B. Fort, A. Canis, Y. T. Chen, H. Hsiao, S. Brown, F. Ferrandi, J. Anderson, and K. Bertels, "A survey and evaluation of FPGA high-level synthesis tools," IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems, vol. 35, no. 10, pp. 1591{1604, Oct 2016.

[13] "Conformant companies," https://www.khronos.org/opencl/,2017, [Online; accessed Jun 28, 2017].

[14] A. Munshi, "The OpenCL specification 1.2," https://www.khronos.org/registry/OpenCL/specs/opencl-1.2.pdf, 2012, [Online; accessed Jun 28, 2017].

[15] K. Morris, “The path to acceleration: Altera bets on OpenCL," http://www.eejournal.com/archives/articles/20121106-opencl/, Nov 2012, [Online; accessed Jun 28, 2017].

[16] “OpenCL on FPGAs for GPU programmers," http://design.altera.com/openclforward, Acceleware Corp., 2014, [Online; accessed Jun 28, 2017].

[17] E. Rucci, C. Garcia, G. Botella, A. E. D. Giusti, M. Naiouf, and M. PrietoMatias, \OSWALD: OpenCL SmithWaterman on Altera's FPGA for Large Protein Databases," The International Journal of High-Performance Computing Applications, June 2016.

[18] S. O. Settle, “High-performance dynamic programming on FPGAs with OpenCL," in Proc. IEEE High Perform. Extreme Comput. Conf.(HPEC), 2013, pp. 1-6.

[19] Li, H. (2017, April 14). ELECTRONIC THESES AND DISSERTATIONS. *Acceleration of Deep Learning on FPGA.*Retrieved June 28, 2017, from http://scholar.uwindsor.ca/etd/5947/

[20] "Intel FPGA SDK for OpenCL best practices guide," https://www.altera.com/content/dam/altera-www/global/en US/pdfs/literature/hb/opencl-sdk/aocl-best-practices-guide.pdf, 12 2016, [Online; accessed Jan 2, 2017].

[21] G. Gan, C. Ma, and J. Wu, Data Clustering Theory, Algorithms, and Applications, M. T. Wells, Ed. Society for Industrial and Applied Mathematics, 2007.

[22] Pangborn, Andrew D., "Scalable data clustering using GPUs" (2010). Thesis. Rochester Institute of Technology. Accessed from http://scholarworks.rit.edu/theses/5464, [Online; accessed Jun 2, 2017].

[23] (n.d.). Intel FPGA and SoC. *Intel FPGA SDK for OpenCL - Overview.* Retrieved July 1, 2017, from http://www.altera.com/products/design-software/embedded-software-developers/opencl/overview.html

[24] (n.d.). Nallatech . *Nallatech 385 – with Stratix V A7 FPGA.*Retrieved July 1, 2017, from http://www.nallatech.com/store/pcie-accelerator-cards/385-a7/

[25] (n.d.). Nallatech. *Nallatech 385A FPGA Accelerator Card.*Retrieved July 1, 2017, from http://www.nallatech.com/store/fpga-accelerated-computing/pcie-accelerator-cards/nallatech-385a-arria10-1150-fpga/

[26] (n.d.). Intel. *Intel® Xeon® Processor E5-2637.* Retrieved July 1, 2017, from http://ark.intel.com/products/64598/Intel-Xeon-Processor-E5-2637-5M-Cache-3_00-GHz-8_00-GTs-Intel-QPI

[27] (n.d.). Intel. *Intel® Xeon® Processor E5-2620.* Retrieved July 1, 2017, from http://ark.intel.com/products/64594/Intel-Xeon-Processor-E5-2620-15M-Cache-2_00-GHz-7_20-GTs-Intel-QPI

[28] “Watt's up pro power meter specifications, Retrieved July 1, 2017" https://www.wattsupmeters.com/secure/products.php?pn=0&wai=303&spec=4

[29] Kreiss, T. (2009, January 21). Tom’s Hardware: For The Hardcore PC Enthusiast. *Actual Power Consumption And Current Requirements - How Much Power*

Does Your Graphics Card Need?. Retrieved July 1, 2017, from http://www.tomshardware.com/reviews/geforce-radeon-power,2122-6.html

[30] (n.d.). Artificial Intelligence Computing Leadership from NVIDIA. *NVIDIA® Quadro® FX 5800 provides professionals with visual supercomputing from their desktops delivering results that push visualization beyond traditional 3D.* . Retrieved January 1, 2017, from http://www.nvidia.com/object/product_quadro_fx_5800_us.html

[31] (n.d.). TechPowerUp. *NVIDIA GeForce GTX 480 Fermi Review | techPowerUp.* Retrieved July 1, 2017, from http://www.techpowerup.com/reviews/NVIDIA/GeForce_GTX_480_Fermi/30.html

[32] (2016). ACM Digital Library. *Acceleration of k-Means Algorithm Using Altera SDK for OpenCL.* Retrieved from http://dl.acm.org/citation.cfm?id=2964910

[33] (n.d.). Wikipedia, the free encyclopedia. *Mean squared error - Wikipedia.* Retrieved July 1, 2017, from http://en.wikipedia.org/wiki/Mean_squared_error

[34] (n.d.). Nallatech - FPGA Accelerated Computing & Datacentric Computing. *Intel Stratix 10 FPGA - Nallatech 520 FPGA Acceleration Card - Nallatech.* Retrieved July 1, 2017, from http://www.nallatech.com/store/fpga-accelerated-computing/pcie-accelerator-cards/nallatech-520-compute-acceleration-card-stratix-10-fpga/

[35] (n.d.). Nallatech - FPGA Accelerated Computing & Datacentric Computing. *Nallatech 510T Compute Acceleration Card - Nallatech.* Retrieved July 1, 2017, from http://www.nallatech.com/store/fpga-accelerated-computing/pcie-accelerator-cards/nallatech-510t-fpga-computing-acceleration-card/

Appendix A: EM-GMM Opencl Kernel Code

```
//inclusing header
#include "../host/inc/gmm_em.h"

//enabling channel
#pragma OPENCL EXTENSION cl_altera_channels: enable

//----------------------
// Channel declarations
//----------------------


////kernel 1: K_M / mstep
channel data_type CH_MEAN [NUM_DIMENSIONS]
        __attribute__((depth(NUM_CLUSTERS)));
channel data_type CH_MEAN1 [NUM_DIMENSIONS]
        __attribute__((depth(NUM_CLUSTERS)));
channel data_type CH_NSUM
        __attribute__((depth(NUM_CLUSTERS)));
channel data_type CH_PROB
        __attribute__((depth(NUM_CLUSTERS)));
channel data_type CH_CONSTANT
        __attribute__((depth(NUM_CLUSTERS)));
channel data_type CH_RINV [NUM_DIMENSIONS*NUM_DIMENSIONS]
        __attribute__((depth(NUM_CLUSTERS)));
channel data_type CH_RINV1 [NUM_DIMENSIONS*NUM_DIMENSIONS]
        __attribute__((depth(NUM_CLUSTERS)));

////K_E1 / estep_1
channel int E2_START;
channel int T_START

//output channels.
#define K_E1_IN_CH_MEAN CH_MEAN1
#define K_E1_IN_CH_PROB CH_PROB
#define K_E1_IN_CH_CONSTANT CH_CONSTANT
#define K_E1_IN_CH_RINV CH_RINV1


//**************************************************************************
*******
//START >> Kernel 1: mstep_1 (calculate: N_sum,mean, prob)
//**************************************************************************
*******
__kernel
__attribute__((task))
void mstep_1(          __global data_type * restrict K_sample_DN,
                               __global data_type * restrict K_membership)
```

```
{
	//local storage : N_SUM
	cl_data_type N_SUM = 0.0;

	//local storage : prob
	cl_data_type prob = 0.0;

	//local storage : mean
	data_type local_member [K_M_MEM_BS];
	data_type local_data [K_M_BS];
	//data_type temp_mean_sum;
	data_type  local_mean [NUM_DIMENSIONS];

	//printf("\nSTART: MSTEP\n");
	for (int m = 0; m < NUM_CLUSTERS; m++)
	{
		//-------------------
		//initializing value
		//-------------------
		N_SUM = 0.0;
		prob = 0.0;

		//-------------------------
		//Calculation: N_SUM, MEAN
		//-------------------------
		for (int nb = 0; nb < K_M_NB; nb++)  //nb = number of blocks
		{
			//---------------------------------------------------------
			//transfer data from Global memory to local memory: membership
			//---------------------------------------------------------
			#pragma unroll K_M_MEM_BS
			for (int bs = 0; bs < K_M_MEM_BS; bs++) //
			{local_member [bs] = K_membership[m * NUM_OF_POINTS + nb *
K_M_BS + bs];}				//read membership[M,N]

			//-------------------
			//Calculation: N_SUM
			//-------------------
			#pragma unroll K_M_LU_NSUM
			for (int bs = 0; bs < K_M_MEM_BS; bs++)
			{N_SUM += local_member[bs];}

			//---------------------------------------------------------
			//temporarily summing membership and sample_DN data for mean
			//--------------------------------------------------------
			for (int d = 0; d < NUM_DIMENSIONS; d++)
			{
				//---------------------------------------------------------
				//transfer data from Global memory to local memory:
sample_DN
				//---------------------------------------------------------
```

```
                    #pragma unroll K_M_BS           // K_M_BS (check gmm_em.h)
                    for (int k = 0; k < K_M_BS; k++)          //bs = block size
                    {local_data [k] = K_sample_DN [d * NUM_OF_POINTS + nb *
K_M_BS + k];}

                    //----------------------------------------------------------
                    //temporarily summing membership and sample_DN data for
mean
                    //----------------------------------------------------------
                    #pragma unroll K_M_LU_MEAN          // K_M_BS (check
gmm_em.h)

                    for (int j = 0; j < K_M_BS; j++)
                         {local_mean[d] += local_data [j] * local_member[j];}

               }


          } //end nb loop



          //--------------------------
          // Mean/average calculation
          //--------------------------
          #pragma unroll K_M_LU_MEAN1
          for (int dm = 0; dm < NUM_DIMENSIONS; dm++)
               {
                    local_mean[dm] = local_mean[dm] / N_SUM;


                    write_channel_altera(CH_MEAN[dm], local_mean [dm]);

               }

          //----------------------
          //write to channel: NSUM
          //----------------------
          write_channel_altera(CH_NSUM, N_SUM);

          //--------------------------------------
          //Calculation and write to channel: prob
          //--------------------------------------
          prob = N_SUM / NUM_OF_POINTS;
          write_channel_altera(CH_PROB, prob);

     } //mloop

} //end of kernel
```

```
//****************************************************************************
*******
//END >> Kernel 1: mstep_1 (calculate: N_sum,mean, prob)
//****************************************************************************
*******

//****************************************************************************
*******
//START >> Kernel 2: mstep_cov (calculate: covariance)
//****************************************************************************
*******

__kernel
__attribute__((task))
void mstep_cov(                 __global data_type * restrict K_sample_ND,
                                    __global data_type * restrict K_membership)
{

        cl_data_type N_SUM = 0.0;
        data_type local_member [K_M_MEM_BS];
        data_type local_data [K_M_BS];
        data_type  local_mean [NUM_DIMENSIONS];

        //local storage : Covariance
        data_type local_R [NUM_DIMENSIONS * NUM_DIMENSIONS];
        data_type local_sample [NUM_DIMENSIONS];

        //printf("\nSTART: MSTEP\n");
        for (int m = 0; m < NUM_CLUSTERS; m++)
        {
                //------------------------
                //read from channel: mean
                //------------------------
                #pragma unroll K_LU_CH_MEAN
                for (int i = 0; i <NUM_DIMENSIONS; i++)
                {
                        local_mean [i] = read_channel_altera(CH_MEAN[i]);
                }

                N_SUM = read_channel_altera(CH_NSUM);

                #pragma unroll K_M_LU_I
                for (int z = 0; z < INIT_FACTOR; z++)
                {
                        local_R [z] = 0.0;
                        local_R [INIT_FACTOR + z] = 0.0;
                }


                //-----------------------
                //Calculation: Covariance
```

```
//-----------------------
for (int nb = 0; nb < K_M_MEM_NB; nb++) //nb = number of blocks
{
	//--------------------------------------------------------
	//transfer data from Global memory to local memory: membership
	//--------------------------------------------------------
	#pragma unroll K_M_MEM_BS
	for (int bs = 0; bs < K_M_MEM_BS; bs++)	//bs = block size
	{local_member [bs] = K_membership[m * NUM_OF_POINTS + nb *
K_M_BS + bs];}		//read membership[M,N]

	for (int bs = 0; bs < K_M_MEM_BS; bs++) //bs = block size
	{
		//--------------------------------------------------------
		//transfer data from Global memory to local memory:
sample_DN
		//--------------------------------------------------------
		#pragma unroll K_M_LU_COV_L1
		for (int ns = 0; ns < NUM_DIMENSIONS; ns++)
		{
			local_sample[ns] = K_sample_ND[(K_M_MEM_BS *
NUM_DIMENSIONS * nb) + (bs * NUM_DIMENSIONS) + ns];
		}

		//-----------------------------------------------------------------------
---------------------------
		//Calculation: Covariance > summing membership , mean and
sample_ND data for covariance / sum of n points
		//-----------------------------------------------------------------------
---------------------------
		#pragma unroll K_M_LU_COV_L2
		for (int i = 0; i < NUM_DIMENSIONS; i++)
		{
			#pragma unroll K_M_LU_COV_L3
			for (int j = 0; j < NUM_DIMENSIONS; j++)
			{
				local_R [i * NUM_DIMENSIONS + j] +=
(cl_data_type)((local_sample[i]-local_mean[i]) * (local_sample[j]-local_mean[j]) *
local_member[bs] / (cl_data_type) N_SUM);

			} //j loop=d
		} //i loop = d
	} //bs loop
} //nb loop

//----------------------
//write to channel: RINV
//----------------------
#pragma unroll K_M_RINV
for (int j = 0; j < K_M_RINV; j++)
{
```

```
				write_channel_altera(CH_RINV[j], local_R[j]);
				write_channel_altera(CH_RINV[K_M_RINV + j], local_R[K_M_RINV
+ j]);
			}


			//----------------------
			//write to channel: mean
			//----------------------
			#pragma unroll K_LU_CH_MEAN
			for (int i = 0; i <NUM_DIMENSIONS; i++)
			{
				write_channel_altera(CH_MEAN1[i], local_mean [i]);
			}


	} //m loop
} //end kernel

//**************************************************************************
*******
//END >> Kernel 2: mstep_cov (calculate: covariance)
//**************************************************************************
*******


//**************************************************************************
*******
//START >> Kernel 3: mstep_inv (calculate: inverse covariance)
//**************************************************************************
*******

__kernel
__attribute__((task))
void mstep_inv()

{
	//local storage : Covariance
	data_type local_R [NUM_DIMENSIONS * NUM_DIMENSIONS];

	//local storage : Inverse Covariance
	data_type temp_sum_determinant = 0.0;
	data_type x = 0.0;
	data_type y = 0.0;
	data_type sum_U = 0.0;
	data_type sum_L = 0.0;
	data_type sum_final = 0.0;

	//local storage : constant
	data_type const_local;
```

```
	//printf("\nSTART: MSTEP\n");
	for (int m = 0; m < NUM_CLUSTERS; m++)
	{
		//---------------------
		//read from channel: R
		//---------------------
		#pragma unroll K_M_RINV
		for (int j = 0; j < K_M_RINV; j++)
		{
			local_R[j] = read_channel_altera(CH_RINV[j]);
			local_R[K_M_RINV + j] = read_channel_altera(CH_RINV[K_M_RINV
+ j]);
		}


		//--------------------------------
		//Calculation: Inverse Covariance
		//--------------------------------
		//////////////////
		// normalize row 0
		//////////////////
		//#pragma unroll K_M_LU_ICOV_L1
		#pragma unroll
		for (int i=1; i < NUM_DIMENSIONS; i++)
			{local_R[i] /= local_R[0];}

		//////////////////
		//LU decomposition
		/////////////////
		//#pragma unroll K_M_LU_ICOV_L3
		for (int i=1; i < NUM_DIMENSIONS; i++)
		{
			//#pragma unroll K_M_LU_ICOV_L2
			#pragma unroll
			for (int j=i; j < NUM_DIMENSIONS; j++)
			{ // do a column of L
				sum_L = 0.0;

				//#pragma unroll K_M_LU_ICOV_L4
				#pragma unroll
				for (int k = 0; k < i; k++)
				{sum_L += local_R[j*NUM_DIMENSIONS+k] *
local_R[k*NUM_DIMENSIONS+i];}

				local_R[j*NUM_DIMENSIONS+i] =
local_R[j*NUM_DIMENSIONS+i] - sum_L;

			} //j loop

			if (i == NUM_DIMENSIONS-1) continue;
```

```
				//#pragma unroll K_M_LU_ICOV_L2
				#pragma unroll
				for (int j=i+1; j < NUM_DIMENSIONS; j++)
				{// do a row of U
					sum_U = 0.0;

					//#pragma unroll K_M_LU_ICOV_L4
					#pragma unroll
					for (int k = 0; k < i; k++)
					{sum_U +=
local_R[i*NUM_DIMENSIONS+k]*local_R[k*NUM_DIMENSIONS+j];}

					local_R[i*NUM_DIMENSIONS+j] =
(local_R[i*NUM_DIMENSIONS+j]-sum_U) / local_R[i*NUM_DIMENSIONS+i];
				}// j loop
			} //iloop


			temp_sum_determinant = 0.0;

			////////////////////////
			//calculate determinate
			///////////////////////

			#pragma unroll K_M_LU_ICOV_L2
			for(int i=0; i<NUM_DIMENSIONS; i++)
			{ temp_sum_determinant += logf(fabs(local_R[i*NUM_DIMENSIONS+i])); }

			//-----------------------------------------
			//Calculation and write to channel: constant
			//-----------------------------------------
			const_local = (cl_data_type)(-(NUM_DIMENSIONS * 0.5 * log(2.0 * PI)) - (0.5
* temp_sum_determinant));
			write_channel_altera(CH_CONSTANT, const_local);

			///////////
			//invert L
			//////////
			//#pragma unroll K_M_LU_ICOV_L3
			#pragma unroll
			for ( int i = 0; i < NUM_DIMENSIONS; i++ )  // invert L
			{
				//no pragma unroll: compiler error
				for ( int j = i; j < NUM_DIMENSIONS; j++ )
				{
					x = 1.0;

					if ( i != j )
					{
						x = 0.0;
```

```
							//no pragma unroll: compiler error
							for ( int k = i; k < j; k++ )
							{x -=
local_R[j*NUM_DIMENSIONS+k]*local_R[k*NUM_DIMENSIONS+i];}
						}
						local_R[j*NUM_DIMENSIONS+i] = x /
local_R[j*NUM_DIMENSIONS+j];
				} //j loop
			}



			/////////////
			// invert U
			////////////
			//#pragma unroll K_M_LU_ICOV_L3
			#pragma unroll
			for ( int i = 0; i < NUM_DIMENSIONS; i++ )
			{
				//no pragma unroll: compiler error
				for ( int j = i; j < NUM_DIMENSIONS; j++ )
				{
					if ( i == j ) continue;

					y = 0.0;

					//no pragma unroll: compiler error
					for ( int k = i; k < j; k++ )
					{y += local_R[k*NUM_DIMENSIONS+j]*( (i==k) ? 1.0 :
local_R[i*NUM_DIMENSIONS+k]);}

					local_R[i*NUM_DIMENSIONS+j] = -y;
				} //j loop
			}

			//-----------------
			// final inversion
			//-----------------
			//#pragma unroll K_M_LU_ICOV_L3
			#pragma unroll
			for ( int i = 0; i < NUM_DIMENSIONS; i++ )
			{
				//#pragma unroll K_M_LU_ICOV_L2
				#pragma unroll
				for ( int j = 0; j < NUM_DIMENSIONS; j++ )
				{
					sum_final = 0.0;

					//no pragma unroll: compiler error
					for ( int k = ((i>j)?i:j); k < NUM_DIMENSIONS; k++ )
					{sum_final +=
```

```
((j==k)?1.0:local_R[j*NUM_DIMENSIONS+k])*local_R[k*NUM_DIMENSIONS+i];}

                                        local_R[j*NUM_DIMENSIONS+i] = sum_final;
                              }
                    }

                    //-----------------
                    //write to channel
                    //-----------------
                    #pragma unroll K_M_RINV
                    for (int j = 0; j < K_M_RINV; j++)
                    {
                              write_channel_altera(CH_RINV1[j], local_R[j]);
                              write_channel_altera(CH_RINV1[K_M_RINV + j],
local_R[K_M_RINV + j]);
                    }




          } // mloop
} // end kernel


//*****************************************************************************
*******
//END >> Kernel 3: mstep_inv (calculate: inverse covariance)
//*****************************************************************************
*******


//*********************************************************************
//START >> Kernel 4: estep_1 (calculating numerator of Expectation)
//*********************************************************************

#define PROB (1.0/(data_type)NUM_CLUSTERS)




__kernel
__attribute__((task))
void estep_1(     __global data_type* restrict k_sample_ND,
                                        __global data_type* restrict k_e1_membr)
{
          //local storage
          cl_data_type sample_local [NUM_DIMENSIONS];
          cl_data_type Rinv_local [NUM_DIMENSIONS * NUM_DIMENSIONS];
          data_type mean_local [NUM_DIMENSIONS];
          cl_data_type prob_local = 0.0;
          cl_data_type constant_local = 0.0;
```

```
int e2_start = 0;

cl_data_type temp_membership = 0.0;
cl_data_type membership = 0.0;
int index = 0; //indexing channel


//block/global index
//uint gid_m = get_global_id(0);

//local index
//uint lid_d = get_local_id(0);

//printf("\n\nestep_1\n\n");
for (int m =; m < NUM_CLUSTERS ; m++)
{
	//------------------------------------------------
	//transfer data from channel to local memory
	//------------------------------------------------

	//transfer data from channel: mean
	#pragma unroll K_LU_CH_MEAN
	for (int i = 0; i <NUM_DIMENSIONS; i++)
		{mean_local[i] = read_channel_altera(K_E1_IN_CH_MEAN[i]);}



	//transfer data from channel: Rinv
	#pragma unroll K_M_RINV
	for (int j = 0; j <K_M_RINV; j++)
		{
			Rinv_local[j] = read_channel_altera(K_E1_IN_CH_RINV[j]);
			Rinv_local[K_M_RINV + j] =
read_channel_altera(K_E1_IN_CH_RINV[K_M_RINV + j]);
		}



	//transfer data from channel: prob
	prob_local = read_channel_altera(CH_PROB);


	//transfer data from channel: constants
	constant_local = read_channel_altera(CH_CONSTANT);

	//-----------------------
	//calculation: Numerator
	//-----------------------

	//for (int n = gid_n; n <= gid_n; n++)
	for (int n = 0; n < NUM_OF_POINTS; n++)
```

```
			{
				//-------------------------------------------
				//transfer data from global memory: sample_ND
				//-------------------------------------------
				#pragma unroll K_E1_LU_S_ND
				for (int s = 0; s <NUM_DIMENSIONS; s++)
					{
					sample_local [s] = k_sample_ND [n * NUM_DIMENSIONS +
s];
					}


				//------------
				//calculation
				//------------
				membership = 0.0;

				//membership [n] = 0.0;
				#pragma unroll K_E1_LU_L1
				for (int i = 0; i < NUM_DIMENSIONS; i++)
				{
					temp_membership = 0.0;

					#pragma unroll K_E1_LU_L2
					for (int j = 0; j < NUM_DIMENSIONS; j++)
						{temp_membership += (sample_local[i] - mean_local[i])
* (sample_local[j] - mean_local[j]) * Rinv_local[i * NUM_DIMENSIONS + j];} //j loop=d
				}



				membership  = (cl_data_type)(-0.5 * temp_membership + constant_local
+ log(prob_local));


				//----------------------------------------------------------
				//transfer data from local memory to Global memory: membership
				//----------------------------------------------------------
				k_e1_membr [m * NUM_OF_POINTS + n] = membership;

				//-----------------
				//start kernel K_2
				//-----------------
				if (m == (NUM_CLUSTERS-1) && n == ((NUM_OF_POINTS-1)/8))
				{
					e2_start = 1;
					write_channel_altera(E2_START, e2_start);
				}
				//printf("E1: e2start=%d\n",e2_start);
```

```
		}//n loop

	} // m loop


}//end kernel

//****************************************************************
//END >> Kernel 4: estep_1 (calculating numerator of Expectation)
//****************************************************************


//****************************************************************
//START >> Kernel 5: estep_2 (Final calculation Expectation)
//****************************************************************


__kernel
__attribute__((task))
void estep_2(		__global data_type * restrict k_e2_membr,
					__global data_type* restrict k_e1_membr)
{
	//local storage
	//__local cl_data_type membership_local1 [NUM_CLUSTERS];
	cl_data_type membership_local [NUM_CLUSTERS];
	cl_data_type maximum = 0.0;
	cl_data_type denominator_sum = 0.0;
	int t_start = 0;
	int e2_start = 0;

	e2_start = read_channel_altera(E2_START);


	//printf("\nSTART: ESTEP2\n");
	for (int n = 0; n < NUM_OF_POINTS; n++)
	{
		//------------------------------------------------
		//transfer data from global memory to local memory
		//------------------------------------------------

		if(e2_start == 1)
		{
			#pragma unroll K_E2_LU_L1
			for (int j = 0; j <NUM_CLUSTERS; j++)
			{
				membership_local[j] = k_e1_membr[j * NUM_OF_POINTS +
n];

			}
```

```
		}



		//mem_fence(CLK_CHANNEL_MEM_FENCE); //Not sure if need this

		//------------------------
		//calculation: Denomerator
		//------------------------

		// find the maximum likelihood for this event
		//max = h_memberships[n];
		maximum = 0.0;

		#pragma unroll K_E2_LU_L1
		for (int ml = 1; ml < NUM_CLUSTERS; ml++)
		{maximum = (cl_data_type)fmax (membership_local[0],
membership_local[ml]);}

		// Compute P(x_n), the denominator of the probability (sum of weighted
likelihoods)
		denominator_sum = 0.0;

		#pragma unroll K_E2_LU_L1
		for (int ds = 0; ds < NUM_CLUSTERS; ds++)
		{denominator_sum += (cl_data_type)exp(membership_local[ds] - maximum);}

		denominator_sum = maximum + (cl_data_type)log(denominator_sum);

		// Divide by denominator, also effectively normalize probabilities
		// exp(log(p) - log(denom)) == p / denom
		#pragma unroll K_E2_LU_L1
		for (int m = 0; m < NUM_CLUSTERS; m++)
		{membership_local[m] = (cl_data_type)exp(membership_local[m] -
denominator_sum);
		}

		//------------------------------------------------------
		//transfer data from local memory to global memory/channel
		//------------------------------------------------------


			//printf("ESTEP2:DDR\n");
			#pragma unroll K_E2_LU_L1
			for (int m = 0; m < NUM_CLUSTERS; m++)
			{
				k_e2_membr[n * NUM_CLUSTERS + m] =
membership_local[m];


				if (m == (NUM_CLUSTERS-1) && n ==
```

```
((NUM_OF_POINTS-1)))
                                        {
                                                t_start = 1;
                                                write_channel_altera(T_START, t_start);
                                        }
                                }


                                //wait for the entire block to be loaded
                                //barrier(CLK_GLOBAL_MEM_FENCE);




        } //n loop = NUM_OF_POINTS

        //printf("\nEND: ESTEP2\n");
}//end kernel

//****************************************************************
//END >> Kernel 5: estep_2 (Final calculation Expectation)
//****************************************************************


//****************************************************************
//START >> Kernel 6: transpose (transpose membership)
//****************************************************************

__kernel
__attribute__((task))
//__attribute__((num_compute_units(4)))
//__attribute__((num_simd_work_items(NUM_CLUSTERS)))
void transpose(         __global data_type * restrict K_membership,
                                        __global data_type* restrict k_e2_membr)
{

        int t_start = 0;
        //printf("T: tstart = %d", t_start);
        t_start = read_channel_altera(T_START);
        //printf("T: tstart = %d", t_start);
        if (t_start == 1)
                {
                        //printf("Transpose\n");
                        #pragma unroll 2
                        for (int m = 0; m < NUM_CLUSTERS ; m++)
                        {
                                #pragma unroll 8
                                for (int n = 0; n < NUM_OF_POINTS ; n++)
                                {K_membership[m * NUM_OF_POINTS + n] = k_e2_membr[n
* NUM_CLUSTERS + m];}
                        }
```

```
                }

}

//***************************************************************
//END >> Kernel 6: transpose (transpose membership)
//***************************************************************
```

Vita Auctoris

NAME:	Mohammad Abdul Momen
PLACE OF BIRTH:	Dhaka, Bangladesh
YEAR OF BIRTH:	1993
EDUCATION:	Bachelor of Science in Electrical & Electronic Engineering (2011-2015) North South University, Dhaka, Bangladesh. Master of Applied Science in Electrical Engineering (2016-2017) Department of Electrical and Computer Engineering, University of Windsor, Windsor, ON, Canada.