

12-2014

# Scientific Application Acceleration Utilizing Heterogeneous Architectures

Edwin Weill

*Clemson University*, [eweill@g.clemson.edu](mailto:eweill@g.clemson.edu)

Follow this and additional works at: [https://tigerprints.clemson.edu/all\\_theses](https://tigerprints.clemson.edu/all_theses)



Part of the [Computer Engineering Commons](#), and the [Computer Sciences Commons](#)

---

## Recommended Citation

Weill, Edwin, "Scientific Application Acceleration Utilizing Heterogeneous Architectures" (2014). *All Theses*. 2046.  
[https://tigerprints.clemson.edu/all\\_theses/2046](https://tigerprints.clemson.edu/all_theses/2046)

This Thesis is brought to you for free and open access by the Theses at TigerPrints. It has been accepted for inclusion in All Theses by an authorized administrator of TigerPrints. For more information, please contact [kokeefe@clemson.edu](mailto:kokeefe@clemson.edu).

# SCIENTIFIC APPLICATION ACCELERATION UTILIZING HETEROGENEOUS ARCHITECTURES

---

A Thesis  
Presented to  
the Graduate School of  
Clemson University

---

In Partial Fulfillment  
of the Requirements for the Degree  
Master of Science  
Computer Engineering

---

by  
Edwin Weill  
December 2014

---

Accepted by:  
Dr. Melissa C. Smith, Committee Chair  
Dr. Amy Apon  
Dr. Walter Ligon

# Abstract

Within the past decade, there have been substantial leaps in computer architectures to exploit the parallelism that is inherently present in many applications. The scientific community has benefited from the emergence of not only multi-core processors, but also other, less traditional architectures including general purpose graphical processing units (GPGPUs), field programmable gate arrays (FPGAs), and Intel's many integrated cores (MICs) architecture (i.e. Xeon Phi). The popularity of the GPGPU has increased rapidly because of their ability to perform massive amounts of parallel computation quickly and at low cost with an ease of programmability. Also, with the addition of high-level programming interfaces for these devices, technical and non-technical individuals can interface with the device and rapidly obtain improved performance for many algorithms. Many applications can take advantage of the parallelism present in distributed computing and multithreading to achieve higher levels of performance for the computationally intensive parts of the application.

The work presented in this thesis implements three applications for use in a performance study of the GPGPU architecture and multi-GPGPU systems. The first application study in this research is a K-Means clustering algorithm that categorizes each data point into the closest cluster. The second algorithm implemented is a spiking neural network algorithm that is used as a computational model for machine learning. The third, and final,

study is the longest common subsequences problem, which attempts to enumerate comparisons between sequences (namely, DNA sequences).

The results for the aforementioned applications with varying problem sizes and architectural configurations are presented and discussed in this thesis. The K-Means clustering algorithm achieved approximately 97x speedup when utilizing an architecture consisting of 32 CPU/GPGPU pairs. To achieve this substantial speedup, up to 750,000 data points were used with up to 30,000 centroids (means). The spiking neural network algorithm resulted in speedups of about 33x for the entire algorithm and 160x for each iteration with a two-level network with 1000 total neurons (800 excitatory and 200 inhibitory neurons). The longest common subsequences problem achieved speedup of greater than 10x with 100 random sequences up to 500 characters in length. The maximum speedup values for each application were achieved by utilizing the GPGPU as well as multi-core devices simultaneously. The computations were scattered over multiple CPU/GPGPU pairs with the computationally intensive pieces of the algorithms offloaded onto the GPGPU device.

The research in this thesis illustrates the ability to scale a heterogeneous cluster (i.e. CPUs and GPUs working collaboratively) for large-scale scientific application performance improvements. Each algorithm demonstrates slightly different types of computations and communications, which can be compared to other algorithms to predict how they would perform on an accelerator. The results show that substantial speedups can be achieved for scientific applications when utilizing the GPGPU and multi-core architectures.

# Dedication

I dedicate this thesis to my parents and my academic advisor without whose constant support and guidance this thesis would not have been possible. I also dedicate this thesis to my wife and my younger sister, who have always stood by my side and been an inspiration.

# Acknowledgements

This thesis was made possible by the help and support of the following faculty members, family members, and colleagues.

I begin by thanking my academic advisor Dr. Melissa Smith, who graciously accepted me as her pupil. Dr. Smith's words of wisdom, immense experience, constant encouragement, and insightful suggestions allowed for the completion of this research and thesis.

I would also like to thank Dr. Amy Apon and Dr. Walt Ligon for serving on my committee and instructing me in various aspects of computer architecture.

I am also thankful for my parents, Edwin and Jeanine, my Sister, Shannon, and my wife, Hiliary, who provided constant support during my research and writing this manuscript.

I also owe a debt of gratitude to my fellow colleagues from the Future Computing Technologies Lab at Clemson University: Ashraf, Vivek, Karan, Nimisha and many others whom I fail to mention for providing me a stimulating work environment during this process.

I would also like to acknowledge the National Science Foundation MRI Grant Award No. 1228312 for making this research possible (PI Amy Apon; Co-PIs Melissa C Smith, Walter Ligon, Jill Gemmill, and Steven Stuart).

# Table of Contents

	Page
Title Page .....	i
Abstract .....	ii
Dedication.....	iv
Acknowledgements.....	v
List of Tables .....	ix
List of Figures .....	xi
 Chapter	
1 Introduction.....	1
2 Related Work.....	6
2.1 K-Means.....	6
2.2 Izhikevich SNN.....	8
2.3 Longest Common Subsequences.....	11
2.4 Summary .....	12
3 Background .....	13
3.1 K-Means Clustering.....	13
3.1.1 K-Means Algorithm.....	14
3.1.2 K-Means Clustering Real-world Applications .....	16
3.2 Spiking Neural Networks .....	17
3.2.1 Izhikevich SNN Model .....	19
3.2.2 SNN Real-world Applications.....	21
3.3 Longest Common Subsequences Problem .....	21
3.3.1 LCS Real-world Applications .....	24

## Table of Contents (Continued)

	Page
3.4 Summary .....	24
<b>4 Computing Architectures and Programming Models .....</b>	<b>26</b>
4.1 Computing Architectures .....	27
4.1.1 Multi-Core Architecture .....	27
4.1.1.1 Intel Xeon E5-2665.....	28
4.1.2 GPGPU Architecture .....	29
4.1.2.1 NVIDIA Tesla K20 (Kepler GK110) .....	32
4.2 Programming Models.....	35
4.2.1 MPI.....	35
4.2.2 CUDA.....	36
4.3 Palmetto Cluster.....	39
4.4 Summary .....	40
<b>5 Experimental Setup and Implementation .....</b>	<b>41</b>
5.1 Experimental Setup .....	41
5.1.1 Heterogeneous System Setup .....	42
5.1.2 Compiler.....	42
5.2 Implementations .....	43
5.2.1 K-Means .....	44
5.2.2 Spiking Neural Networks.....	45
5.2.3 Longest Common Subsequences.....	46
5.3 GPU Implementations.....	47
5.3.1 Implementation 1 (Global Memory) .....	47
5.3.2 Implementation 2 (Constant Memory) .....	50
5.3.3 Implementation 3 (GPGPU Math Library).....	51
5.3.4 Multi-GPU Implementations .....	52
5.4 Summary .....	53
<b>6 Results and Analysis .....</b>	<b>54</b>
6.1 K-Means.....	55
6.1.1 Single-Core Implementation.....	55
6.1.2 Multi-Core Implementation.....	57
6.1.3 Single-GPU Implementation .....	58
6.1.4 Multi-GPU Implementation .....	60
6.1.5 Speedup .....	62
6.2 Spiking Neural Networks (SNN) .....	64
6.2.1 Single-Core Implementation.....	64

## Table of Contents (Continued)

	Page
6.2.2 Multi-Core Implementation.....	64
6.2.3 Single-GPU Implementation.....	68
6.2.4 Multi-GPU Implementation.....	70
6.2.5 Speedup .....	75
6.3 Longest Common Subsequences (LCS).....	79
6.3.1 Single-Core Implementation.....	79
6.3.2 Multi-Core Implementation.....	80
6.3.3 Single-GPU Implementation.....	83
6.3.4 Multi-GPU Implementation.....	85
6.3.5 Speedup .....	87
6.4 Summary .....	88
<b>7 Conclusions and Future Work .....</b>	<b>90</b>
7.1 Conclusions .....	90
7.2 Future Works.....	93
<b>References.....</b>	<b>98</b>

# List of Tables

Table	Page
4.1 Intel Xeon E5-2665 (“Romley”) Statistics .....	28
4.2 Kepler GK110 (Tesla K20) Statistics.....	34
4.3 Commonly used MPI Functions .....	36
5.1 Features available on the NVIDIA Tesla K20m device .....	43
5.2 Data Configurations for K-Means Clustering .....	44
5.3 Data Configurations for Izhikevich SNN Algorithm.....	45
5.4 Data Configurations for LCS Algorithm.....	46
6.1 Execution Times for Single-Core K-Means Algorithm .....	56
6.2 Execution Times for Multi-Core K-Means Algorithm .....	57
6.3 Execution Times for Single-GPU K-Means Algorithm.....	59
6.4 Execution Times for Multi-GPU K-Means Algorithm.....	60
6.5 Speedup over Single-Core for all K-Means Implementations .....	63
6.6 Execution Times for Single-Core Izhikevich SNN .....	64
6.7 Execution Times for Multi-Core Izhikevich SNN .....	65
6.8 Execution Times for Single-GPU Izhikevich SNN .....	68
6.9 Total Execution Times for Multi-GPU Izhikevich SNN .....	71
6.10 Iteration 1 Execution Times for Multi-GPU Izhikevich SNN .....	72

## List of Tables (Continued)

Table	Page
6.11 Iteration 2 Execution Times for Multi-GPU Izhikevich SNN .....	74
6.12 Speedup of Total Execution Times for Izhikevich SNN .....	76
6.13 Speedup of Iteration 1 Execution Times for Izhikevich SNN .....	77
6.14 Speedup of Iteration 2 Execution Times for Izhikevich SNN .....	78
6.15 Execution Times for Single-Core LCS Algorithm.....	79
6.16 Execution Times for Multi-Core LCS Algorithm.....	82
6.17 Execution Times for Single-GPU LCS Algorithm .....	84
6.18 Execution Times for Multi-GPU LCS Algorithm .....	86
6.19 Speedup Values over Single-Core LCS Algorithm .....	87

# List of Figures

Figure	Page
3.1 Example of LCS Algorithm .....	23
4.1 NVIDIA GeForce 8800 GTX Architecture [Reference].....	30
4.2 NVIDIA Fermi Architecture [Reference] .....	31
4.3 NVIDIA Kepler GK110 Architecture [Reference] .....	32
4.4 GK110 SMX Architecture [Reference] .....	33
4.5 Thread block layout in CUDA [Reference] .....	38
6.1 Execution Times for Single-Core K-Means Algorithm .....	56
6.2 Execution Times for Multi-Core K-Means Algorithm .....	58
6.3 Execution Times for Single-GPU K-Means Algorithm.....	59
6.4 Execution Times for Multi-GPU K-Means Algorithm.....	61
6.5 Execution Times for Multi-GPU Comparison with Multi-Core K-Means....	62
6.6 Speedup over Single-Core for all K-Means Implementations .....	63
6.7 Execution Times for Multi-Core Izhikevich SNN .....	66
6.8 Execution Times for Multi-Core (Large # of Cores) Izhikevich SNN .....	68
6.9 Execution Times for Single-GPU Izhikevich SNN .....	69
6.10 Execution Times for Single-GPU vs. Single-Core Izhikevich SNN .....	70
6.11 Total Execution Times for Multi-GPU Izhikevich SNN .....	71

## List of Figures (Continued)

Figure	Page
6.12 Total Execution Times Comparison to Multi-Core Izhikevich SNN.....	72
6.13 Iteration 1 Execution Times for Multi-GPU Izhikevich SNN .....	73
6.14 Iteration 1 Execution Times Comparison to Multi-Core Izhikevich SNN ...	73
6.15 Iteration 2 Execution Times for Multi-GPU Izhikevich SNN .....	74
6.16 Iteration 2 Execution Times Comparison to Multi-Core Izhikevich SNN ...	75
6.17 Speedup of Total Execution Times vs. Single-Core Izhikevich SNN .....	76
6.18 Speedup of Iteration 1 Execution Times vs. Single-Core Izhikevich SNN...	77
6.19 Speedup of Iteration 2 Execution Times vs. Single-Core Izhikevich SNN...	78
6.20 Execution Times of Single-Core LCS Algorithm .....	80
6.21 Execution Times of Multi-Core LCS Algorithm .....	82
6.22 Execution Times for Single-GPU LCS Algorithm .....	84
6.23 Execution Times for Multi-GPU LCS Algorithm .....	87
6.24 Speedup over Single-Core for LCS Algorithm.....	88

# Chapter 1

## Introduction

For a significant time in the past, scientific researchers relied on the advancement of computer architectures, in terms of higher clock speeds and other low-level optimizations, to increase the performance of their applications. However, due to hardware limitations including memory, clock speed, and physical processor size, parallel computing architectures are necessary to bridge the gap between the need for performance improvements in applications and the lack of significant advancements in the low-level architecture design (i.e. lithography limits, clock wall, power density, etc.). The trends have moved toward parallel computing architectures, leading vendors to increase the number of cores per device and increase the number of processors per machine.

However, computationally intensive applications need more than just multi-core architectures to perform well. Therefore, along with the advances that have emerged with multi-core architectures, advances in the field of heterogeneous computing have developed simultaneously to comply with the computational needs. Heterogeneous computing can be described as the combination of a CPU host and one or more special-purpose computing devices or accelerators such as general-purpose graphical processing units (GPGPUs), field programmable gate arrays (FPGAs), and Intel's many- integrated cores (MICs) architecture

(i.e. Xeon Phi [1]). From the list of special-purpose computing devices, GPGPUs have gained significant leverage in the parallel computing niche due to the ability to perform massively parallel tasks at a relatively low cost and easy programming interface through open source languages such as OpenCL [2] and architecture specific languages such as NVIDIA's Compute Unified Device Architecture (CUDA) [3]. With the growing need for increased performance in computationally intensive applications, GPGPUs provide a means for more efficient execution exploiting many-core resources. The release of NVIDIA's CUDA allowed researchers to explore the parallelism of GPGPUs with their applications. With the growing popularity of NVIDIA's CUDA, GPGPUs became the mainstream accelerator in high-performance computing (HPC) and therefore forced vendors to continually evolve them to include advanced features and increase the number of compute cores.

Even with the emergence of GPGPUs in the scientific community, there are still needs for more, faster parallel computations. Therefore, heterogeneity is achieved through the combination of the multi-core and many-core architectures. These systems allow the developer to optimally parallelize code sections, while the remaining code is executed on a sequential processor or core. In heterogeneous systems, the CPU host executes sequential computations such as the data management and file I/O, while the accelerator(s) perform the computationally intensive parts of the application or algorithm, thereby improving the overall application performance. The CPU host and accelerators are connected via special interconnect technologies such as Infiniband [4] or PCI express [5].

Many scientific applications use complex calculations on very large data sets. To perform these calculations, CPUs serialize each computation, which can cause very long execution times if the algorithm is complex or the data set is large. For this reason, it is

preferred to exploit the parallelism of GPGPUs. The algorithms used in this research study (clustering, neural networks, and longest common subsequences) create a diverse algorithm set specifically selected to prove that multiple types of algorithms can utilize GPGPU parallelism. The research in this thesis illustrates the efficacy of attempting to parallelize an application if the execution time is large or there is a large percentage of computation that can be parallelized.

With the emergence of heterogeneous systems, researchers are concerned about how well these computing systems fit in their research paradigm and improve performance of their applications. This research study conducts a performance analysis of a variety of scientific applications using heterogeneous systems including GPGPUs to understand how well different types of applications scale with the system. There are two programming models used in this study including CUDA and MPI. The use of these models allows for a comparison between multi-core and heterogeneous systems as well as an analysis of the algorithms that later will be described in Chapter 3.

The first algorithm discussed as part of this research study in Chapter 3 is the K-Means clustering algorithm [6] where  $n$  observations are clustered into  $k$  clusters. K-Means is an iterative process where each of the  $n$  observations is moved into a group (one of the  $k$  clusters) such that each observation is clustered to the closest cluster centroid. An implementation of this algorithm utilizing a heterogeneous GPGPU architecture is developed and is described in Chapter 5.

The second algorithm discussed as part of this research study in Chapter 3 is the Izhikevich Spiking Neural Network algorithm [7], which is a computational model used for machine learning algorithms such as speech or facial recognition and computer vision. The

Izhikevich algorithm is highly biologically accurate and computationally efficient allowing for more computation in a given time period compared to other neural network algorithms. An implementation of this algorithm is developed utilizing a heterogeneous GPGPU architecture and will be described in Chapter 5.

The final algorithm discussed as part of this research study in Chapter 3 is the Longest Common Subsequences (LCS) [8] problem that finds the longest common subsequence in a given set of sequences. The solution to the LCS problem uses dynamic programming (i.e. breaking the problem into smaller sub-problems) to obtain the optimal longest common subsequence for the given set of sequences. Not only can the LCS problem be applied to simple problems such as comparing two text files, but can also be used for more complex tasks such as DNA sequence matching [9]. An implementation of this algorithm is developed utilizing a heterogeneous GPGPU architecture and is described in Chapter 5.

Each of the algorithms described above are implemented using the CUDA and MPI models which are discussed in Chapter 4. The implementations that will be discussed include a CPU-only implementation, an MPI (multi-core) implementation, a CUDA (single CPU-GPGPU pair) implementation, and a CUDA-MPI (multiple CPU-GPU pair) implementation. Chapter 6 presents a comparative study of the algorithm implementations using the aforementioned programming paradigms.

The completion of this thesis is structured as follows. Chapter 2 reviews the work that has been completed on the clustering, bioinformatics, and neural network algorithms mentioned above. Chapter 3 provides background on the scientific algorithms used in this research study. Chapter 4 explores the GPGPU architecture and provides an overview of the programming paradigms used in this research study. Chapter 5 provides an overview of the

details of the experimental setup and implementation for each of the algorithm studies. This chapter also includes optimization strategies explored in this research study. Chapter 6 presents the results and analysis for each of the algorithms. Chapter 7 provides conclusions and discusses suggestions for future work.

# Chapter 2

## Related Work

This chapter introduces the work that motivates and supports this research. Each section gives an overview of the work that has been completed for each of the applications discussed in this work as well as the results that were achieved. Section 2.1 discusses implementations of the K-Means algorithm including other multi-core and single-GPU implementations, section 2.2 discusses previous implementations completed using a Spiking Neural Network, and section 2.3 gives an overview of the work that has been done with the Longest Common Subsequences problem in the past.

### 2.1 K-Means

In [10], Farivar et al. present a GPU-based parallel implementation of the K-Means clustering algorithm on an NVIDIA G80 (the NVIDIA 8600GT). When compared to a 3 GHz Intel Pentium(R) processor running the same algorithm, the CUDA implementation is shown to improve 13x over the single processor implementation. The proposed implementation randomly generates the data points as well as the original  $k$  clusters and then in each step re-computes the new cluster centers. The CUDA implementation for this algorithm includes having each thread process single data points, computing the distance

between the point and each centroid. An important aspect of this implementation is the use of constant memory for the centroids. The use of constant memory allows for a faster access through the cacheable part of the device memory. Farivar et al. also make a prediction of a 68x performance increase using the NVIDIA GeForce 8800GT with the same algorithm, but the hardware was not available at the time of publication to verify the assumptions.

In [11], Zechner and Granitzer present an optimized CUDA implementation of the K-Means algorithm on NVIDIA's G80 GPU. To observe how the algorithm performs with differing number of data points, data sets of 500, 5,000, 50,000, and 500,000 were generated with 2, 20, and 200 dimensions. The CPU takes the role of the master thread in this implementation and prepares all of the data points to send to the GPU. The data points are uploaded to the GPU once and then each iteration of the algorithm labels each point as belonging to a specific centroid. Each iteration, the GPU performs the labeling of each point with the nearest centroid and then the results are sent back to the CPU to calculate the new centroid for each cluster. For the GPU-based implementation performance values range from 23 GB/s to 44 GB/s computational performance based on the cluster count and dimensionality.

Hong-tao et al. demonstrate a GPU-based k-means implementation on the NVIDIA G80 architecture, which performs as high as 40x better than the CPU-base k-means implementation in [12]. This paper presents a novel approach to the algorithm (compared to previous implementation), where both the data objects assignment and the centroid recalculation are done on the GPU. Li, Zhao, Chu, and Liu in [13] also present a novel approach for parallelizing k-means on GPUs with two different strategies: one for low-dimensional data sets and one for high-dimensional data sets to achieve best performance.

For low-dimensional data sets, on-chip registers are used on the GPU to decrease latency to access data. For high-dimensional data sets, on-chip shared memory is used to gain a better compute-to-data-access ratio. The algorithm presented is shown to be three to eight times faster than the other GPU-based implementations it was compared to.

The motivation for this thesis can be seen by the results of this literary review. All of the work done previously has shown that the k-means algorithm is a great candidate for accelerations using the GPU. All of the work has been done on a single-GPU; therefore, the research in this thesis continues utilizing the GPU and extends the k-means algorithm to utilize multiple CPU/GPU pairs. The implementation presented in this thesis also utilizes constant memory as well as device functions to further optimize the algorithm.

## **2.2 Izhikevich SNN**

In his well-known paper [14], Izhikevich described the “biological plausibility and computational efficiency” of spiking and bursting neurons and provides models that mimic these properties. Some of these spiking models include “Integrate & Fire” (I&F), I&F with Adaptation, Integrate-and-Fire-or-Burst, Resonate-and-Fire, Spiking Model by Izhikevich, Morris-Lecar [15], Wilson [16], and Hodgkin-Huxley [17]. The rest of the works detailed in this section use one of the aforementioned models to perform neural network simulations.

In [18], Fidjeland and Shanahan make use of the Izhikevich neuron model to perform real-time large-scale simulations of biological brain structures. The GPU implementation in [18] can deliver up to 550 million spikes per second with a single device, which is approximately 55,000 neurons with 1000 synapses per neuron while portraying biologically

accurate conditions in the simulation. Through the use of a GPU kernel a throughput of up to 22 times the original was achieved.

Gupta and Long in [19] use a slightly different approach than previous papers. Instead of using the neural network to perform biological simulations, the spiking neural network model in their research is used to identify characters in a character set. The GPU is not used in their research, however, it is a great example of how SNNs can be used to solve real-world problems. Similar to the research done in this thesis, both excitatory and inhibitory connections are made within the network to train using a known data set.

Han and Taha also present a similar pattern recognition software based on both Izhikevich and Hodgkin-Huxley models in [20]. Three GPU platforms are examined including the GeForce 9800 GX2, the Tesla C1060, and the Tesla S1070. The research presented attempts to prove the efficacy of using the GPU to accelerate a SNN based character recognition networks for large scale systems. The results show accelerations of up to 5.6x (for Izhikevich) and 84.4x (for Hodgkin-Huxley) over the fastest tested CPU (a quadcore 2.67 GHz Xeon processor).

Bhuiyan, Pallipuram, and Smith in [21] investigate optimization techniques as well as performance improvements for SNNs utilizing GPUs and compare the results to a 2.66 GHz Intel Core 2 Quad for the Izhikevich and Hodgkin-Huxley models. These two models are chosen for their study because of the significant differences in computational intensity: the Hodgkin-Huxley model is very computationally intensive whereas the Izhikevich model is much more compute efficient. It is shown that for a small flops/bytes ratio, it is not desirable to offload the computation to the accelerator, however, benefits can be seen when the application has a larger flops/bytes ratio. The Izhikevich model results in about 0.65

flops/byte while the Hodgkin-Huxley model stand at 6.02 flops/byte. Speedups of approximately 9x are achieved by the Izhikevich model where speedups of about 115x are achieved using the Hodgkin Huxley model. The conclusions show that the best speedup over all test cases for the Izhikevich model is the optimized Intel Xeon implementation while the GPU implementation with coalesced global memory accesses and texture lookup proved better for the Hodgkin-Huxley model. Pallipuram, Bhuiyan, and Smith also present a performance analysis comparing NVIDIA's Fermi architecture with AMD's Radeon 5870 using OpenCL in [22]. The four SNN models used for the performance analysis include [15], [16], [17], and [18] with optimization techniques for each algorithm. Speedups of 857x and 658x were achieved on the Fermi and Radeon respectively for the Hodgkin Huxley model with 9.72 million neurons.

All previous work done with SNNs has led to the research completed in this thesis. The research in this thesis provides a simulation of spiking neural networks performed on GPUs with one caveat, the neural network can polychronize. A network that can polychronize "exhibits reproducible time-locked but not synchronous firing patterns with millisecond precision" [23]. The neural network simulation presented in this thesis has this property and is therefore slightly different than previously completed research. It provides a computationally efficient method for simulation as well as reproducibility of biological characteristics found in realistic simulations.

## 2.3 Longest Common Subsequences (LCS)

In [24], Khajeh-Saeed et al. explore the problem of sequences alignment by developing a GPU implementation of the Smith-Waterman algorithm. As mentioned in this research, sequence matching (or sequence alignment) is well-known for its use in testing DNA and protein sequences with large databases. The implementation presented demonstrates the use of up to four GPUs executing the Smith-Waterman algorithm in parallel. The results show that for large enough problems the single-GPU can accelerate the CPU version up to 45x and the speedup linearly scales as the number of GPUs increases (up to 4 GPUs).

McGuinness et al. present a performance study for four very different applications in [25] including the STREAM Benchmark, Smith-Waterman, Graph Analysis, and the Unbalanced Tree Search. Each of these applications is evaluated using single and multiple GPUs and have much different memory needs illustrating the types of scientific applications that could make use of the GPU architecture and its parallelism. For the Smith-Waterman algorithm (a.k.a. the LCS problem), when compared to a single core CPU the speedup is 100x for a single GPU while speedups of 5,335x are achieved when utilizing 120 GPUs.

In [26], Steinbrecher et al. also present a case study that includes the LCS but does not utilize an accelerator, only multi-core systems. The two machines used in the research presented by Steinbrecher include a 12-core and a 4-core machine. The use of techniques such as altering the sequential loops and skewing the loop by changing the linear schedule achieves speedups of approximately 42x. Further optimization such as computing entire rows in one thread led to speedups of approximately 60x.

The concepts described in each of these papers were taken into account when studying the LCS algorithm for this research. In this research both multi-core and multi-GPU implementations are studied compared to the previous works that only include multi-core or single-GPU.

## **2.4        Summary**

In this chapter, an overview of the related work that motivates the use of these algorithms is given. This chapter also provides a brief explanation of the contribution of this research relative to the previously completed work (mentioned at the end of Sections 2.1, 2.2, and 2.3. The next chapter provides a detailed description of the algorithms that were used in this research along with the governing equations used to implement each of the algorithms.

# Chapter 3

## Background

This chapter presents background information on the algorithms used in this research, specifically, k-means clustering, Izhikevich spiking neural networks, and the longest common subsequences problem. The underlying equations for all three algorithms are presented and explained. This chapter is concluded with a discussion of how each of these applications can be used in real-world circumstances.

### 3.1 K-Means Clustering

The k-means clustering algorithm [6] is a method for analyzing clusters of data, typically in data mining applications. The goal of the algorithm is to categorize each data point in a cluster to belong to the cluster with the closest centroid, or mean, to achieve the minimal sum of distances between each point and centroid. More generally, the k-means algorithm attempts to make  $k$  clusters out of the  $n$  observations, or data points, where each observation is a member of the cluster with the nearest mean. The initial centroids can be chosen to be any values within the bounds of the dataset, but there are available methods that lead to better performance of the algorithm. These methods however are not the aim of this research study.

In this study, the initial centroids are defined by choosing pseudo-random data points in the data set.

### 3.1.1 K-Means Algorithm

The k-means algorithm can be described using the following steps:

1. Initialize the points that will be categorized as the initial means.
2. Assign each observation (data point) in the data set to belong to the cluster with the closest centroid.
3. Recalculate the positions of the new centroids based on the new clusters that have been created.
4. Continue steps 2 and 3 until the centroids for each cluster are no longer changing or they are oscillating between a set of points with similar function values.

The k-means algorithm categorizes data points into clusters to minimize the mean distance between all the points; therefore, the main computation in this algorithms involves the distance formula between two points. The main goal of the algorithm can be expressed as an objective function,  $Z$ , utilizing the equation for Euclidean distance (Eq. 3.1):

$$\min Z = \sum_{i=1}^k \sum_{j=1}^n \|x_j - c_i\|^2 \quad (3.1)$$

where  $\|x_j - c_i\|^2$  is the Euclidean distance between the point in question,  $x_j$ , and the centroid of the cluster,  $c_i$ ,  $k$  is the number of clusters, and  $n$  is the number of data observations within the set.

The first step of the algorithm involves initializing the centroids for the first computation. There are numerous ways to initialize these values, some more efficient than others. Some of these methods include:

1. Locate the minimum and maximum data points in the set and initialize the centroids in a way that they are evenly spaced across the domain of the set. This method works well if all data points are somewhat evenly distributed within the domain. However, without doing preprocessing to determine the distribution (which uses computation time), there is not a reasonable method to determine if the points fit this distribution.
2. Use completely random data points (either points inclusive in the data set or just random points located within the domain). This method allows for quick initialization of centroids as well as semi-distributed centroids because of the random number generator. However, randomization could lead to inconsistent performance caused by the distribution of the random values and the choice of initial means. If every execution chooses different initial means, there is no guarantee the algorithm will execute in the same number of clock cycles as it would with a different set of means.
3. Choose initial centroids based on the size of the data set and the number of clusters that are needed. For example, the means of a given set  $\mathbf{S}$  with size  $n$  would have initial means  $k$  given by the equation  $\mathbf{m}_i = \left\{ 0 \quad \frac{n}{k} \cdots \left( n - \frac{2*n}{k} \right) \quad \left( n - \frac{n}{k} \right) \right\}$ , where each  $\mathbf{m}_i$  is an initial mean. This method is used in the study because of its quick computation.

After the initial centroids are determined the next step is to attribute each of the points within the set to the nearest centroid. This is completed by using the Euclidean distance formula to find the distance between every point and each centroid as shown in Eq. 3.2.

$$\mathbf{c}_i = \{\mathbf{x}_c: \|\mathbf{x}_c - \mathbf{m}_i\|^2 \leq \|\mathbf{x}_c - \mathbf{m}_j\|^2 \forall j, 1 \leq j \leq k\} \quad (3.2)$$

Based on Euclidean distance formulas the k-means algorithm determines which points belong to each cluster, also known as the “assignment step”

After each point has been assigned to a cluster, new cluster means should be calculated, also called the “update step”. In this step, the new members of the cluster are taken into account when calculating the new cluster centroid. To calculate the new mean,  $\mathbf{m}_i$ , the observations in each cluster,  $\mathbf{x}_c$ , are summed together and then the total is normalized by the cardinality of the cluster  $|\mathbf{c}_i|$ , given by Eq. 3.3.

$$\mathbf{m}_i^{(t+1)} = \frac{1}{|\mathbf{c}_i|} \sum_{\mathbf{x}_c \in \mathbf{c}_i} \mathbf{x}_c \quad (3.3)$$

Once the new means have been determined, the algorithm can continue. The process of assigning all points to the closest mean and then recalculating each mean is repeated until the centroids in two adjacent iterations of the algorithm are the same. This means that the algorithm has converged to a local optimum solution; however it is not guaranteed by this algorithm that the global optimum solution will be found.

### 3.1.2 K-Means Clustering Real-world Applications

K-means clustering is a somewhat general algorithm; therefore, there are many uses where the algorithm can be utilized to solve real-world problems. In [27], Ray and Turi illustrate that a major disadvantage of the k-means algorithm is that the user must specify the number of clusters,  $k$ , that should be used in the calculation. In this study, segmented images are produced for 2 clusters up to  $k_{max}$  clusters, which is followed by calculations that determine which number of clusters finds the minimum value for the given parameter being measured. This concept could be used to further the research in this thesis by studying a given data set and determining what is the optimal number of clusters to achieve a minimum optimal value.

In [28], Huang derives an extension of the k-means algorithm (called k-modes) that allows for the use of categorical data rather than purely numerical data. In the paper, soybean

disease and credit approval data sets are used to demonstrate the clustering performance of the k-modes algorithm, but many other types of categorical data could be used with the same result. For instance, in automotive manufacturing, there have been strides to predict when faults or warranty claims will occur for vehicles. Models have been constructed based on very large sets of data on each of the cars being produced as well as each car that was returned for a warranty claim. Based on the different types of cars and car parts, categorical data can be constructed and then clustered to detect faulty parts or even faulty manufacturers.

In [29], Oyelade et al. describe a method to utilize k-means clustering to monitor students' academic performance in a higher education academic environment. Oyelade et al. used the created models to predict students' academic performance in English and Mathematics studies. This study provides a method for instructors and institutions to monitor performance of students as well as use the models to improve on future performance of the academic results provided by the institution.

There are numerous applications that can utilize clustering algorithms such as the k-means algorithm. Along with the three uses mentioned, there are still a broad range of applications that benefit from clustering algorithms.

## 3.2 Spiking Neural Networks

Neural networks or artificial neural networks (ANNs) is a paradigm of processing techniques that strive to perform pattern recognition and machine learning types of algorithms, to model the functionality of the biological nervous system, namely the human brain. Simon Haykin described neural networks as “*a massively parallel distributed processor made*

*up of simple processing units, which has a natural propensity for storing experiential knowledge and making it available for use. It resembles the brain in two respects:*

- 1. Knowledge is acquired by the network from its environment through a learning process.*
- 2. Interneuron connection strengths, known as synaptic weights, are used to store the acquired knowledge.”*

in his book [30].

Neural networks model the smallest fundamental component present in the human brain, the *neuron*. Neurons simply use signals from other neurons to determine if they will “fire” or not, meaning that they are active in a given layer of the neural network. Propagation of these signals through multiple layers of the network produces the output, which is how the human brain functions at a high level. In order to accurately represent the human brain and mimic functionality, millions of neurons per network are required, which sometimes is not feasible. Therefore, there are not many models that can accurately represent the human brain activity, but simply mimic the functionality to perform other tasks.

Spiking Neural Networks (SNNs) are the third generation artificial neural networks that attempt to make the modeling more biologically accurate and have a realistic simulation of the human brain. The difference in ANNs and SNNs is that the neurons in an ANN typically “fire” during each propagation cycle while neurons in SNNs only “spike” or “fire” at certain points in time. The properties of the spike including time of the spike are determined solely by the input to the network, which is where information is propagated and processed. Because SNNs have neurons that “spike” only at certain time steps they work very well with applications that incorporate a time component, such as signal processing or image/video recognition.

There are many models that have been developed for SNNs, ranging in computation intensity, complexity, and efficiency. Some of these models include the Izhikevich Model [7], the Wilson Model [16], the Morris-Lecar Model [15], and the Hodgkin Huxley Model [17]. Each of these four models has different properties and complexities and, therefore, each perform differently when implemented. In previous research [31], these models are evaluated and implemented. However in this research, a variation of the Izhikevich model is studied. The overall model, described in the next section, is identical to the original Izhikevich model, however there is one slight difference. The implementation studied in this research allows the spiking neural network to polychronize, which means that the model exhibits reproducible time-locked firing patterns that are not necessarily synchronous. In the following section, the Izhikevich SNN Model is described and the governing equations are defined and explained.

### **3.2.1 Izhikevich SNN Model**

Eugene M. Izhikevich, in [Simple Model of Spiking Neurons], presents a model that is able to replicate the spiking behavior of certain types of cortical neurons. To develop this model, Izhikevich “combined the biological plausibility of [the] Hodgkin-Huxley-type dynamic model and the computationally efficient models [such as the Morris-Lecar and Wilson models]”. By reducing the complexity of the Hodgkin-Huxley model and using computations similar to those of the more efficient models, the computations for the Izhikevich model are able to be completed quicker and therefore used for real-time simulations. The Izhikevich model can be described using the following ordinary equations:

$$v' = 0.04 * v^2 + 5 * v + 140 - u + I \quad (3.4)$$

$$u' = a * (b * v - u) \quad (3.5)$$

$$\text{if } v \geq +30 \text{ mV, then } \left\{ \begin{array}{l} v \leftarrow c \\ u \leftarrow_+ d \end{array} \right\} \quad (3.6)$$

“The variable  $v$  represents the membrane potential of the neuron and  $u$  represents a membrane recovery variable, which accounts for the activation of  $K^+$  ionic currents and inactivation of  $Na^+$  ionic currents, and it provides negative feedback to  $v$ . After the spike reaches its apex (+30 mV), the membrane voltage and the recovery variable are reset according to Eq. 10 [Simple Model of Spiking Neurons].” By selecting the model variables  $a$ ,  $b$ ,  $c$ , and  $d$  the model is able to accurately mimic firing patterns for neurons as well as compute each stage of neurons very quickly. The parameters of the Izhikevich models which govern its behavior are as follows:

- $a$  – describes the time scale of recovery for the variable  $u$  (if smaller values of this parameter, the variable  $u$  “recovers” slower)
- $b$  – describes the sensitivity of the recovery for the variable  $u$
- $c$  – describes the reset value of the membrane potential  $v$  after a spike occurs
- $d$  – describes the reset value of the recovery variable  $u$  after a spike occurs

Selecting these parameters can be described as a large research area to optimize the efficiency of the model, but this is not the aim of this research. The set of parameters chosen for this work is based on the typical values mentioned in [Simple Model of Spiking Neurons].

### 3.2.2 SNN Real-world Applications

Neural Networks have numerous applications that are used every day in the real-world. From speech/audio/video recognition to machine learning, neural networks can be used for a variety of problems. In [32], Ghosh-Dastidar and Adeli present a model using supervised learning with SNNs to classify EEF readings. There are three applications that are used and tested with this model including the XOR problem, the Fisher iris classification problem, and the epilepsy and seizure detection (EEG classification). Using the single-spiking SNN and 82% classification accuracy was achieved for the EEG classification problem while a 90.7%-94.8% accuracy was achieved by the multi-spiking neural network (MuSpiNN).

## 3.3 Longest Common Subsequences Problem

The longest common subsequences (LCS) [8] problem analyzes two sequences and performs comparisons between the two to compute the longest subsequence that is common to both of the sequences. The idea behind this algorithm is to use dynamic programming methods to break the problem into smaller sub-problems to obtain the optimal solution. A mathematical definition of the LCS algorithm can be given by the equation (Eq. 3.7):

$$LCS(\mathbf{s}_i^{(1)}, \mathbf{s}_j^{(2)}) = \begin{cases} \emptyset & \text{if } i = 0 \text{ or } j = 0 \\ LCS(\mathbf{s}_{i-1}^{(1)}, \mathbf{s}_{j-1}^{(2)}) + 1 & \text{if } \mathbf{s}_i^{(1)} = \mathbf{s}_j^{(2)} \\ \max(LCS(\mathbf{s}_i^{(1)}, \mathbf{s}_{j-1}^{(2)}), LCS(\mathbf{s}_{i-1}^{(1)}, \mathbf{s}_j^{(2)})) & \text{if } \mathbf{s}_i^{(1)} \neq \mathbf{s}_j^{(2)} \end{cases} \quad (3.7)$$

where  $\mathbf{s}^{(1)}$  and  $\mathbf{s}^{(2)}$  are the two subsequences and can be defined as the following:

$$\mathbf{s}^{(1|2)} = \{ \mathbf{s}_1^{(1|2)} \quad \mathbf{s}_2^{(1|2)} \dots \mathbf{s}_{(m-1|n-1)}^{(1|2)} \quad \mathbf{s}_{(m|n)}^{(1|2)} \} \quad (3.8)$$

and  $LCS(s_i^{(1)}, s_j^{(2)})$  represents the entire set of longest common subsequences which have prefixes  $s_i^{(1)}$  and  $s_j^{(2)}$ . To find the longest common subsequence, the algorithm simply compares all of the elements  $s_i^{(1)}$  and  $s_j^{(2)}$ . If the two elements are equal (second line in the above equation) then the entire subsequence is extended by that common element. If the two elements are not equal (third line in the above equation), then the longest of the two subsequences  $LCS(s_i^{(1)}, s_{j-1}^{(2)})$  and  $LCS(s_{i-1}^{(1)}, s_j^{(2)})$  is kept.

The longest common subsequences problem uses “traceback” matrices as its main form of displaying the answer. Within these traceback matrices, the length of the longest subsequence can be found as well as the subsequence itself. Without the matrix, only the length would be available at the end of the algorithm. There are often very large problems that use this method to find the solution to a subsequencing problem, but the main drawback is matrices must be stored in memory, which can be costly. Dynamic programming helps in this aspect of the problem. Since the problem is already constructed in a way that it is broken up into subproblems, these very large systems (matrices) can be partitioned into subproblems and the solutions can be constructed from the combination of the results. During the process of computing the LCS score for the set of sequences, additional information can be added to the matrix for the backtracking step of the algorithm. The chosen method for representing the “traceback” elements uses directional arrows which can be determined using the equation (Eq. 8):

$$b(s_i^{(1)}, s_j^{(2)}) = \left\{ \begin{array}{ll} \text{if } s_i^{(1)} = s_j^{(2)} & \\ \text{" } \nearrow \text{"} & \text{if } s_i^{(1)} \neq s_j^{(2)} \text{ and } LCS(s_{i-1}^{(1)}, s_j^{(2)}) \geq LCS(s_i^{(1)}, s_{j-1}^{(2)}) \\ \text{" } \uparrow \text{"} & \\ \text{" } \leftarrow \text{"} & \text{if } s_i^{(1)} \neq s_j^{(2)} \text{ and } LCS(s_{i-1}^{(1)}, s_j^{(2)}) < LCS(s_i^{(1)}, s_{j-1}^{(2)}) \end{array} \right\} \quad (3.9)$$

If the two elements  $s_i^{(1)}$  and  $s_j^{(2)}$  are the same, the arrow simply points to the upper left corner of the matrix. If the two elements  $s_i^{(1)}$  and  $s_j^{(2)}$  are different, the arrow points toward the value either above or to the left with the higher value. After the entire matrix has been filled with traceback elements, the arrows illustrate how to construct the subsequence that was found using the LCS algorithm.

This method of calculation is frequently used with bio-informatics type algorithms; therefore, there are many examples that use “AGCT” (Adenine, Guanine, Cytosine, and Thymine) combinations, which appear in DNA. The following is a simple example that illustrates the traceback matrix concept.

Example: Given the two sequences *GTCAG* and *AGCGA*, use the LCS algorithm and a traceback matrix to compute the longest common subsequence for the two given sequences. For each element  $s_i^{(1)}$  in the sequence, compare it to the corresponding element  $s_j^{(2)}$  in the other sequence. The following table illustrates the completed table for the LCS algorithm including backtracking, which gives the solution *GCG* as the longest common subsequence of *GTCAG* and *AGCGA*.

	0	G	T	C	A	G
0	0	0	0	0	0	0
A	0	↖ 0	↖ 0	↖ 0	↖ 1	← 1
G	0	↖ 1	← 1	← 1	↖ 1	↖ 2
C	0	↖ 1	↖ 1	↖ 2	← 2	↖ 2
G	0	↖ 1	↖ 1	↖ 2	↖ 2	↖ 3
A	0	↖ 1	↖ 1	↖ 2	↖ 3	↖ 3

Figure 3.1 – Example of LCS Algorithm

### 3.3.1 LCS Real-world Applications

The longest common subsequences problem is a general mathematical algorithm, but is most widely used in the field of genetics and biology because it works very well with DNA and RNA sequences. In [33], Bereg et al. develop a model for RNA multiple sequence structural alignment, which is based on the longest common subsequences algorithm. The model presents a polynomial  $O(n^2)$  time algorithm as well as a Maximum Loop Chain algorithm with  $O(n^5)$  time, which investigates many sequences simultaneously using the dynamic programming paradigm found in the LCS algorithm. In [34], Iyer and Saxena investigate the flowshop scheduling problem, which is an algorithm that schedules jobs on an assembly line while minimizing the completion time of the process. The two methods in [34] include using standard implementation of the flowshop scheduling problem and a problem that is tailored using specific information. The LCS algorithm was used in [34] to solve the minimization problem used on an assembly line.

Aside from these, there are many other problems and algorithms that make use of the LCS algorithm. Any time subsequencing is performed in the context of genetics or biological information, most likely the LCS algorithm will be used at some point during the process to compute subsequences.

## 3.4 Summary

In this chapter, an in-depth description of the three algorithms that are used in this research is provided. These algorithms include k-means clustering, Izhikevich spiking neural networks, and the longest common subsequences problem. Each of these algorithms represents a different type of computation and illustrate why the performance study using

heterogeneous systems is relevant. The k-means algorithm involves computation with double precision values including the data points and centroids. The spiking neural network involves computing the next stage spikes and firing patterns for the network. The LCS algorithm involves comparisons between sequences of characters that are represented differently in hardware compared to double precision values. The next chapter provides a detailed description of the computing architecture and programming models use in this research.

## Chapter 4

# Computing Architectures and Programming Models

Over the past few years, multi-core and many-core architectures have become popular technologies used in numerous areas of computational research. Multi-core architectures can be described as systems that contain more than 2 cores and are typically used for general purpose processing rather than parallel processing. Many-core architecture can be described as systems that contain hundreds or thousands of cores and are built specifically to perform parallel tasks. With the advent of multi-core and many-core architectures, programming models such as MPI [35] and CUDA [3] were developed to aid developers in efficiently utilizing the resources available.

This chapter reviews the two different programming models that are used in this research as well as introduces the GPGPU and multi-core architectures as well as mentions the Xeon Phi [1] coprocessor architecture. Clemson University's Palmetto Cluster [36] is also described. (The heterogeneous system used in this research.) Although the Xeon Phi coprocessor was not used for results in this research, it can be used in future work to perform the same types of parallel tasks as the GPGPU.

## 4.1 Computing Architectures

In this research, multi-core and many-core architectures are evaluated in the context of the scientific algorithms described in Chapter 3. In this chapter, the multi-core and many-core architectures will be described along with the corresponding programming models. The Palmetto Cluster will also be detailed as the heterogeneous system used for this research.

### 4.1.1 Multi-Core Architecture

Processors have been around for several decades, but not until the early 2000s did the major processor vendors (Intel, AMD, etc.) begin to realize the need for multi-core processors. Multi-core architectures began with simple two-core (dual-core) designs present in the AMD Phenom II X2 [37] and the Intel Core Duo [38] and have evolved today to incorporate up to ten or more cores. Multi-core processors provide multiprocessing capabilities that allow the user to parallelize applications while only utilizing a single device. Not every application can benefit from the multi-core architectures, however. If the application can be run in parallel (i.e. simultaneously, not sequentially), then the application has a good chance to perform well on these architectures. Today's CPUs comprise numerous advancements over their predecessors that allow for performance improvements including caching, pipelining, wider data paths, superscalar execution, increased transistor density, and increased transistor performance.

In this study, Intel's Xeon E5-2665 [39] will be used as the primary CPU device when performing serial computations and as well as utilizing pairs of them for highly parallel computations without accelerators. This CPU is one of the primary CPUs present in the

Palmetto Cluster, which is used for most of the results. The following section discusses the architecture details of the Xeon E5-2665.

#### 4.1.1.1 Intel Xeon E5-2665

The Xeon E5 [39] series architecture (codenamed Romley) is a 32nm octo-core device with each core based on Intel’s Sandy Bridge-E architecture and runs at 2.4 GHz with a max turbo frequency of 3.1 GHz with overclocking. Each Romley core includes two 32KB, 8-way L1 caches (one for instructions and one for data), a 256KB, 8-way L2 cache, and a 20MB L3 cache. Each socket (LGA2011) allows for up to 2 processors for multiprocessing capabilities. Table 4.1 gives a more concise overview of the important properties of the Xeon E5-2665 architecture. In this study, the Palmetto Cluster [36] will be used with the Intel Xeon E5-2665 in a dual socket configuration creating 16 cores per node with 32 threads.

Table 4.1: Intel Xeon E5-2665 (“Romley”) Statistics

	Intel Xeon
Processor Name	E5-2665
Clock Speed	2.4 GHz
Max Turbo Frequency	3.1 GHz
# of Cores	8
# of Threads	16
Max CPU Configuration	2
L3 Cache	20 MB
Instruction Set	64-bit
Lithography	32 nm
Max Memory Size	384 GB
Memory Types	DDR3-800/1066/1333/1600
# of Memory Channels	4

### 4.1.2 GPGPU Architecture

The advent of fully programmable graphical devices has changed the face of parallel programming. The previous generation of parallel programming involved multi-core processors only, which are capable of parallel computations but lack the inherent parallel nature of today's General Purpose Graphical Processing Units (GPGPUs). Software applications are growing ever larger and more complex; hence there is a need to utilize concurrency (such as that present on a GPGPU) to achieve dramatically increased speed and execution efficiency.

The past decades have seen impressive leaps in GPGPU technology beginning with devices specifically designed for graphical processing to current devices with thousands of cores designed for parallel computations, not limited to graphical processing and image rendering [40]. At its inception, the GPU was used for graphics rendering on personal computers, gaming consoles, and mobile devices. The highly parallel nature of the GPGPU has allowed for a paradigm shift, making the devices much more useful for developing complex software and applications by utilizing the quantity of processor cores compared to a typical CPU.

In 2006, NVIDIA introduced the GeForce 8 series revolutionizing the GPU market bringing to light the massively parallel nature of the GPU and exposing the device as a frontrunner in general-purpose computing. NVIDIA's G80 [41] based GeForce 8800 GTX GPGPU (2006) shown in Figure 4.1 was the first GPGPU architecture to introduce a unified pipeline, which replaced all vertex and pixel pipelines present in older model GPUs. This

generation of GPUs was also the first to utilize Streaming Processors (SPs), simple compute units grouped together within a small area on the device.

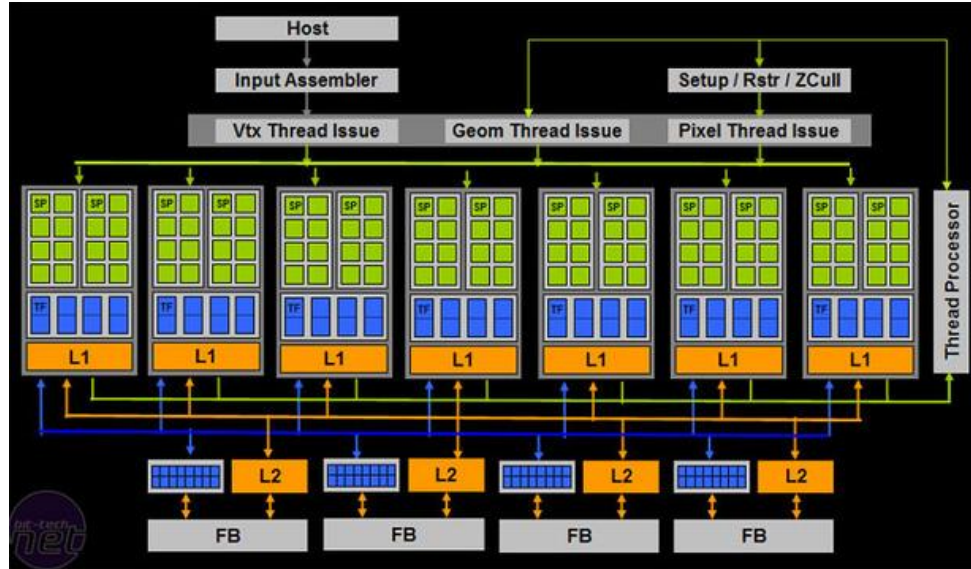


Figure 4.1: NVIDIA GeForce 8800 GTX Architecture [41]

Each SP can produce a result that can either be utilized by other SPs or stored in local memory for later calculations. The advantage of an SP is that similar simultaneous computations can be performed on different SPs on different elements in a data set utilizing the high-speed decode and execute logic present on each SP. A group of SPs is used in the device to execute single instruction multiple data (SIMD) type instructions providing tremendous parallel processing capabilities. Also introduced in the G80 series GPU was the concept of shared memory. Located inside each SP, this fast on-chip memory allows for barrier synchronization and cache-like storage of data for fast retrieval for threads active on a given SP.

Many GPGPUs used in HPC clusters today are still modeled after many of the same features as the G80 (GeForce 8800 GTX) architecture. After the G80 architecture, the next

substantial development for NVIDIA was in 2009 with the introduction of the Fermi [42] architecture shown in Figure 4.2.

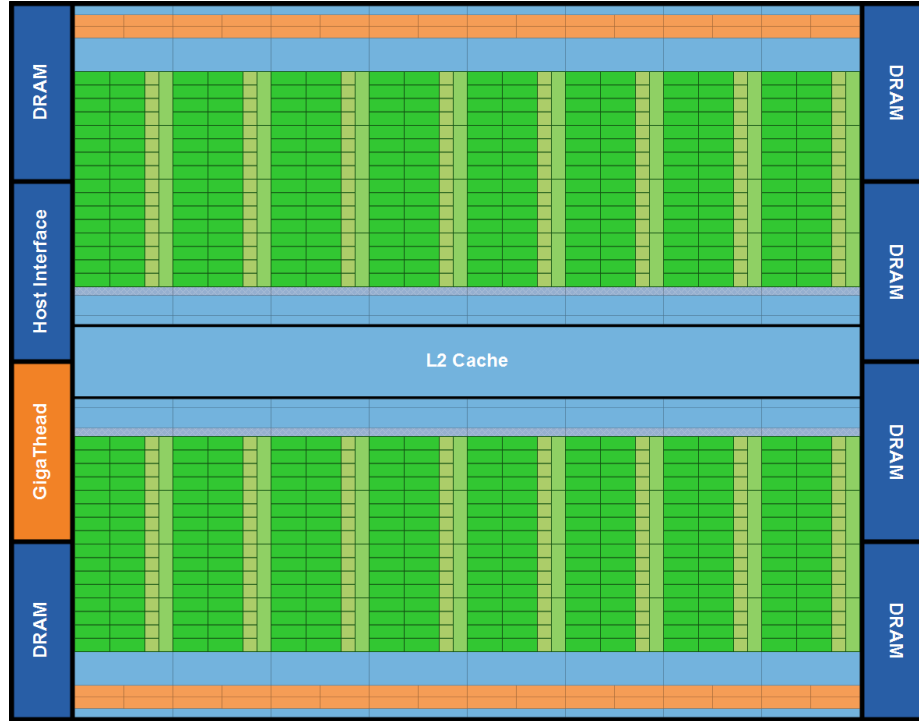


Figure 4.2: NVIDIA Fermi Architecture [42]

The Fermi architecture was an extension of the G80 architecture that included the advent of streaming multiprocessors (SMPs), which incorporates a group of SPs, a double-precision compute unit, and shared memory within the SMP. With the introduction of the Fermi architecture, new terms were coined to explain the GPGPU and how it functioned at a low level. A *thread* is the basic unit of execution in a GPGPU and is executed on a SP within a SMP. Within a SMP, groups of threads, *thread blocks*, are executed on the multiprocessors. The *thread blocks* are further divided into SIMD groups of 32 threads called *warps*, which can also be divided into groups of 16 threads called half-warps.

The first Fermi based GPU contained 16 SMPs, each containing a total of 32 cores creating a device total of 512 CUDA cores. Each SMP is equipped with both an integer

arithmetic logic unit (ALU) and a floating point unit; therefore, in each clock cycle, each SMP can execute either a floating point or integer instruction. The Fermi architecture also included a dual warp scheduler allowing for two warps to be scheduled on the device simultaneously.

The most recent development in the GPU architecture made by NVIDIA is the Kepler [Reference: Kepler Architecture] architecture. This architecture was used in this research and will be explained in the subsequent section along with details about the specific device used.

#### 4.1.2.1 NVIDIA Tesla K20 (Kepler GK110)

In 2012, NVIDIA introduced the Kepler GK110 [43] architecture offering the GPU market a significant improvement over the previous Fermi architecture including improvements in compute capabilities and performance and reduced power consumption. Of the GK110 architecture (Figure 4.3) devices, the K20 and K20X GPUs are two of the most popular devices being used in machines and clusters today.



Figure 4.3: NVIDIA Kepler GK110 Architecture [43]

The GK110 Kepler GPUs typically have 5GB of GDDR5 memory, with a GPU clock speed of 706 MHz and a memory clock speed of 1300 MHz. Each GK110 GPU supports CUDA compute capability 3.5. The Kepler GK110 contains *Next Generation Streaming Multiprocessors* (SMX), which provide astounding performance improvements while lowering the necessary power consumption compared to earlier generations of the GPU containing SMPs. Each SMX (Figure 4.4) inside a Kepler GK110 contains 192 single-precision CUDA cores, while the SMX still holds the ability for single and double-precision arithmetic computations. The Kepler family of GPUs can support a total of 16 SMX units per block, but all GK110 devices do not contain the maximum number of SMX units. For example, the NVIDIA Tesla K20 (the device used in this research) contains 13 SMX units, meaning the device contains a total of 2496 CUDA cores instead of the maximum 3072 CUDA cores.

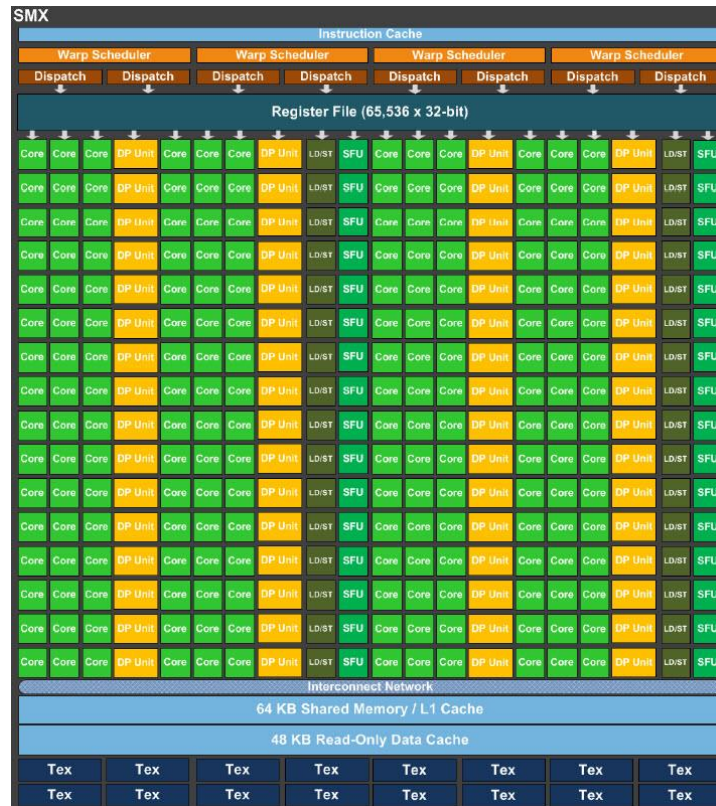


Figure 4.4: GK110 SMX Architecture [43]

The Kepler GK110 GPU has a warp size of 32 threads and supports up to 64 warps per SMX. This architecture also includes a quad warp scheduler (compared to the dual warp scheduler of the Fermi), which allows for a total of four warps to be executed concurrently. Other important information about the GK110 architecture (in particular, the NVIDIA Tesla K20) can be found in Table 4.2.

The Kepler GK110 also has a few features that are altogether new to the realm of GPU programming including Hyper Q and dynamic parallelism. Hyper Q [44] enables multiple CPU threads to offload tasks to the GPU simultaneously, which creates less idle time for the CPU while increasing the utilization of the GPU. Dynamic parallelism allows the developer to have an application directly launched by the GPU instead of going through the CPU as a middle-man. This allows for more effective load balancing on the GPU as well as lower communication times between the host and device.

Table 4.2: Kepler GK110 (Tesla K20) Statistics

	NVIDIA Tesla K20
GPU Name	GK110
Processor Size	28 nm
GPU Clock Speed	706 MHz
Memory Clock Speed	1300 MHz
Memory Size	5120 MB (GDDR5)
Memory Width	320-bits
Threads/Warp	32
Max Warps/SMX	64
Max Threads/SMX	16
Max Thread Blocks/SMX	2048
Max Registers/Thread	255
SMX Memory (Local)	64 KB (48 KB shared/16 KB L1 cache or 16 KB shared/48 KB L1 cache)
# of CUDA Cores	2496
Single Precision Performance	3.52 teraflops
Double Precision Performance	1.17 teraflops

## 4.2 Programming Models

This section introduces the programming models used in this study: MPI [35] and CUDA [3]. Message Passing Interface (MPI) is a standardized “message-passing” system design that allows programmers to utilize the parallel nature of modern processors and processor cores. This provides programmers with the ability to parallelize applications and algorithms when only multi-core processors are available. Compute Unified Device Architecture (CUDA) is a widely used programming platform and framework for parallel computing on GPGPUs. CUDA is designed for use with NVIDIA GPGPUS, and furthermore only supports NVIDIA GPGPUs. For a more generic solution, a programmer could use a model such as OpenCL [2], which supports a wide variety of HPC architectures such as GPGPUs and FPGAs while not being vendor specific. CUDA was used in this research study instead of OpenCL or another generic models because of the availability of NVIDIA GPGPUs.

### 4.2.1 MPI

One of the more popular techniques for utilizing multi-core processors is to “scatter” the data and computations to multiple processor cores to achieve a faster result. Message Passing Interface (MPI) [35] is a standardized system designed by collaborators in academia and industry to allow “message-passing” in a parallel environment. MPI provides system calls and functions for users to easily parallelize computations and spread data across a multi-core system. MPI allows users of a wide variety of programming languages including Fortran, C, C++, and Java to utilize the parallel functionality of message passing. Table 4.3 gives a few

of the commonly used functions and their descriptions to explain the premise of the programming language.

Table 4.3: Commonly used MPI Functions

Function	Description
MPI_Send	Send a buffer of data to another process
MPI_Recv	Receives a buffer of data from another process
MPI_Bcast	Send a buffer of data to every process from “root” process
MPI_Scatter	Send a buffer from one process to all other processes (dissemination)
MPI_Gather	Gather together values from a group of processes
MPI_Init	Initialize MPI execution environment
MPI_Finalize	Terminate MPI execution environment
MPI_Allreduce	Performs a specified operation on each processes set of data (i.e. addition)

## 4.2.2 CUDA

The Compute Unified Device Architecture (CUDA) [3] showcases the power of GPGPUs by providing the programmer a C-like experience when developing. CUDA was introduced in 2007 by NVIDIA to be a single programming language for use with NVIDIA GPUs. CUDA is vendor specific (meaning that it is only possible to utilize CUDA on NVIDIA GPUs) however most details in the CUDA programming language have been optimized to work well with its family of GPUs. OpenCL is another language that can be used to program any type of GPU, but there are limited optimizations performed for architectures because it is designed for being open source and working on a multitude of devices.

The CUDA architecture, being architecture specific, is able to exploit all specialties of the NVIDIA GPU including shared and textured memory and utilizing all of the processing

(CUDA) cores. In CUDA, the code that is executed on the device is known as a *kernel*. Kernels are C-like functions (with CUDA specific directives) that are executed in parallel by utilizing every CUDA core on the device. In most situations, one kernel is executed on the device followed by another in a sequential fashion. In newer models of NVIDIA GPUs, it is possible to launch multiple kernels simultaneously, but this is not studied in this research.

Each *thread* that is created for a given application executes the kernel in parallel. CUDA executes threads in groups called *thread blocks* as a grid in either one-, two-, or three-dimensions. Each *thread block* is executed on a separate SMX, which are grouped into 32 thread groups called warps. The threads inside each thread block can be accessed through device parameters *threadIdx*, *blockIdx*, and *blockDim*, which give the programmer the ability to access any thread by using its global index. The *threadIdx* variable specifies the thread index within a given block, *blockIdx* gives the number of the current block, and the number of threads per dimension is given by the variable *blockDim*. When a CUDA kernel is launched, the information about number of blocks and number of threads per block is defined and setup on the device. Figure 4.5 shows the CUDA thread hierarchy and how the threads and blocks inside a kernel interact.

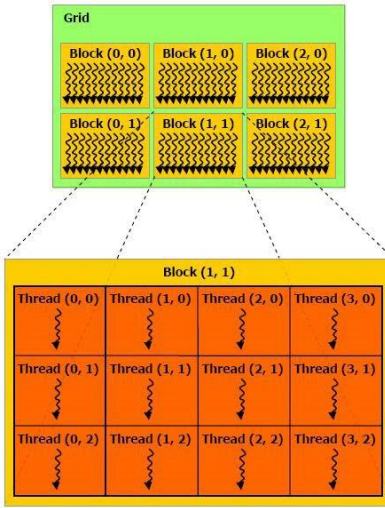


Figure 4.5: Thread block layout in CUDA [3]

The CUDA programming model provides a wide range of memory types for which each thread has the opportunity to access throughout the kernel execution. At the per-thread level, there is local memory and registers that each thread can access for a private memory location separate from other threads. Even though the local memory is only accessible to a single thread it actually resides in the external device memory (global memory) and is therefore slower than other types of memory. All threads in a thread block share a bank of memory, coincidentally named *shared memory*, in which each thread in the block can access and modify. Thread synchronization is needed for shared memory as well because multiple threads could possibly change the same memory location. In addition to the shared and local memory, there is also *global memory* and *constant memory*. Global memory is the largest of the memory banks on the GPU device (off-chip memory) and is accessible to any thread in the kernel. The constant memory is also accessible to all threads active in a kernel however it is read-only memory so it cannot be modified. To make constant memory accesses faster, because the memory locations are read-only and it resides in the global memory, the values in memory are cached and

therefore only take a few clock cycles to retrieve rather than a few hundred retrieving data from the global memory.

There are many optimization techniques that can be used when performing GPU programming using CUDA including memory optimization, varying execution configurations, and instruction optimizations. Rather than mentioning all optimization techniques for CUDA, the particular optimization techniques and implementation strategies that are used for the applications mentioned in Chapter 3 will be described in detail in Chapter 5.

## 4.3 Palmetto Cluster

For the purposes of this research, the Palmetto Cluster [36] is used as the heterogeneous computing platform. This cluster allows the user to specify whether a CPU-only implementation is sufficient or whether a homogenous system utilizing CPU-GPU pairs is necessary. Each node (that was utilized) in the cluster is an HP SL250s containing two Intel Xeon E5-2665 (see Section 4.1.1.1) processors with 16 cores and 64GB of RAM together on the machine. To utilize all cores on a given node, MPI is used to handle all communications and transfers. Along with the pair of processors on each node, they are also equipped with 2 NVIDIA Tesla K20 GPU devices. For applications that will utilize the GPU, it is possible to use open source languages such as OpenCL to program the GPU, but for the purposes of this study, CUDA is the only programming language used.

For applications that can utilize multiple GPUs to perform the computations, other strategies can be employed. The CUDA-MPI programming model can be applied to distribute the data to each of the GPUs to perform a larger scale parallel computation. The same idea is applied for programming on a single CPU-GPU pair except the data that will be computed

will be divided onto different nodes so the processing can be performed in parallel. The communication between the CPU and GPU is done over the PCI-Ex bus while the communication between nodes uses 56g Infiniband [4].

When retrieving nodes for computation, even if all 16 cores on a node will not be utilized, the entire node should be allocated. If only a few cores are allocated, the PBS scheduler could schedule other jobs to occur on the same node using the other cores altering runtime of your application, leading to irrelevant or outlying results. The Palmetto cluster is used for all implementations including MPI-only (CPU-only), CUDA-only (single CPU-GPU pair), and CUDA-MPI (multiple CPU-GPU pairs).

## **4.4 Summary**

In this chapter, the computing architectures used in this research (namely, Intel Xeon E5-2665 and NVIDIA Tesla K20) were discussed as well as the programming models (MPI and CUDA) used to program these devices. A discussion of the Palmetto Cluster was also given explaining the heterogeneous nature of the system. Chapter 5 will present a detailed description of the single-GPU and multi-GPU setup for each algorithm as well as implementation details about the applications used to exploit features of the architectures.

## Chapter 5

# Experimental Setup and Implementation

This chapter presents the single-GPU and multi-GPU setup for each of the algorithms described in Chapter 3. This chapter also discusses the features of the GPU architecture used in this study and the different implementations that are used to exploit these features. This chapter concludes with a detailed section on the parallelization of each algorithm and how they were mapped successfully to single-GPU and multi-GPU systems.

### 5.1 Experimental Setup

In this section, the experimental setup for this research is discussed. The experimental setup includes the setup of the heterogeneous system on which each application is executed as well as the compiler and runtime environment that is utilized. Section 5.1.1 describes the single-GPU and multi-GPU experimental setup and section 5.1.2 describes the compiler and runtime environment used in this research.

### 5.1.1 Heterogeneous System Setup

The single-GPU heterogeneous system setup consists of a 2.4 GHz Intel Xeon E5-2665 [39] host processor coupled with a single NVIDIA K20m (Kepler GK110) [43]. The multi-GPU heterogeneous system setup includes multiple 2.4 GHz Intel Xeon E5-2665 host processors along with an equivalent number of K20m GPUs (1:1 ratio). On the Palmetto Cluster [36], each node equipped with K20m GPUs is designed to have 2 host processors paired with 2 GPGPUs. To incorporate larger heterogeneous systems, these nodes can be agglomerated together utilizing the two CPUs and GPUs on each node. For instance, to create a system with 16 GPUs, 8 nodes are initialized (each with two host processors and two GPGPUs).

All algorithms discussed in this research were developed using CUDA 5 installed on the host system running Scientific Linux. Table 5.1 illustrates some of the features offered by the accelerators located on the Palmetto Cluster. The results for this table were acquired by using NVIDIA's *deviceQuery* utility, which allows developers to view device properties. The two GPUs on each node of the cluster are identical, therefore it does not matter which device the CUDA runtime environment chooses to be Device 0 (default first device).

### 5.1.2 Compiler

All source files that contain CUDA kernels and any other device functions that utilize the extension `.cu` and are compiled using the *nvcc* compiler [45]. The *nvcc* compiler automatically calls all other necessary compilers and tools to compile the device code and create a device executable. All CUDA code requires the CUDA runtime library (*cuda*) along

with the CUDA core library (*cuda*). Both are necessary for a complete compilation of device code.

Table 5.1: Features available on the NVIDIA Tesla K20m device

Features	NVIDIA K20m
CUDA Driver Version / Runtime Version	6.0 / 5.5
CUDA Compute Capability	3.5
Total Global Memory	4800 MB
Memory Clock Rate	2600 MHz (2.6 GHz)
Memory Bus Width	320-bit
Number of Multiprocessors	13
Number of CUDA Cores / MP	192
Total Number of Cores	2946
L2 Cache Size	1280 KB
Total Constant Memory	64 KB
Total Shared Memory / Block	48 KB
Total Registers / Block	65536
Warp Size	32
Max Number of Threads / MP	2048
Max Number of Threads / Block	1024
Max Dimension of Thread Block	1024 x 1024 x 64
Max Dimension Size of Grid	2147483647 x 65535 x 65535
Clock Rate	706 MHz ( 0.71 GHz)
Concurrent Copy and Kernel Execution	Yes (2 copy engines)

## 5.2 Implementations

In this section, the implementations for each of the algorithms described in Chapter 3 are discussed. Some of the optimization strategies that include device specific information are described multiple times for each algorithm simply because each algorithm lends itself slightly differently to each optimization technique. Described below are the optimization techniques and input information used to obtain the results described later in Chapter 6.

## 5.2.1 K-Means

Table 5.2 provides information about the test cases created in this research. To create test cases, there are many ways to generate input data sets including structured data points and random data points. In this research, the data sets were created to be in structured format meaning each data set was constructed with the idea of belonging to a single cluster and only a single cluster. Therefore, each data set only has a single “best/minimum” solution. If the data had been generated randomly, depending on the starting values for each mean of the clusters, the algorithm may not converge to a global minimum, but will return a local minimum value as the result. This behavior could cause the results to vary dramatically based on the starting values (with random data points), causing each execution of the algorithm to be unpredictable. For this reason, the data was structured in a way to perform similarly every time regardless of the starting values allowing for a fair comparison of the multi-core implementation with the GPU implementations.

Table 5.2: Data Configurations for K-Means Clustering

Input Data Size (Number of Data Points)	Number of Clusters			
	0.4%	1.0%	2.0%	4.0%
1000	4	10	20	40
2500	10	25	50	100
5000	20	50	100	200
7500	30	75	150	300
10000	40	100	200	400
25000	100	250	500	1000
50000	200	500	1000	2000
75000	300	750	1500	3000
100000	400	1000	2000	4000
250000	1000	2500	5000	10000
500000	2000	5000	10000	20000
750000	3000	7500	15000	30000

As seen in Table 5.2, the input data size for the test cases generated range from 1,000 data points to 750,000 data points. For each data size there are four test cases (number of clusters) based on the number of data points. The number of clusters is determined by taking a percentage of the entire data set size (i.e. 0.4%, 1%, 2%, and 4%).

## 5.2.2 Spiking Neural Networks

The spiking neural network application used in this research is slightly different from the other two applications. Unlike the other two applications discussed, this application was originally written in MATLAB. Therefore instead of changing the input data and parameters, this application was converted to C/C++ and then the C implementation was used to construct a GPGPU implementation. Table 5.3 provides details on the parameters used in the original algorithm (and in turn, the converted implementations).

Table 5.3: Data Configuration for Izhikevich SNN Algorithm

Parameter	Value
Excitatory Neurons	800
Inhibitory Neurons	200
Total Neurons	1000
Synapses/Neuron	100
Max Axonal Conduction Delay	20
Max Synaptic Strength	10
Max Fired Neurons/Second	100000

The excitatory neurons form long-term connections within the networks while the inhibitory neurons make local connections. The pulsing of these excitatory (+) and inhibitory (-) synapses will determine how the neurons will fire along the axon. The maximum axonal conduction delay and maximum synaptic strength are characteristics that model physical phenomena about each neuron. Also, a value of 100,000 is set as the maximum number of

neurons allowed to fire every second. Changing this value can change the behavior of the entire network.

### 5.2.3 Longest Common Subsequences

Table 5.4 provides information about the test cases created for the longest common subsequences problem. Initially  $n$  number of sequences were generated and placed in a file and for each test case, a fraction of the input file was used for the testing. Each of the  $n$  sequences was generated using a minimum length of 30 and a maximum length of MAX\_LENGTH, which varied from 50 to 500 to monitor the performance.

Table 5.4: Data Configurations for LCS Algorithm

Number of Sequences	Maximum Length of Sequence			
	50	100	250	500
5	50	100	250	500
10	50	100	250	500
20	50	100	250	500
25	50	100	250	500
50	50	100	250	500
75	50	100	250	500
100	50	100	250	500

Each sequence has a length between the minimum and maximum values and contains the number of characters between  $a$  and  $z$ . The algorithm then uses a subset of the entire collection of sequences as its input data to perform the algorithm described in Chapter 3. An example of an input sequence (with length 52) generated could be as follows:

**aopueqdnirpiscphywkatcgknvrqfhwuccoobszgqlmnmhvoscq**

After each sequence is generated the algorithm then uses subsequences to compute comparisons. The algorithm generates similarity values and alignment sequences, which

illustrate the maximal alignment of the two subsequences. The final result reveals the alignment comparisons between each sequence with every other sequence in the set.

## **5.3 GPU Implementations**

In this section, the GPU implementation details will be described. The techniques described in sections 5.3.1, 5.3.2, and 5.3.3 are similar across all applications although some have minor differences because of algorithmic restrictions. As described in Chapter 4, there are many different types of memory (with different latencies) as well as other details of the GPU architecture that can be utilized to accelerate the execution of each application. The following implementation strategies illustrate some of the possible acceleration techniques that can be used with the GPGPU. Implementation 1 describes converting the serial CPU-only implementation into a GPU implementation. This is a very naïve approach simply because it only uses global memory, which can lead to high latency. Implementation 2 makes use of constant memory, which is located within the global memory but is cached, therefore leading to faster access times for the data. Implementation 3 utilizes the same techniques as implementation 2, however also utilizes built-in mathematical operations where possible. Because the built-in math operations have been designed to execute relatively fast on the hardware, theoretically they should be faster than user defined functions to perform the same task.

### **5.3.1 Implementation 1 (Global Memory)**

Implementation 1 simply converts the traditional CPU-only code into a naïve implementation for the GPGPU. For implementation 1, all data is placed into the global memory of the device and then accessed without moving to any other types of memory. Since

global memory is off chip and not cached, each memory access is lengthy and will cause the application to be slower than an optimized version. Other implementations improve on this fact, attempting to decrease memory latency by utilizing cached memory, which will cut down on the significant overhead required to access the high latency global memory.

### *K-Means*

For the k-means algorithm, the data points as well as the initial centroids (which will be calculated on the CPU because it is a very small computation) will be placed into global memory on the device. Because of the configurations described in Table 5.2, the entire data set as well as centroids will fit in global memory, therefore leading to only one host to device transfer. For larger data sets the data points and centroids would require partitioning before transfer to the GPU and partitioned execution causing multiple kernel calls, in turn slowing down the execution of the entire algorithm. For each iteration of the algorithm, new means are calculated by going through each of the data points and the cluster to which it belongs. New means are created until either they are no longer changing between iterations or the means are oscillating between a set of points. For accurate performance comparisons, the input data is created in a deterministic fashion so the new means will not oscillate causing inaccurate performance results.

The performance improvement over the CPU-only version will be achieved simply by the parallelization of the computations. Each data point belongs to a cluster and does not depend on any other data point; therefore, each data point can be dealt with independently (in parallel). Each thread inside the GPGPU kernel will calculate a small portion of the result (i.e. compute the new mean as well as the data points that belong to that mean).

### *Spiking Neural Network*

The spiking neural network algorithm presented in this research has many possible sections for parallelization that update the firing patterns and the activity variables and many other parameters described in Table 5.3. In the initialization step of the algorithm, when the synaptic weights and inhibitory delays are calculated and set, there are nested *for* loops, which is the very first thing to look for when parallelizing an application. To parallelize the loops, the innermost *for* loop can be unrolled and each iteration can be calculated by a different thread while the outermost loop can utilize thread blocks for each of its iterations. During the training phase of the algorithm there are nested loops in which the same manner of parallelization can be utilized. The overall output of the trained system will be the firing rate for the neurons as well as the indices and timing of each of the spikes that occur throughout the process.

### *Longest Common Subsequences*

The longest common subsequences problem, similar to the k-means algorithm, has a very large percentage of its computations completed on the GPGPU device because of the inherent parallel nature of the application. Each of the sequences that is to be tested is loaded into the global memory of the GPGPU device along with the sizes and starting positions of each sequence within the collection of sequences. The data sizes to be tested are displayed in Table 5.4. The larger data sets give the GPGPU a better opportunity to outperform the CPU implementation because of more data to process and many more computations. Having substantially more computations is beneficial for the GPGPU when compared to the CPU because it is inherently parallel and with more computation, the latency of memory operations can be overlooked or hidden with enough computation.

As with the k-means algorithm, performance improvement will be achieved in this implementation simply by the parallelization of the computations on the GPGPU device. Each thread or block inside the GPGPU will calculate a small set of subsequence similarities and then place the results in global memory for final processing. This operation can also be accelerated with the use of optimization techniques described in the following two implementation strategies.

### **5.3.2 Implementation 2 (Constant Memory)**

Implementation 2 is a continuation of implementation 1 with the addition of the use of device constant memory. Constant memory is used for data that will not change throughout the course of the kernel execution. Constant memory, however residing inside the global memory of the device is cached, therefore leading to much faster access of the data. To declare constant memory, the programmer must simply use the `__constant__` keyword.

In the k-means algorithm, the data points are the same throughout the execution of the algorithm. Therefore, the entire set of data points can be placed into constant memory on the device for faster access. For the Izhikevich spiking neural network algorithm, there are many candidates that can be placed in constant memory on the device. Some of these include arrays of postsynaptic neurons, synaptic weights and their derivative, distributions of delays, and numerous others. The tradeoff for this algorithm as well as the k-means algorithm is the size of the constant memory on the device. As mentioned in Table 5.1 there is only 64KB of constant memory on the device. Therefore, entire data sets or arrays will not fit into constant memory for access. Therefore, a tradeoff must be made between moving data to the constant memory and accessing it in the global memory. In this research as much data as possible is

moved into the constant memory for each iteration of each algorithm and then the remaining data is simply accessed through global memory instead of repopulating the constant memory. The data chosen to be placed in the constant memory is the data that was found to be used most in one iteration of the algorithm.

In contrast, the longest common subsequences problem has a very small input data set and therefore the entire set of input characters and sequence positions can be placed in constant memory. This means that all of the global memory access times will be negated in this algorithm implementation, leading to substantially better execution times.

### **5.3.3 Implementation 3 (GPGPU Math Library)**

Implementation 3 is an attempt to improve slightly on the already accelerated implementation 2. There are many commonly used mathematical functions defined in the CUDA Toolkit [46] that have been optimized for NVIDIA hardware, therefore executing in fewer clock cycles than user-defined device functions. Wherever applicable in each algorithm implementation, user defined functions or typical C programming functions are replaced with the hardware accelerated functions for slightly better performance.

For example, in the k-means algorithm, the `max` function, the `absolute_value` function, and the `square_root` function are used in each iteration of the calculation. Therefore, by replacing each of these with the GPGPU device math library function, a performance improvement should be observed, however slight for smaller data sets. For the longest common subsequences problem, the `max` and the `power` functions are used in every iteration of the algorithm and can be replaced by the equivalent GPGPU device functions for improved/more efficient device performance. Unfortunately, all of the mathematical

operations for the spiking neural network algorithm, although many in number, are quite simple mathematical operations (i.e. addition, multiplication, etc.). Therefore, there is no need to try and accelerate any of these mathematical operations for this algorithm; although to make the research complete, the device functions are utilized.

### **5.3.4 Multi-GPU Implementations**

The above implementation details illustrate the single-GPU implementations. However, in this research, multi-GPU systems and implementations are developed and studied. The implementation details are exactly the same except for inherent need to spread the data across nodes. The data is still parallelized on the GPGPU, but before the computation is done, the data is partitioned and distributed to different nodes. This means that each node will be operating on smaller chunks of the data and, in theory, execute faster.

The k-means algorithm is partitioned based on the data points and centroids. There are far fewer centroids than there are data points, therefore, all centroids can be copied to every node's memory. The data points are partitioned so that each node has a similar amount of data points. For most test cases, each node receives the same number of data points, but for cases with an odd number of data points, the data is partitioned in a way so that each node has a similar number of data points. The data points do not change throughout the algorithm, so the only communication between nodes during each iteration is the new centroids that are calculated.

The Spiking Neural Network algorithm is partitioned based on the neurons in the network. The neurons in the network are split evenly across the nodes so that the computations on each set of neurons can be calculated in parallel.

The Longest Common Subsequences problem is very simple to partition into a multi-GPU system for computation. The sequences are predetermined in the initialization phase of the algorithm and are not changed throughout the algorithm, therefore, each node is given a subset of the entire data set for computation. Once each node has completed the computation on the sequences it was given, the results are then collected on one node and compiled for final display.

## **5.4 Summary**

In this chapter, the setup for each of the algorithms described in Chapter 3 has been presented for the single-GPU and multi-GPU implementations. The implementation details have been presented for each algorithm as well as the different optimizations for each. Parallelization of each algorithm and the mapping of each algorithm to the GPGPU resource has been described. Chapter 6 will present a detailed discussion of the results and analysis following the setup of each algorithmic experiment.

# Chapter 6

## Results and Analysis

In Chapter 2, the computing architectures and programming models used in this research are described while Chapter 5 describes the various implementations of the algorithm details in Chapter 3. In this chapter, the implementations of each algorithm are analyzed and the results are given. Initially the single-core implementation results are shown as a baseline for comparisons of other implementations of the algorithm. Following the single-core implementation are the multi-core, single-GPU, and multi-GPU implementation results. The performance of the multi-core and single-GPU implementations are evaluated by comparing the execution times of the single-core implementation. It would not, however, be a fair comparison to compare the multi-GPU implementation to the single-core implementation, so this particular implementation is compared to the multi-core implementation. These comparisons will show how much performance the GPU adds to the computation as an accelerator compared to a traditional CPU. The comparison will also be made, as mentioned before, in terms of how much the algorithm “speeds up” compared to other implementations. Speedups for each algorithm are given in its corresponding final subsection.

## 6.1 K-Means

### 6.1.1 Single-Core Implementation

As mentioned in Section 5.2.1 and Table 5.2, the data sets for the K-Means algorithm range from 1,000 data points to 750,000 data points while the cluster configurations consist of 0.4%, 1%, 2%, and 4% of the data size. Table 6.1 shows the execution time of the single-core version of the algorithm and Figure 6.1 illustrates the execution times. It can be seen in Table 6.1 as well as Figure 6.1 that as the cluster configuration percentage increases, the execution time also increases. For smaller data sizes, this time is much less noticeable because the entire algorithm executes in fractions of a second. However, with larger data sizes, the difference in execution times is apparent when increasing the percentage of the points that are centroids. The increased execution time is an intuitive finding however, because the algorithm works on locating centroids and calculating for the next iteration. With more centroids there would be more calculation, leading to higher execution times.

As previously mentioned, as the input size is increased for most algorithms, the execution time will also increase, which is not a new observation. Also, Figure 6.1 illustrates that as the cluster configuration (percentage of centroids) increases, the execution time increases as well (at approximately the same rate). For example for a data set of size 750,000 data points for the 2% configuration (15,000 centroids) the execution time is around 300 seconds while for the 4% configuration (30,000 centroids) the execution time is around 600 seconds. This means that as the number of centroids in the algorithm is increased, the execution time of the algorithm increases linearly.

Table 6.1 – Execution Times for Single-Core K-Means Algorithm

Input Data Size (Number of Data Points)	Execution Time(s)			
	0.40%	1%	2%	4%
1000	0.032	0.01	0.003	0.005
2500	0.033	0.005	0.007	0.009
5000	0.084	0.018	0.032	0.031
7500	0.017	0.037	0.069	0.068
10000	0.029	0.064	0.061	0.119
25000	0.156	0.371	0.367	0.724
50000	0.598	1.458	1.447	2.884
75000	1.662	1.639	3.216	6.249
100000	2.841	2.804	5.572	11.101
250000	14.007	17.694	35.933	71.744
500000	57.629	72.029	142.971	285.797
750000	126.443	156.65	312.294	621.724

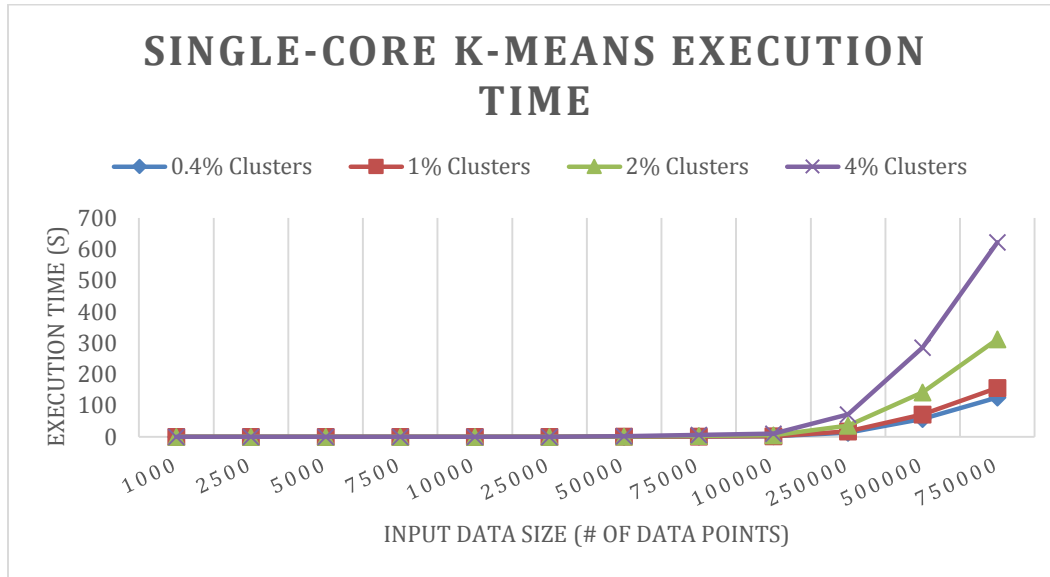


Figure 6.1 - Execution Times for Single-Core K-Means Algorithm

However, through the advent of accelerators (GPGPUs) and heterogeneous systems (multi-core with GPGPUs), the execution times for the algorithm can be greatly improved, achieving super linear performance. Sections 6.1.2, 6.1.3, and 6.1.4 explore the results when the K-Means algorithm utilizes accelerators and multi-core systems to achieve better performance (i.e. better overall runtime).

## 6.1.2 Multi-Core Implementation

After studying the K-Means algorithm, it can be seen that the bulk of the computation is done when computing which centroid each one of the data points belongs to as well as computing the new centroids. These two methods will be parallelized in the multi-core implementation in order to study its performance. For succinctness (so as not to have a graph for every input data size), only the largest value used in the single-core implementation will be used, 750,000 data points. By using the largest data size, a direct comparison can be made between the multi-core and single-core implementations. Smaller data sets would be ill suited for the multi-core system because most of the execution time would be dominated by data communication and pre-processing. Table 6.2 shows the execution times for the multi-core implementation of the K-Means algorithm while Figure 6.2 illustrates these execution times.

Table 6.2 – Execution Times for Multi-Core K-Means Algorithm

Number of Nodes	Execution Time (s)			
	0.40%	1%	2%	4%
1	126.443	156.65	312.294	621.724
2	69.823	91.237	171.3	291.52
4	35.127	43.93	92.832	163.285
8	18.928	25.873	49.661	81.92
16	7.293	13.375	24.51	43.791
32	3.39	6.118	13.269	18.14

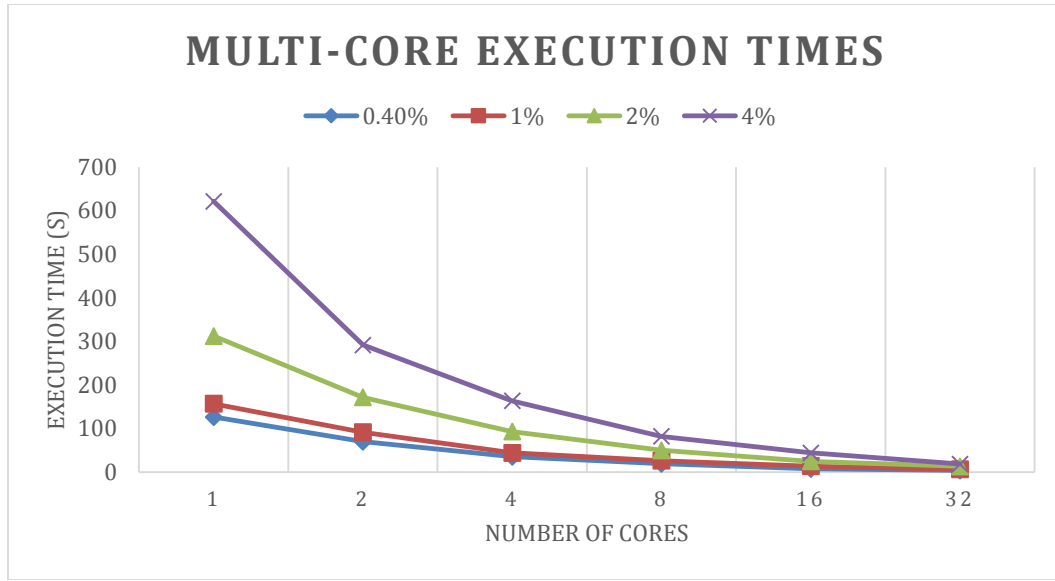


Figure 6.2 – Execution Times for Multi-core K-Means Algorithm

Figure 6.2 illustrates that as the number of cores increases, the overall execution time of the algorithm decreases for a given input configuration. As expected, the multi-core system allows parallelization of the independent sections of the algorithm thereby achieving a faster result.

### 6.1.3 Single-GPU Implementation

As with testing on the multi-core system, only the largest data size is used when evaluating the single-GPU implementations. Table 6.3 shows the execution times of the single-GPU implementations of the K-Means algorithm. Figure 6.3 illustrates the runtimes of the single-GPU implementations utilizing optimization techniques described in Section 5.3. Implementation 1 is a simple global memory implementation where all data is placed and accessed in global memory. Implementation 2 utilizes constant memory to cache the data for much faster access. Implementation 3 adds in the functionality of mathematical device functions.

Table 6.3 – Execution Times for Single-GPU K-Means Algorithm

Input Data Size	Execution Time (s)		
	Implementation 1	Implementation 2	Implementation 3
0.40%	46.008	40.657	39.761
1%	71.614	65.014	62.251
2%	126.238	115.725	109.696
4%	161.286	140.587	136.98

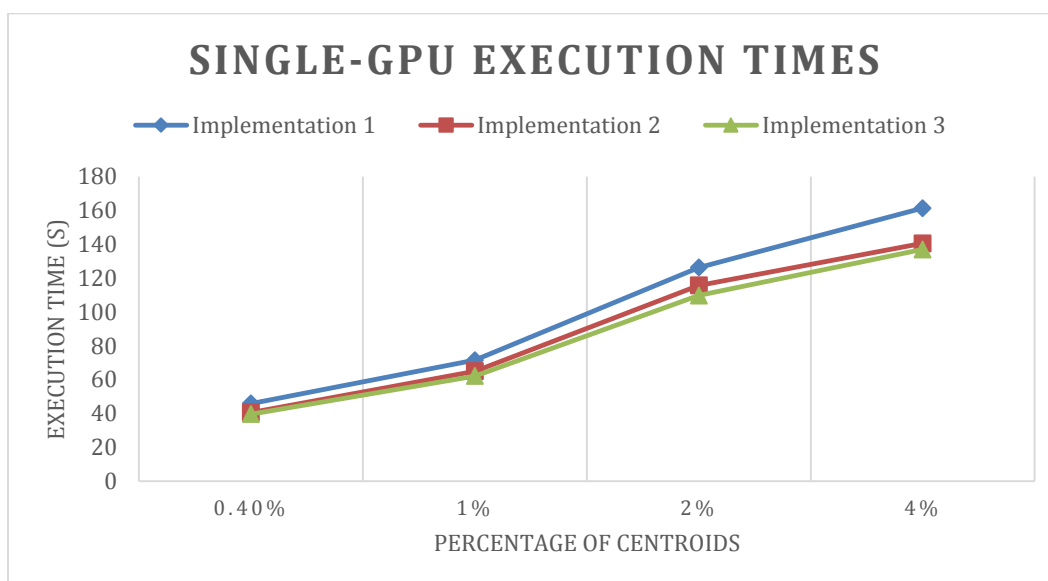


Figure 6.3 – Execution Times for Single-GPU K-Means Algorithm

Figure 6.3 sheds light into another important feature of the GPGPU and why it is very useful. As the amount of computation increases (the higher percentages means there are more centroids and therefore more computation each iteration), the GPGPU performs better. With little computation the GPGPU is “starved” and the communication overhead is the bulk of the computation time. However when more computation is introduced into the algorithm, the GPGPU parallelism is better exploited. Therefore, since there are many more centroids with 4% than there are with 0.4%, the execution time curve begins to trend down for a percentage of 4% due to the abundance of computation.

## 6.1.4 Multi-GPU Implementation

Analysis of the results from the multi-core and single-GPU sections above reveals potential advantage when exploiting the parallelism present in not only the multi-core systems but also accelerators such as GPGPUs. Therefore, the last study conducted on the K-Means algorithm is a combination of the two; incorporating multi-core and GPGPUs to create a set of CPU/GPGPU pairs for computational purposes. The CPU/GPGPU pairs will allow for distributed workloads as well as computation offload with the GPGPU. Table 6.4 shows the execution times for the multi-GPU implementation of the K-Means algorithm while Figure 6.4 shows a graphical representation of the execution times. Because of its computational intensity, this sections results are based on a data size of 750,000 (as in previous sections), however only the 4% configuration for the centroids will be used.

Table 6.4 – Execution Times for Multi-GPU K-Means Algorithm

Number of Nodes	Execution Time (s)		
	Implementation 1	Implementation 2	Implementation 3
1	161.286	140.587	136.98
2	78.341	71.67	68.723
4	40.287	45.182	43.11
8	24.924	20.551	19.759
16	9.447	8.963	8.392
32	6.832	6.507	6.378

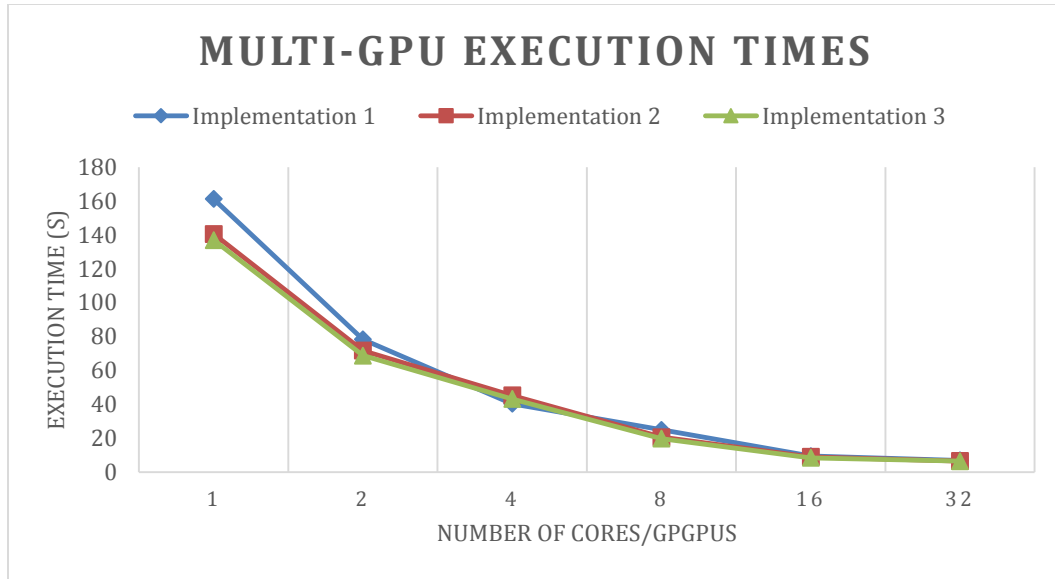


Figure 6.4 – Execution Times for Multi-GPU K-Means Algorithm

As shown in Figure 6.4, there was not a significant different between the three optimization techniques for multi-GPU. However, Table 6.4 shows that there is a difference between them, although slight. Even though the difference is slight, utilizing constant memory and built-in math functions has provided the best performance improvement compared to the multi-core implementation. Figure 6.5 shows the same curves as Figure 6.4 with the addition of the multi-core execution time curve. This gives perspective on exactly how much performance benefit is gained by utilizing multi-GPU systems for this application. It can easily be seen that there is a significant different in the execution time of the multi-core implementation versus all three of the optimized multi-GPU implementations. Section 6.1.5 will discuss exact speedup values for the K-Means algorithm.

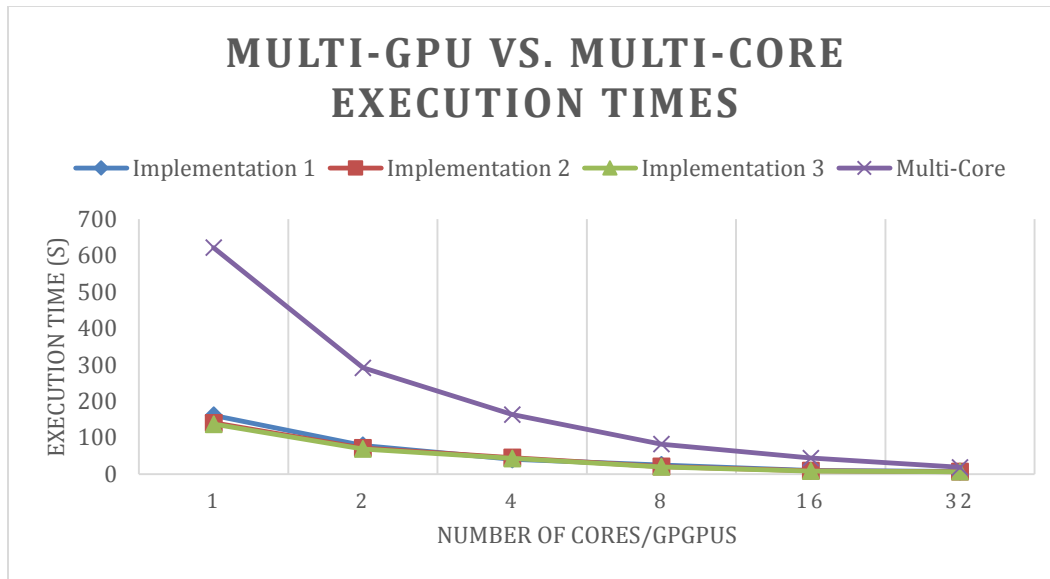


Figure 6.5 – Execution Times for Multi-GPU Comparison with Multi-Core K-Means

### 6.1.5 Speedup

The main reason most programmers use multi-core systems, GPUs, and multi-GPU systems is to accelerate their applications. Speedup is calculated by taking the quotient of the single-core execution time with each implementation execution time. The speedup value gives an idea of how well the accelerated and optimized implementations are performing compared to the original single-core implementation. Table 6.5 shows the speedup values compared to the single-core implementation and Figure 6.6 depicts them and illustrates the usefulness of utilizing a multi-GPU implementation, which in this case yields approximately 97x speedup. With only multi-core implementation, the K-Means algorithm achieved a 34x speedup while multi-GPU implementations boast around a 90x speedup. The single-GPU implementations seem to not do as well as other implementations and this is simply because they are performing all of the computation and requires too much data transfer overhead. For multi-core and multi-GPU the computations are spread out over 32 cores/32 GPGPUs. The speedup for

the single-GPU implementation is approximately 4.5x while the multi-GPU implementation is about 97x faster than the single-core implementation.

Table 6.5 – Speedup over Single-Core for all K-Means Implementations

Implementation	Speedup over Single-Core
Single-Core	1
Multi-Core (32-cores)	34.274
Single-GPU (Implementation 1)	3.855
Single-GPU (Implementation 2)	4.422
Single-GPU (Implementation 3)	4.539
Multi-GPU (Implementation 1)	91.002
Multi-GPU (Implementation 2)	95.547
Multi-GPU (Implementation 3)	97.479

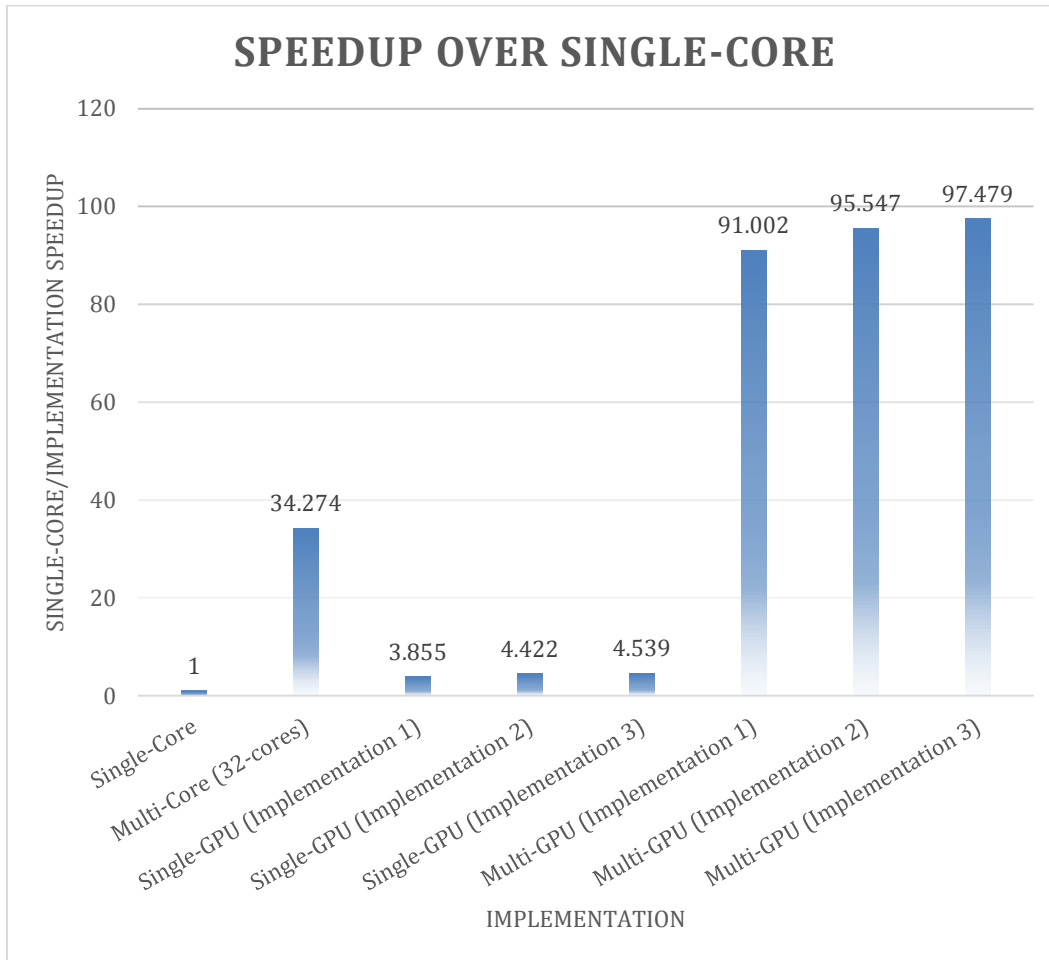


Figure 6.6 – Speedup over Single-Core for all K-Means Implementations

## 6.2 Spiking Neural Network (SNN)

### 6.2.1 Single-Core Implementation

As mentioned in Section 5.2.2, this application was converted from MATLAB into a C/C++ implementation, which was then used to create a parallelized multi-core implementation and GPGPU implementation. Because the parameters were not significantly changed in this particular algorithm, there is not much to compare to other than the speedup of using multi-core systems and accelerators to achieve better performance. The original execution time of the algorithm is shown in Table 6.6, which contains the total time for the application as well as the two iteration timings. The two iterations of the algorithm are performed as two different levels in the neural network, but because the parameters are not changed throughout this experiment, they are not mentioned (except in the description located in Table 5.3 in Section 5.2.2).

Table 6.6 – Execution Times for Single-Core Izhikevich SNN

Algorithm Section	Execution Time (s)
Total	18927
Iteration 1	6251.75
Iteration 2	6263.5

### 6.2.2 Multi-Core Implementation

After implementation of the single-core SNN algorithm, the next step was to parallelize the application to execute it across multiple cores simultaneously while splitting the computations evenly across the cores. There is a very computationally intensive part of this application and then a few other parts that are more communication intensive. For this reason,

only the computationally intensive segments were parallelized in an attempt to minimize the communication overhead between cores and nodes. The SNN algorithm described in Section 3.2.1 makes two calls to this “computationally intensive” section of code (i.e. two levels of neurons). For this reason, the important execution times that are recorded are the total execution time, and the time of each one of the iterations of the SNN algorithm computing neuron firing. Table 6.7 shows the values obtained when executing the code on 1-800 cores. Because of limited number of GPGPUs on the system and the inability to obtain a large number of nodes with GPGPUs, the only results that will be utilized from Table 6.7 are the results that come from node configurations up to 32 cores. Some of the results for larger configurations of nodes are not used but are added here for interesting observations that will be discussed later. Figure 6.7 shows the information in Table 6.7 and allows for a graphical explanation of the multi-core implementation.

Table 6.7 – Execution Times for Multi-Core Izhikevich SNN

Number of Nodes	Execution Time		
	Total	Iteration 1	Iteration 2
1	18926.5	6251.75	6263.5
2	12513.75	3135.25	3077.5
4	10061.25	1657.5	1570
8	8267	845.75	808.75
16	6519	371	519.25
32	5594.75	170	186.75
64	5262.5	94.25	91.25
80	4763.5	79.75	95
100	6603.5	79.25	82.25
128	5152.25	65	84
200	4872.5	756.25	48
256	3732.25	428.25	39.75
400	4261.5	930	25.25
800	4021	515.5	11.25

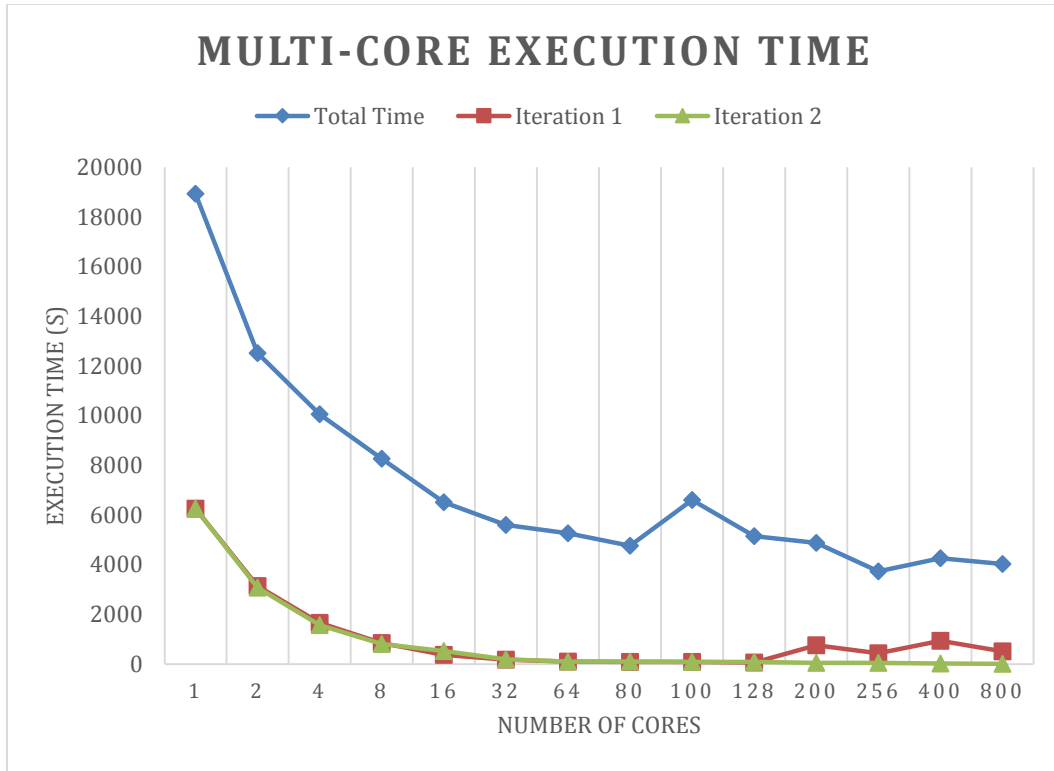


Figure 6.7 – Execution Times for Multi-Core Izhikevich SNN

Figure 6.7 shows that as the number of processors participating in the computation increases, the overall execution time of the algorithm as a whole as well as the iterations execution times decrease. This behavior is expected due to the inherent nature of the parallelism present in a multi-core system. For a comparison, in the single node implementation the total execution time for the application was approximately 18926 seconds (~315 minutes) compared to the 32-core application execution time of approximately 5595 seconds (~93 minutes). This same decrease in execution time can also be seen in the iteration timings (Iteration 1: ~6251 seconds vs. ~170 seconds, Iteration 2: ~6363 seconds vs. 187 seconds). The speedup values that incorporate these execution times will be described and shown in Section 6.2.5.

It is very interesting what occurs when a node configuration of larger than 32-cores is allocated. For the most part, the algorithm behaves the same, apart from some small variations. Figure 6.8 shows only the two iteration curves starting with 32-cores going to 800-cores for an easier view of what is occurring. For all core configurations it seems that iteration 2 steadily decreases, as it is expected to do. However there are some data points that do not agree with this observation for iteration 1. Some of these problems occur because of outliers resulting from a heavily loaded system. For example, it is difficult to allocate 400 nodes at a time without getting queued in the middle of the job for utilizing so many resources. Therefore, some of the error occurs when the job is preempted by other tasks, causing much longer run times. Also, it occurs for some values (mostly for values that are not powers of 2 exactly), that the execution times are much longer because of the configuration. The way the application is set up, it very easily divides the computation between cores when the total number is a power of 2. However, if the total number of cores is not a power of 2, there is a symmetry problem with the computation where some cores are performing much more computation than others leading to slight timing problems, which can be seen in Figure 6.8. Some of the discrepancies in Figure 6.7 and 6.8 include extraneous execution times as well as configuration problems. For example, in Figure 6.7, there is a large jump in the execution time when 100 cores are used for the computation. When 100 cores were allocated on the cluster, there were preemption problems that occurred during the execution causing much longer execution times. Also, for the larger number of cores that are not powers of 2 (i.e. 200, 400), the execution times are much larger due to the design of the algorithm being more balanced with a power of 2 number of nodes.

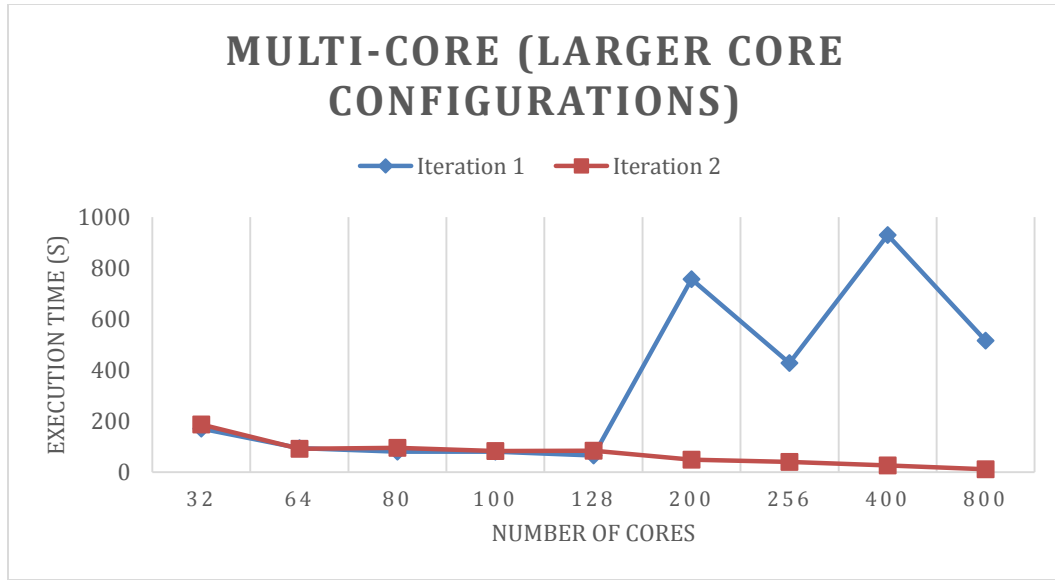


Figure 6.8 – Execution Time for Multi-Core (Large # of Cores) Izhikevich SNN

### 6.2.3 Single-GPU Implementation

As mentioned in Section 6.2.1, there is only a single implementation of the SNN algorithm with a single set of parameters that are used. Unlike the other algorithms, this implementation will only utilize this one set of parameters while incorporating them into 3 different implementations. Table 6.8 shows the total execution time for all 3 implementations described in Section 5.3, 1. Global memory, 2. Constant memory, 3. GPGPU mathematical functions, along with the time for each iteration in the algorithm. Figure 6.9 visualizes these execution times so that they may be easily compared with the single-core implementation.

Table 6.8 – Execution Time for Single-GPU Izhikevich SNN

Algorithm Section	Execution Time (s)		
	Implementation 1	Implementation 2	Implementation 3
Total	5239	3878	3761
Iteration 1	1831	1291	1241
Iteration 2	1848	1304	1238

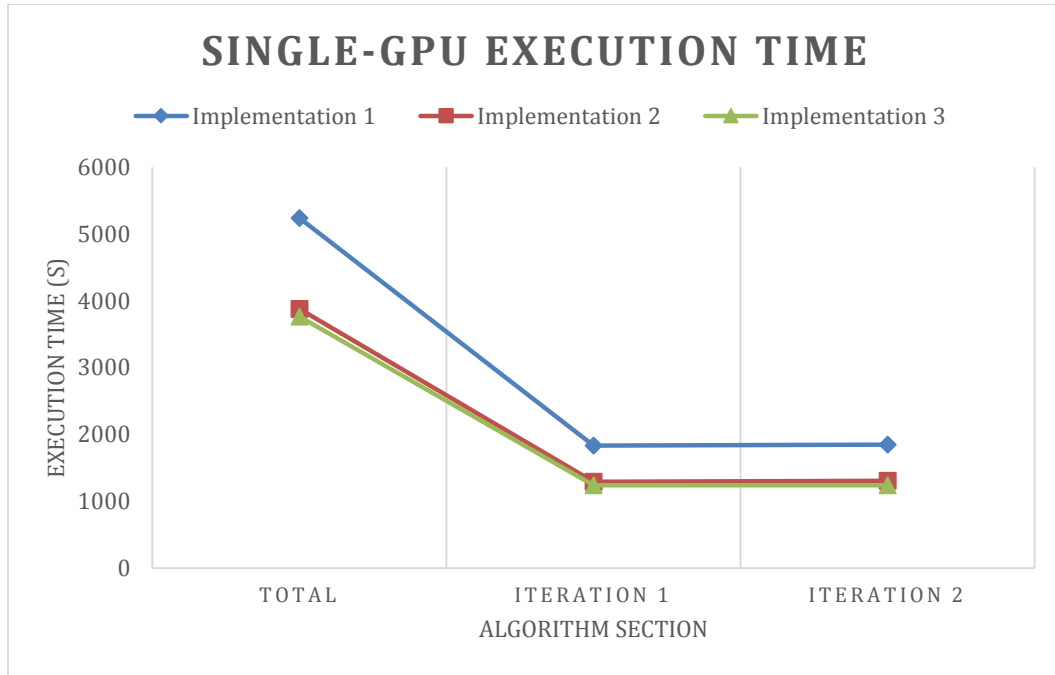


Figure 6.9 – Execution Times for Single-GPU Izhikevich SNN

From Figure 6.9, it can be seen that by optimizing first by the use of constant memory and then by utilizing built-in GPGPU math functions, the execution time for both iterations and the total execution time have been decreased. Figure 6.10 shows the comparison between the three GPGPU implementations and the single-core implementation. It can be seen that simply by utilizing the accelerator, the application has gained a large percentage of performance compared to the single-core implementation. A description of the entire set of speedup values is included in Section 6.2.5.

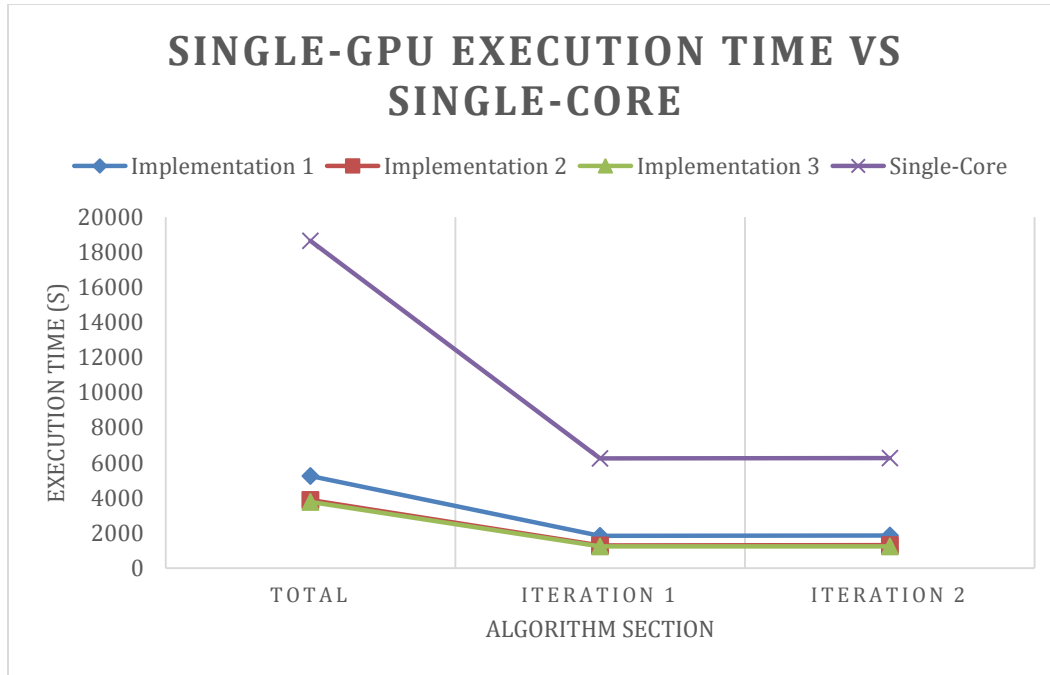


Figure 6.10 – Execution Time for Single-GPU vs. Single-Core Izhikevich SNN

## 6.2.4 Multi-GPU Implementation

As both the multi-core and single-GPU implementations both improved the performance of the algorithm, the logical next step is to utilize them both simultaneously. Therefore, the last set of implementations includes utilizing a CPU/GPGPU pair to perform the computation as in the single-GPU implementation, however, the computation will be spread across many pairs to further enhance the performance. Table 6.9 shows the total execution time of the each optimization technique using varying numbers of CPU/GPGPU pairs, Table 6.10 shows the execution time for Iteration 1, and Table 6.11 shows the execution time for Iteration 2. Figures 6.11 and 6.12 illustrate the total execution time as well as show a comparison between it and the multi-core implementation. Figures 6.13 and 6.14 show the execution time of Iteration 1 and compare it to the multi-core implementation. Figure 6.15 and 6.16 show the execution time of Iteration 2 and compare it to the multi-core

implementation. It can be seen that in all 3 of the implementations for each section of the Izhikevich SNN, each successive implementation performs better than the previous, while all of them perform significantly better than the multi-core implementation without GPGPUs. Explicit speedup and performance values are shown and described in Section 6.2.5.

Table 6.9 – Total Execution Times for Multi-GPU Izhikevich SNN

Number of Nodes	Execution Time (s)		
	Total Time		
	Implementation 1	Implementation 2	Implementation 3
1	5239	3878	3761
2	2772	2142	2076
4	1724	1346	1349
8	1316	976	924
16	1061	781	719
32	671	574	561

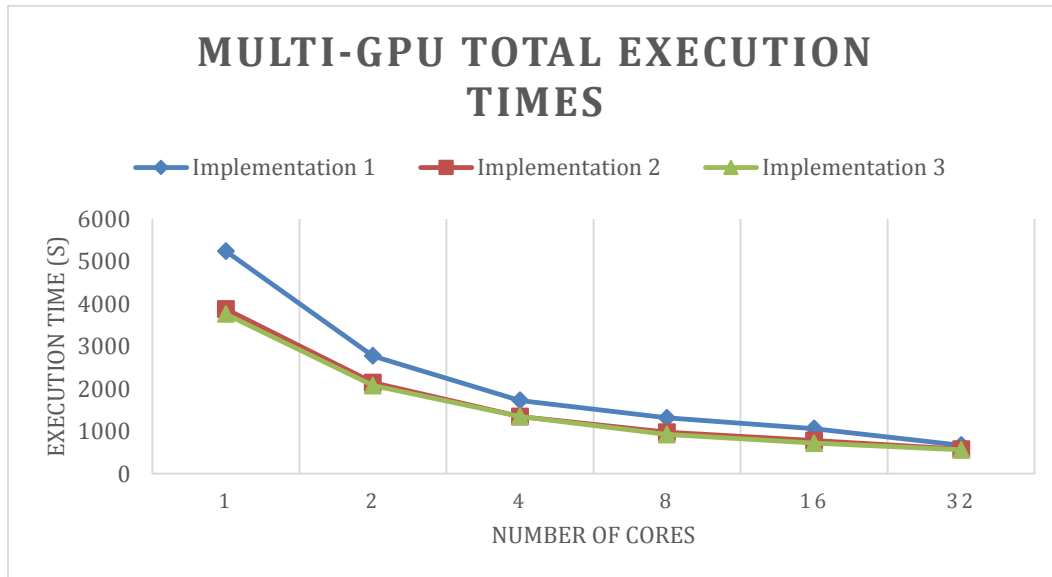


Figure 6.11 – Total Execution Times for Multi-GPU Izhikevich SNN

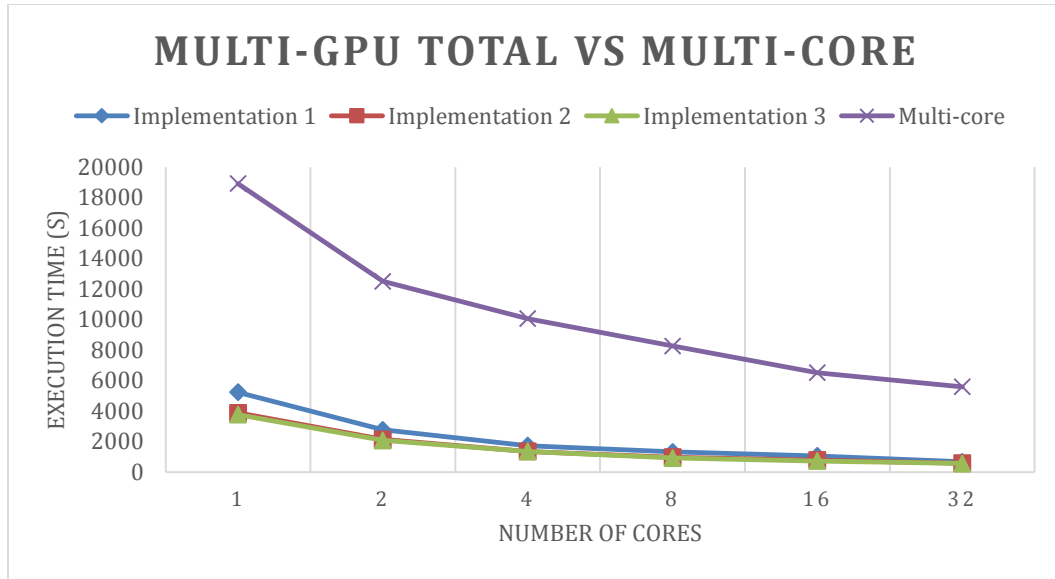


Figure 6.12 – Total Execution Times Comparison to Multi-Core Izhikevich SNN

Table 6.10 – Iteration 1 Execution Times for Multi-GPU Izhikevich SNN

Number of Nodes	Execution Time (s)		
	Iteration 1		
	Implementation 1	Implementation 2	Implementation 3
1	1831	1291	1241
2	910	648	650
4	458	362	329
8	221	172	155
16	119	97	86
32	57	46	41

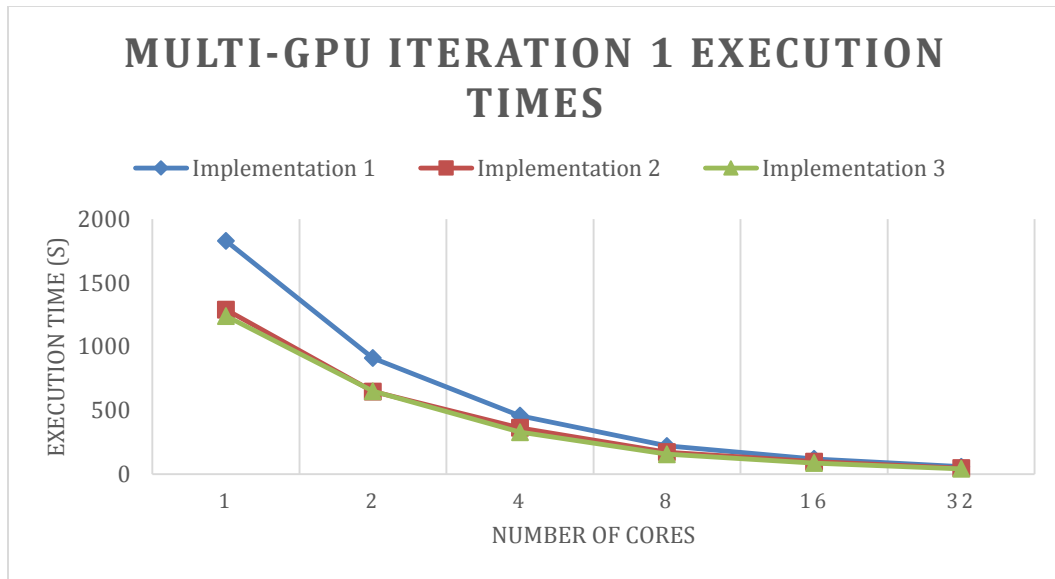


Figure 6.13 – Iteration 1 Execution Times for Multi-GPU Izhikevich SNN

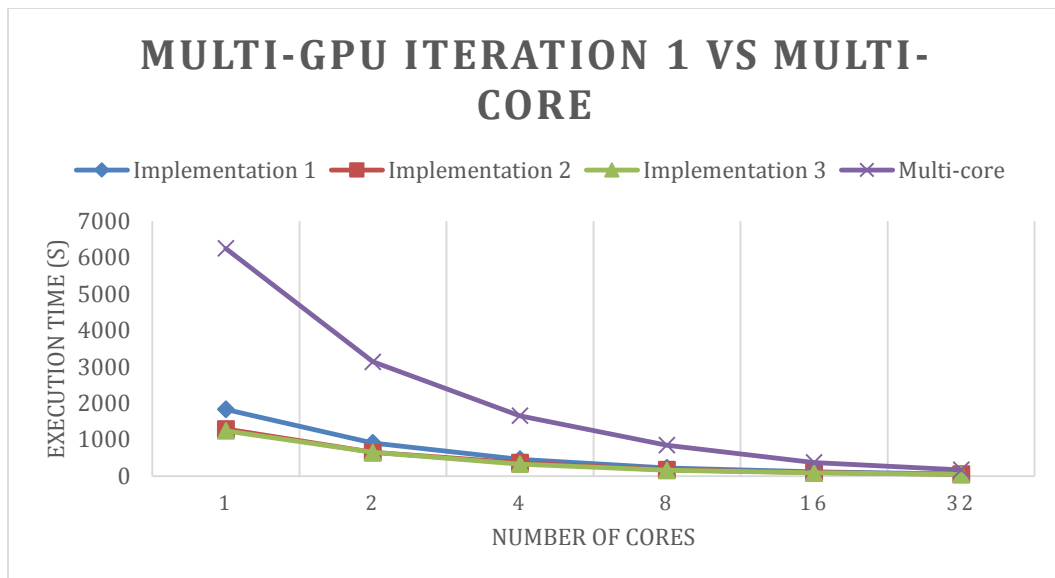


Figure 6.14 – Iteration 1 Execution Times Comparison to Multi-Core Izhikevich SNN

Table 6.11 – Iteration 2 Execution Times for Multi-GPU Izhikevich SNN

Number of Nodes	Execution Time (s)		
	Iteration 2		
	Implementation 1	Implementation 2	Implementation 3
1	1848	1304	1238
2	951	651	658
4	482	371	365
8	219	164	169
16	127	93	93
32	59	46	39

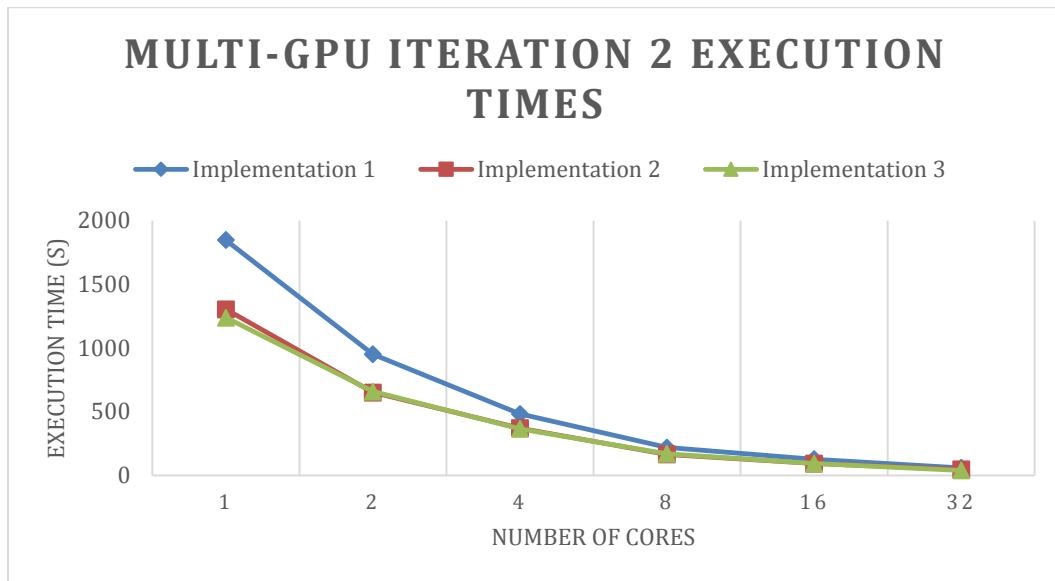


Figure 6.15 – Iteration 2 Execution Times for Multi-GPU Izhikevich SNN

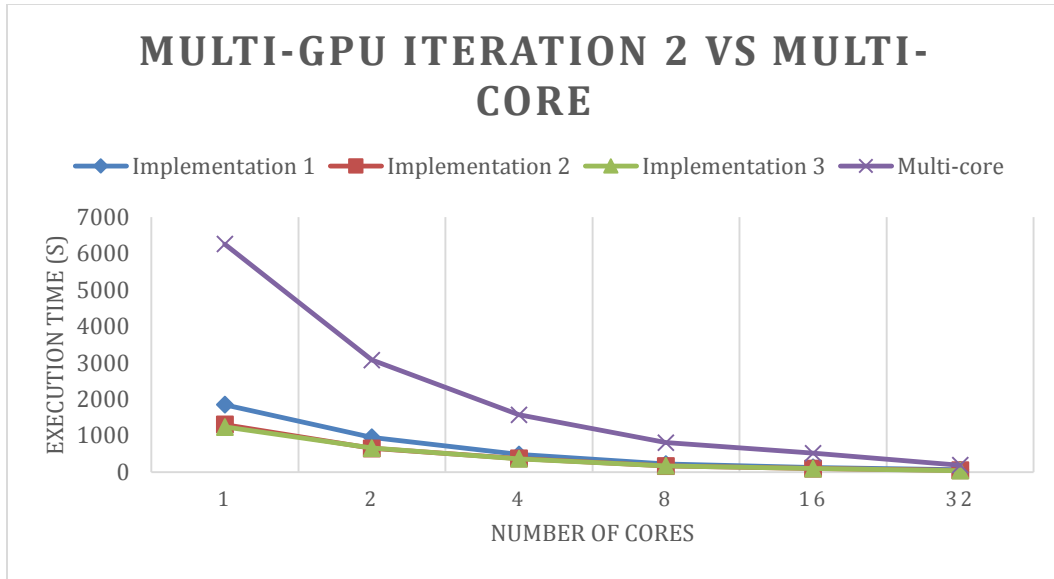


Figure 6.16 – Iteration 2 Execution Times Comparison to Multi-Core Izhikevich SNN

## 6.2.5 Speedup

The performance of algorithms is important in the domain of HPC however, one of the main criteria is speedup of an application when utilizing different optimizations and architectures. Table 6.12 shows the speedup values for the Izhikevich SNN algorithm given by the single-core execution time over the given implementation execution time. Tables 6.13 and 6.14 show the speedup of the individual iterations (1 and 2). Figure 6.17 illustrates the speedup values in a graphical manner and shows that a maximum speedup of 33x is achieved when utilizing the multi-GPU implementation. Figure 6.18 illustrates the speedup values for Iteration 1 in the SNN algorithm, which yields a 152x speedup while Figure 6.19 illustrates speedup values for Iteration 2 in the SNN algorithm, which yields a 160x speedup. Note that all multi-GPU implementation results are evaluated utilizing 32 CPU/GPGPU pairs to achieve highest speedups.

Table 6.12 – Speedup of Total Execution Time for Izhikevich SNN

Implementation	Speedup Over Single-Core
Single-Core	1
Multi-Core (32-cores)	3.3338
Single-GPU (Implementation 1)	3.5602
Single-GPU (Implementation 2)	4.8097
Single-GPU (Implementation 3)	4.9593
Multi-GPU (Implementation 1)	27.7973
Multi-GPU (Implementation 2)	32.4948
Multi-GPU (Implementation 3)	33.2478

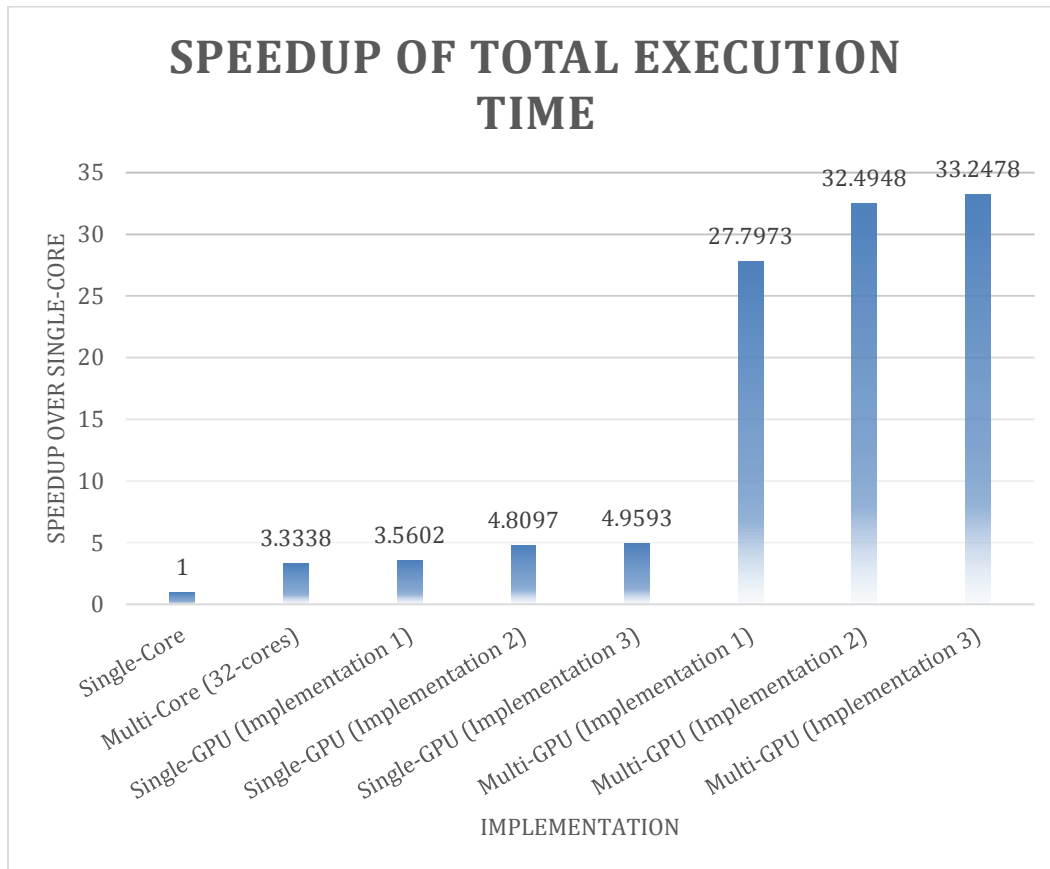


Figure 6.17 – Speedup of Total Execution Time vs. Single-Core Izhikevich SNN

Table 6.13 – Speedup of Iteration 1 Execution Times for Izhikevich SNN

Implementation	Speedup Over Single-Core
Single-Core	1
Multi-Core (32-cores)	36.7882
Single-GPU (Implementation 1)	3.4156
Single-GPU (Implementation 2)	4.8443
Single-GPU (Implementation 3)	5.0395
Multi-GPU (Implementation 1)	109.7193
Multi-GPU (Implementation 2)	135.9565
Multi-GPU (Implementation 3)	152.5366

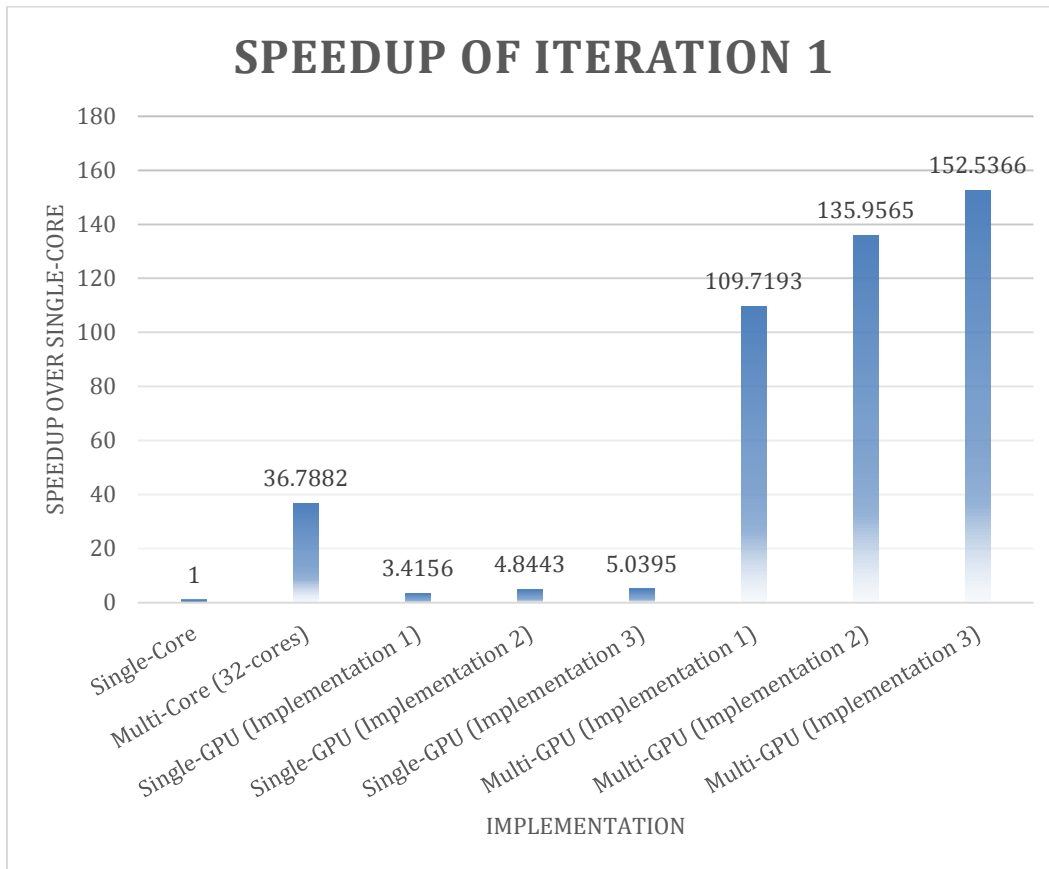


Figure 6.18 – Speedup of Iteration 1 Execution Times vs. Single-Core Izhikevich SNN

Table 6.14 – Speedup of Iteration 2 Execution Times for Izhikevich SNN

Implementation	Speedup Over Single-Core
Single-Core	1
Multi-Core (32-cores)	33.5529
Single-GPU (Implementation 1)	3.3907
Single-GPU (Implementation 2)	4.8052
Single-GPU (Implementation 3)	5.0614
Multi-GPU (Implementation 1)	106.2034
Multi-GPU (Implementation 2)	136.2174
Multi-GPU (Implementation 3)	160.6667

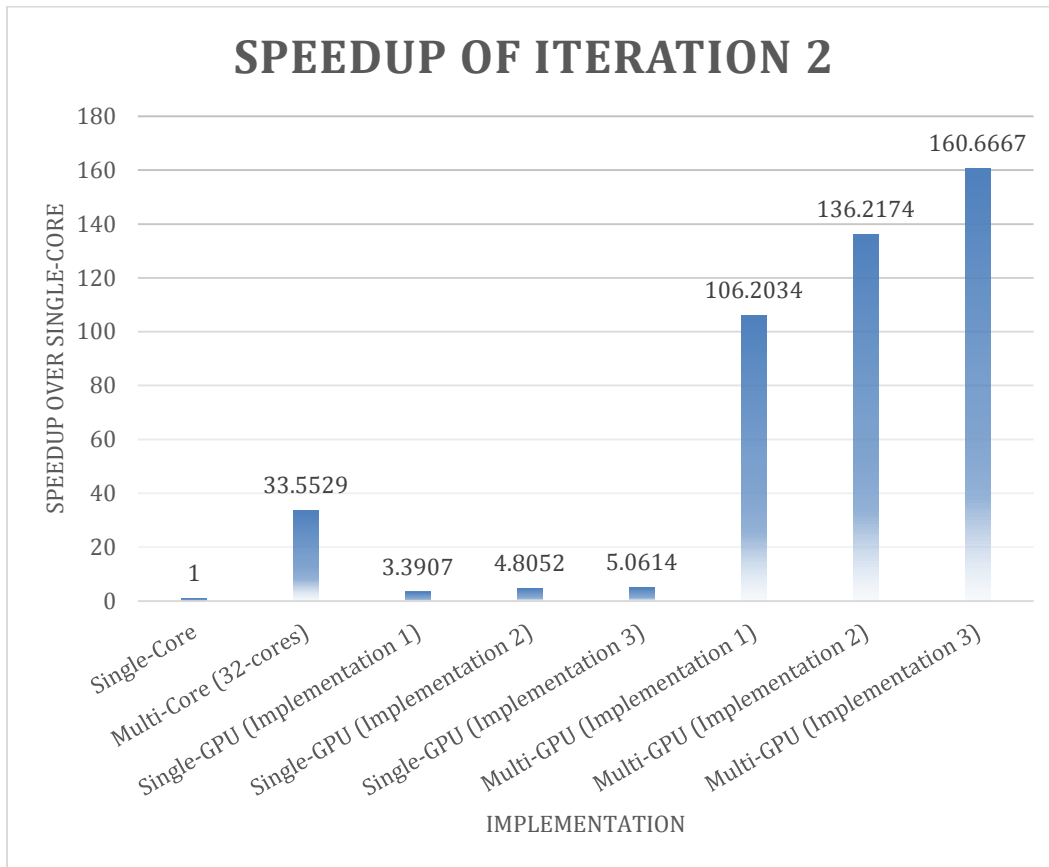


Figure 6.19 – Speedup of Iteration 2 Execution Times vs. Single-Core Izhikevich SNN

## 6.3 Longest Common Subsequences (LCS)

### 6.3.1 Single-Core Implementation

As mentioned in Section 5.2.3 and Table 5.4, there is a single large data set used for the LCS algorithm where smaller subsets of the data set are used to achieve execution time values. For all sequences generated, the maximum length for each sequence ranges from 50 to 500 while the number of sequences that are being tested range from 5 to 100. Table 6.15 shows the execution times of the single-core version of the algorithm and Figure 6.20 illustrates the execution times. Comparisons are made to the single-core results for the multi-core and single-GPU implementations, while the multi-GPU implementation is compared to the multi-core implementation for fairness.

Table 6.15 – Execution of Single-Core LCS Algorithm

Number of Sequences	Execution Time (s)			
	50	100	250	500
5	4.2345	5.2763	5.9817	6.3173
10	4.6782	5.9273	7.1973	7.9437
20	5.1934	6.3847	7.6321	10.894
25	5.4293	6.8119	8.1113	13.1029
50	5.9283	8.1372	15.2837	25.7288
75	6.3123	9.7262	21.7162	38.1152
100	6.8374	10.7283	28.8162	49.1836

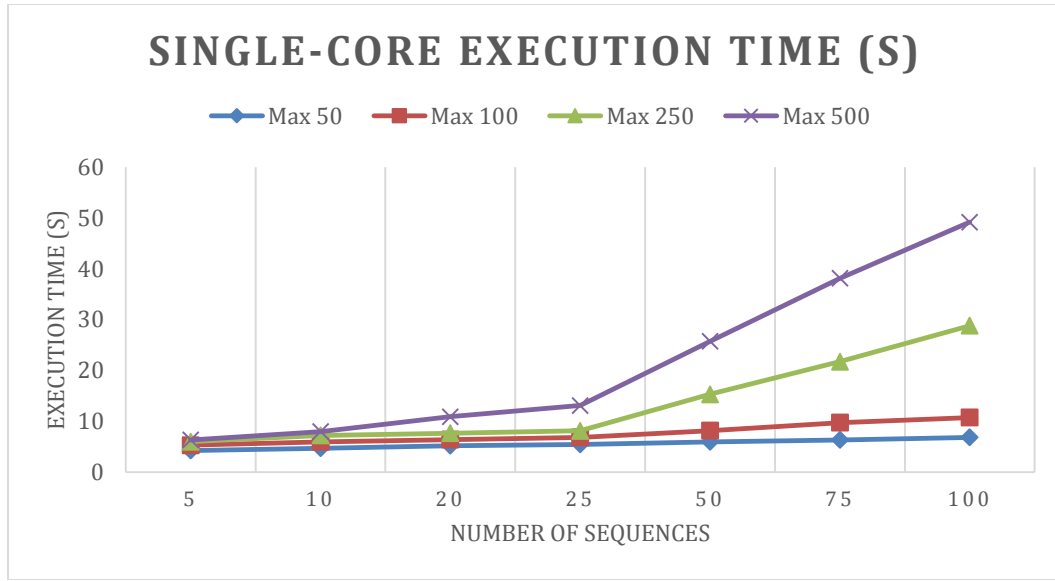


Figure 6.20 – Execution Times of Single-Core LCS Algorithm

Both Table 6.15 and Figure 6.20 show that the execution time for the single-core algorithm is linear as expected. As the number of sequence comparisons increase, the execution time will ultimately increase, and for a set of larger sequences the execution time will be greater. Sections 6.3.2, 6.3.3, and 6.3.4 capture the results of the LCS algorithm utilizing accelerators and multi-cores systems to achieve better performance.

## 6.3.2 Multi-Core Implementation

The LCS algorithm is an inherently parallel algorithm, meaning that each comparison between two subsequences can complete without any knowledge of the other comparison outcomes. Therefore, the algorithm is a very good candidate for parallelization. Two techniques are used for parallelization: one utilizing multi-core CPUs while the other offloads the computations to an accelerator. This section covers the multi-core CPU implementation results while the following two sections cover the accelerator and multi-accelerator implementations.

In the multi-core implementation the computation is simply spread across all of the processors available on a given node of the cluster. In this case, a range of 1-32 processor cores (must be powers of 2 for best performance) is tested and the results are shown in Table 6.16 and Figure 6.21. The 1-core configuration are the same results shown in the previous sections. To test the multi-core implementation, only the maximum sequence length of 500 is used to ensure the most computation possible as well as searching through the maximum number of sequences, 100. A larger number of cores could possibly be tested, however, to compare with the multi-GPU implementation, this relatively small number is used due to lack of hardware in the current cluster. However it can be seen in both Table 6.16 and Figure 6.21, that when increasing the number of cores past a single node, the execution times are low enough that the transfer times are more difficult to hide, leading to slightly larger execution times. With more computation being done on each node, the communication section of the algorithm can be negated therefore leading to better performance with each node. For this reason, when the node configuration reaches 16 with the given data sizes, a threshold is reached where the computation was sufficient to hide the communication. Larger configurations prove to cause communication overheads that in turn lead to higher execution times. Due to the fact that larger data sets were not generated for this experiment, for the multi-GPU implementation, only node configurations up to 16 were tested because the GPU would only speed up the computation being done on the node making the communication overhead more noticeable leading to much worse performance.

Table 6.16 – Execution Times for Multi-core LCS Algorithm

Number of Cores	Execution Time (s)
1	49.1836
2	26.5827
4	16.9123
8	15.6392
16	10.8321
32	11.0052

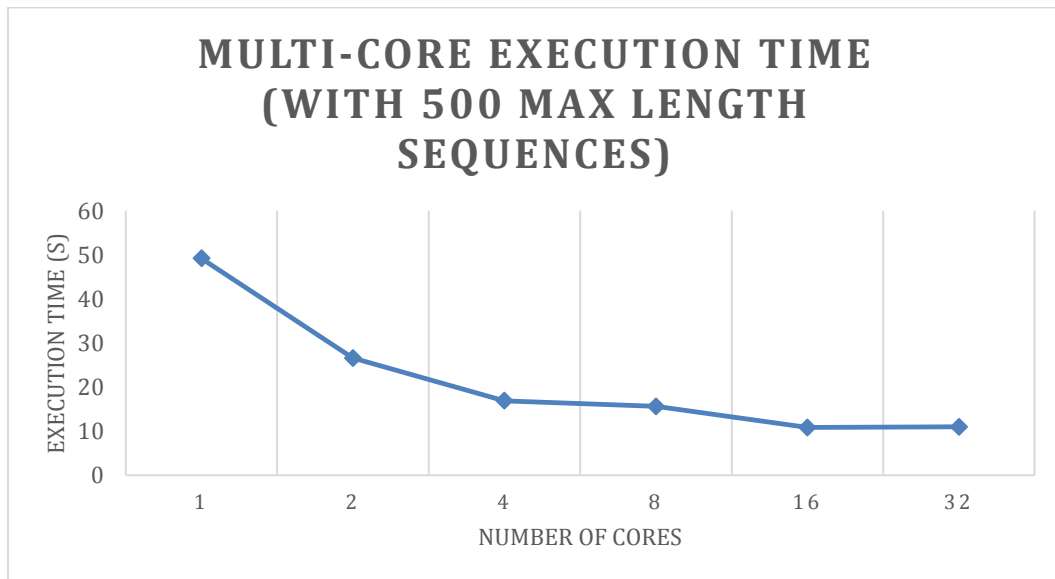


Figure 6.21 – Execution Times for Multi-Core LCS Algorithm

Figure 6.21 illustrates that in general, as the number of cores increase, the execution time decreases, which is expected in a multi-core environment. However, during the testing, the cluster was under a heavy load, which could explain why the execution times of the 4-core implementation and the 8-core implementation are similar. Most of the time when asking for 4-cores, the entire node is not utilized and the unused portion of the node can be allocated to a different job leading to a degradation in execution time because the CPU is actually working on more than one task. But in general, the trend of increased cores leading to decreased execution time has held for this application. Section 6.3.5 will compare this implementation

to the single-node implementation to illustrate how much of a performance benefit is gained by simply using a multi-core system for the algorithm.

### **6.3.3 Single-GPU Implementation**

Although, the multi-core implementation of the LCS algorithm performed well and showed achievements in execution time as more cores were used for computation, there are other concepts that can be utilized to further develop the application performance; for instance, the use of accelerators. With the use of accelerators (i.e. GPGPUs) the parallel computation can be performed much quicker due to the high degree of parallelism in these devices with hundreds of cores. As in the multi-core implementation, the single-GPGPU implementation performs the same tasks on subsets of the data set. Contrastingly however, the GPGPU performs the tasks using threads in parallel while the multi-core implementation must use serialization to perform tasks over a data set. For instance, on an 8-core machine, 8-sequences can be compared simultaneously. However, on a GPGPU device, hundreds of simultaneous comparisons can be made leading to probably performance improvements.

Table 6.17 shows the execution times of each of the different implementations on a single-GPU. Figure 6.22 shows the same execution times compared to the single-core execution time. This allows for comparison between a single-core and a single-GPU. For the comparison shown in Figure 6.22, 100 sequences are used while varying the maximum length of the sequence. The maximum number of sequences is used in order to maximize the runtime giving the GPGPU a chance to exploit its performance capabilities over a typical CPU.

Table 6.17 – Execution Times for Single-GPU LCS Algorithm

Max Length of Sequences	Execution Time (s)		
	Implementation 1	Implementation 2	Implementation 3
50	5.9273	5.1498	4.9981
100	9.2177	7.2514	6.9812
250	17.6548	13.8642	13.2415
500	34.8861	24.4469	23.0133

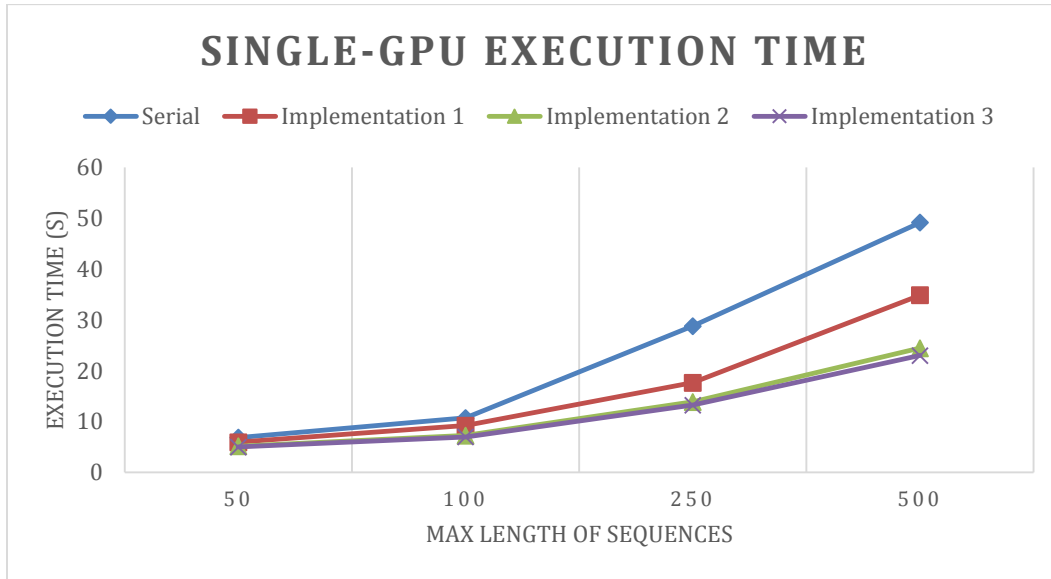


Figure 6.22 – Execution Times for Single-GPU LCS Algorithm

It can be seen by Table 6.17 that a maximum sequence length of 50 does not allow the GPGPU to outperform the single-core version by much because the communication time between host and device provides substantial overhead compared to the required computation. As the amount of computations needed increases (i.e. the input data set increases in size) the GPGPU begins to pull away from the single-core implementation in execution time. Each successive implementation on the GPGPU also provides a slight performance benefit over not only the single-core but also previous implementations of the GPGPU. For instance, with a maximum length for each subsequence set to 500, the single-core execution time is approximately 50 seconds, while the first GPGPU implementation only

utilizing global memory (no other optimizations) completes in approximately 35 seconds. By simply utilizing the GPGPU global memory, a 1.5x speedup was achieved over the single-core implementation.

Implementation 2 incorporates constant memory, allowing the sequences to be accessed very quickly through the caching aspects of constant memory. The constant memory implementation allowed for execution times of about 25 seconds, which is about a 2x speedup over the single-core implementation as well as a slight speedup over the previous implementation. As a last optimization, GPGPU optimized math functions were used to further enhance the performance. Although there were not many occurrences of built-in mathematical functions, there were a few that were used hundreds of times for each sequence. Since there are many of these function calls throughout the algorithm, each clock cycle saved will reduce the overall execution time. Although there was not a significant difference in the last two implementations, there is still a slight performance improvement simply through the use of built-in GPGPU math functions.

### **6.3.4 Multi-GPU Implementation**

To further evaluate the LCS algorithm, a combination of the multi-core and single-GPU implementations is constructed leading to the multi-GPU implementation. This implementation allows for the computation to not only be spread across multiple nodes, but instead of utilizing the compute power of the CPU, the GPGPU may be used to parallelize the computation and receive the result faster. Table 6.18 shows the execution times of each implementation with different node configurations and Figure 6.23 illustrates the execution times as well as gives a comparison between the multi-core implementation. Figure 6.23 shows

that all implementations of the algorithms performed using multiple GPGPUs perform better than the multi-core implementation, however, Table 6.18 illustrates how much of a difference utilizing 16 CPU/GPGPU pairs rather than 1 can make. The results show that in implementation 1, when utilizing 16 CPU/GPGPU pairs instead of 1 shows improvements from 35 seconds to just over 5 seconds in execution time. For implementation 3, which utilizes constant memory and built-in mathematical functions, the improvement is slightly less but still improves from 23 seconds to approximately 4.5 seconds. These results prove that for the LCS algorithm, the multi-GPU configuration performs the best of the implementations, which is to be expected for algorithms that can utilize the computational power of the GPGPU.

Table 6.18 – Execution Times for Multi-GPU LCS Algorithm

Number of Cores/GPGPUs	Execution Time (s)		
	Implementation 1	Implementation 2	Implementation 3
1	34.8861	24.4469	23.0133
2	17.7162	13.6912	13.0198
4	11.1872	7.8392	8.3924
8	7.2934	6.5124	5.8712
16	5.3385	4.9819	4.4928

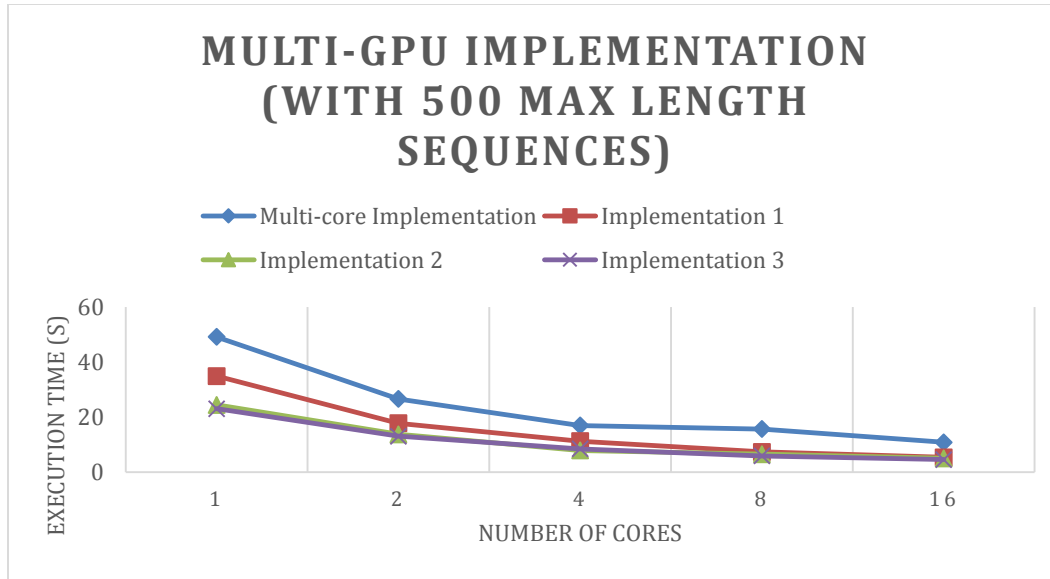


Figure 6.23 – Execution Times for Multi-GPU LCS Algorithm

### 6.3.5 Speedup

This section delves into the speedup of the application over the different implementations. Since some applications only use 500 as the maximum length of the subsequences, only the single-core values that illustrate this maximum length will be used. Table 6.19 shows the speedup values while Figure 6.24 illustrates them and shows that with a multi-GPU implementation, almost an 11x speedup can be achieved.

Table 6.19 – Speedup Values over Single-Core LCS Algorithm

Implementation	Speedup over Single-Core
Single-Core	1
Multi-Core (16-cores)	4.469123687
Single-GPU (Implementation 1)	1.409833716
Single-GPU (Implementation 2)	2.011854264
Single-GPU (Implementation 3)	2.137181543
Multi-GPU (Implementation 1)	9.212999906
Multi-GPU (Implementation 2)	9.872458299
Multi-GPU (Implementation 3)	10.94720442

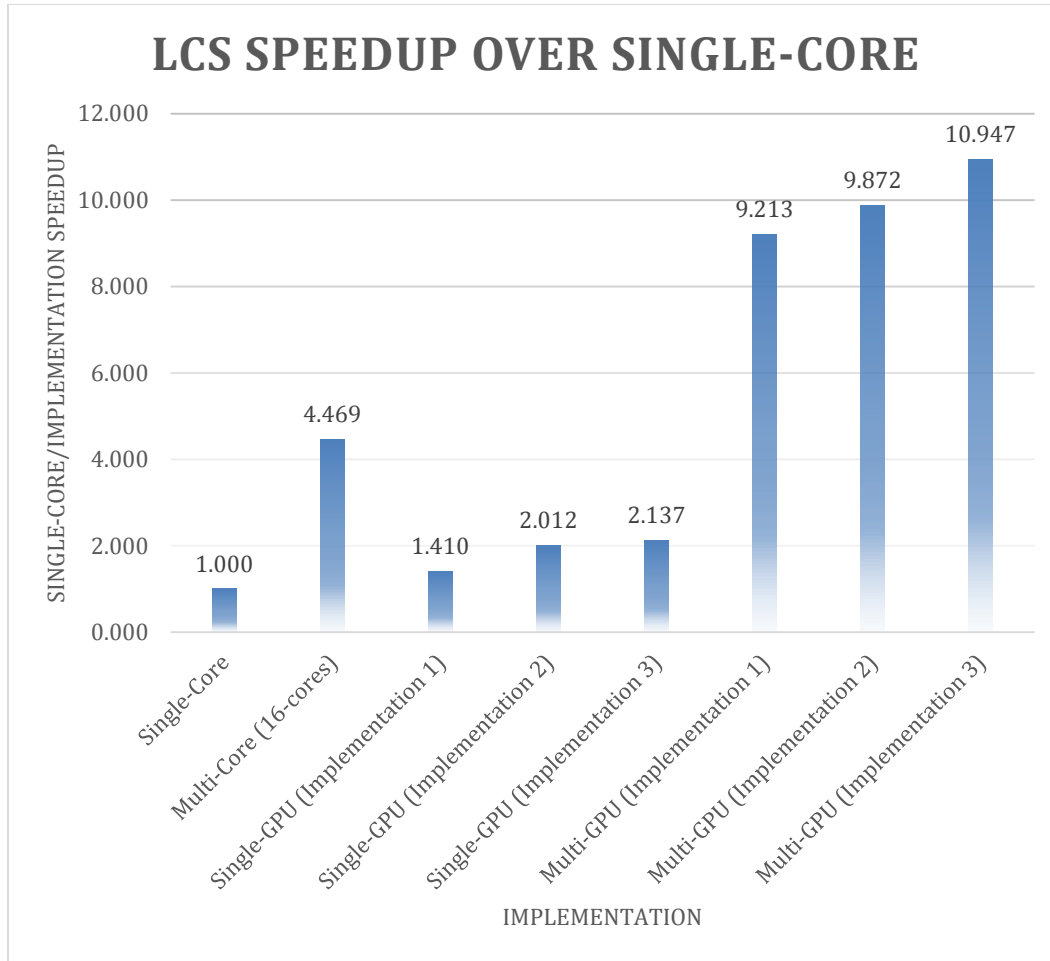


Table 6.24 – Speedup Values over Single-core for LCS Algorithm

## 6.4 Summary

In this chapter, the results were presented and analyzed for each of the three algorithms discussed in Chapter 3 using the optimization techniques and setup described in Chapter 5. It was observed that for all applications, the multi-GPU systems performed best when compared to other implementations. The multi-GPU configurations edged out the multi-core systems because of the extra computational power present in each GPGPU as well as beating out the single-GPU because larger data sets are able to be evaluated as well as similar

size data sets can be partitioned and evaluated faster than on a single GPGPU. Chapter 7 will present the conclusions that were obtained from this research study as well as future work.

# Chapter 7

## Conclusions and Future Work

### 7.1 Conclusions

In this thesis, successful implementations of the K-Means algorithm, the Izhikevich Spiking Neural Network (SNN), and the Longest Common Subsequences (LCS) Problem have been demonstrated. The K-Means algorithm consisted of data sets that contained a range of data points from 1,000 to 750,000 providing a large range of computational difference when comparing optimization techniques and implementation strategies. However, in the end, only the larger data sizes were compared because those results offer more interesting analysis. After implementing the K-Means algorithm using multi-core and GPGPUs, speedup values of  $\sim 90x$  were achieved utilizing 32-cores, each with a GPGPU present for the heavy computation. The Izhikevich SNN, algorithm consisted of a single data set that modeled a two-level neuron network. Since all of the parameters of the algorithm were kept constant, the only thing changing was the GPGPU implementations. Again the multi-GPU implementation performed best in this situation achieving speedup values of  $\sim 150x$ . The LCS problem was the last algorithm studied. 100 sequences were generated for testing with the maximum length of each sequence ranging from 50 to 500. After implementation, speedup

values of approximately 11x were achieved while utilizing 16 CPU/GPGPU pairs. It is significant to mention that GPGPUs perform better when there are higher computational loads put on them rather than smaller loads with massive communications. This is the reason why the larger data sets were used in an attempt to force the GPGPU to perform at its highest potential.

After reviewing the results for the multi-core implementation of the K-Means algorithm, it can be seen that the speedup is superlinear; speedup is greater than the number of cores. Superlinearity can result from a few factors but the most logical for this application is the caching that occurs when utilizing a large number of cores. As more hardware is added (i.e. more nodes are added for the computation), the problem is split among the cores and each additional core provides its own cache. Therefore, more data can be stored in the available cache (distributed across the cores) with more compute nodes leading to the ability to achieve superlinear speedup values. However, with a small number of cores, the speedup values remain sublinear because the cache space is not large enough to house the entire data set. Finally, for smaller data sets, the cost of communication vs. computation distribution negates the cache effect and the speedups remain sublinear. The K-Means application could be studied further to validate this assumption by examining the size of the cache on each node and viewing the cache utilization for each configuration.

Each GPGPU implementation was examined (using each algorithm) for its performance. The first implementation was a simple, global-memory utilization algorithm. This algorithm performed well when compared to single-core simply because it was able to parallelize the computation where the single-core was not. However, this was only the beginning of the optimizations. The second optimization that made tremendous performance

improvements on the applications performance was the use of constant memory. Constant memory is used for data that will be accessed very frequently but will not be changed throughout the duration of the kernel. For example, in the K-Means algorithm, the data points will not change, so they can all use the constant memory and utilize the caching that takes place for quicker memory retrieval. Similarly for the LCS algorithm, none of the sequences change throughout the entire algorithm duration; therefore, they can be placed in constant memory for fast access throughout the execution. The final optimization technique was an attempt to gain slightly more performance out of each of the applications. NVIDIA has built-in mathematical functions that have been optimized for use on the GPGPU. They can be written by hand or traditional math functions can be used, but the optimized functions tend to take significantly fewer clock cycles than other implementations. Therefore, even if it is a small improvement, throughout an application, it could make a big difference to eliminate a few clock cycles each time through a loop. The decrease in clock cycles achieved by utilizing the device optimized mathematical functions is not significantly apparent in this research, however, for other applications, utilizing the optimized functions could make a significant difference. For example, if a particular algorithm consists of a large amount of trigonometric or logarithmic operations, utilizing the device functions over traditional mathematical functions could prove to achieve a much larger benefit in performance.

The research conducted in this thesis has led to several valuable contributions as well as insights into application development, computation on accelerators, and effective utilization of accelerators. As discussed in Section 7.2 this research will be used extensively to validate existing performance modeling frameworks as well as develop improvements to the modeling approach in future work. Each algorithm described in this thesis involves varying amounts

and types of computations creating a diverse test bed for the framework. Many previous implementations of these algorithms have been developed in OpenCL or for single-GPU systems while in this research, the multi-GPU implementation was also developed. The paradigm of High Performance Computing (HPC) is moving rapidly toward large-scale systems, enabling these applications to execute on much larger data sets than those studied in this research.

This research has also improved the ability to develop algorithm applications for use with accelerators. Unlike sequential architectures, parallel programming architectures allow the programmer to perform computation much quicker, with the caveat that the data must be managed much more carefully to insure accurate results. This research allowed for a better understanding of the use of parallel computing architectures with computational models. Developing GPGPU applications also allowed for the use of different types of memories inside the accelerator with different latencies. Utilizing different device memory for subsets of the data illustrated the latency of each bank of memory and enabled a study of which type of data most efficiently takes advantage of each memory type.

## **7.2 Future Work**

The research completed in this thesis has presented the GPGPU and heterogeneous computing as a potential viable solution for problems that are very computationally intensive. In this research, scientific applications such as K-Means, Neural Networks, and Longest Common Subsequences demonstrate the effectiveness of utilizing heterogeneous systems. In this research however, the only programming model that is used for the GPGPU is CUDA due to the abundant supply of NVIDIA GPGPUs in the Palmetto Cluster. Also, there are

many more memory models and hardware optimizations that can be performed to further optimize the application execution.

#### *Other Implementations*

Along with CUDA there are other programming models, namely OpenCL and the like, that could be utilized and possibly compared to the implementations shown above. There are multiple reasons that OpenCL is a possibility for GPGPU implementations including a likeness in syntax to that of CUDA (with slight syntactical differences) as well as a comparison in performance to CUDA. Programming models such as CUDA and OpenCL could be compared for programming efficiency and performance on the same types of applications. Because CUDA is specialized for NVIDIA GPGPUs, CUDA may have an edge when the computation is completed on this family of GPGPUs, however when other families of GPGPUs are available (i.e. AMD/ATI), the only alternative would be OpenCL, which is a cross-platform tool. When using OpenCL, a comparison can also be made utilizing the same programming language for processor and accelerator rather than using C for the host processor and CUDA for the GPGPU kernels as it is presented in this research.

Along with the addition of the programming language comparison, other implementations of the CUDA applications could be explored to compare the performance. For instance, in this research, the shared memory of each block was not utilized and therefore some of the performance could have been degraded slightly from the maximum achievable performance. However, all applications are not necessarily perfect for use with the shared memory because of their construction and algorithms. Therefore a study would be required to determine if utilizing shared memory would benefit the applications performance.

In terms of scaling, the results show that the Izhikevich SNN scales better across multi-GPU systems when compared to the K-Means algorithm while the K-Means algorithm performs better when utilizing only the multi-core architecture. The Izhikevich SNN scales better on the multi-GPU system because of the amount of computation being done by each GPU while in the K-Means algorithm, as the number of nodes increases, the GPU performs less computation causing the communication to be the overbearing part of the algorithm. Performing research on the exact reasoning behind why these algorithms perform better on multi-GPU or multi-core could be a topic of future work, since it is important to understand these characteristics and parameters for performance modeling. Also, the LCS algorithm results show that as the number of CPU/GPU pairs increase, the performance of the multi-core and multi-GPU implementations is converging. Therefore, another possible area of future study is to examine where this overlap occurs to help with performance modeling and guiding scheduling decisions for applications that exhibit similar characteristics.

In this thesis, when designing the experiments for multi-GPU systems, a single CPU/GPU pair per node is used. However, in the Palmetto Cluster, each GPU-capable node is equipped with 2 GPUs. Utilization of both GPUs instead of one per node could yield performance improvements because it would allow for faster communication between the devices (i.e. intranode instead of internode communications). GPU to GPU communication is also an area that has not been studied substantially and therefore, it would be worth researching if communication directly through RDMA between devices would be faster and provide a larger performance improvement over communicating through the CPU as a middle-man.

Aside from HPC, Big Data is also a very substantial community in performance of applications. Big Data allows for distribution of large amounts of data for computation on

different nodes of a cluster or system. Since the data is already distributed, the inherent parallel computation can be completed as in multi-core processing. Hadoop is a programming model that allows for secure data distribution across a system with fault tolerances to avoid data loss. Hadoop allows the user to operate on the distributed data without combining it back on a single machine. The applications developed in this research would benefit from Big Data programming paradigms such as Hadoop on larger data sets. Accompanying the addition of larger data sets could be the inclusion of higher dimensional data sets. In many scientific algorithms, there are very high dimensional sets of data (compared to the 3 dimensional data used in this research). An interesting study would be to see how multi-core and multi-GPU systems perform on higher dimensional data sets with the same algorithms.

#### *Using Implementations for Verification*

In the field of HPC, it is very important to exploit the concurrency existing in heterogeneous systems, which could include clusters of GPGPUs or Many Integrated Cores (MICs) technology. These resources offer several petaflops of computing performance and therefore many developers choose to employ them for massively parallel applications. However, due to factors such as insufficient understanding of the architecture and inefficient load balancing, many times, these resources are not fully utilized and therefore, the application's optimal performance is never reached. To combat this problem, several performance models and strategies have been developed to efficiently tune these applications, but effective use of these models often requires substantial knowledge of the underlying computing architecture.

Many performance prediction models that have been developed were tightly coupled to the underlying architecture of the system [47], but as new architectures evolve and include

new features, the previous architecture specific models are rendered incomplete and irrelevant. The aim of the future research is to extend the modeling framework presented in [48] and [49] to predict the application runtime with better accuracy and allow it to evolve for developing resources. Further, other important HPC resource parameters will be included in the framework including load balancing, CPU-core count, accelerator count, and CPU-to-accelerator ratio, will be included in the framework. This future research provides the developer the ability to enter key parameters of the system and application, and the performance prediction framework will return the expected runtime and a suggested optimal allocation of the resources. The framework will be tested with numerous benchmarks including those developed in this thesis for thorough verification and confirmation, thereby establishing the framework's efficacy to predict application resource allocations quickly and effectively.

The performance modeling framework that will be developed can be verified and extended with observations made in this thesis. For example, it was seen that the Izhikevich SNN performed better on multi-GPU systems while the K-Means algorithm showed best performance using multi-core. The framework will take all architectures available into account and based on algorithm parameters and characteristics determine which architectures would lead to the best overall performance.

# References

- [1] Intel Xeon Phi Coprocessor Developer's Quick Start Guide.  
<https://software.intel.com/en-us/articles/intel-xeon-phi-coprocessor-developers-quick-start-guide>
- [2] OpenCL – The open standard for parallel programming of heterogeneous systems.  
<http://khronos.org/opencvl>
- [3] CUDA C Programming Guide.  
<http://docs.nvidia.com/cuda/cuda-c-programming-guide>
- [4] Introduction to InfiniBand.  
[http://www.mellanox.com/pdf/whitepapers/IB\\_Intro\\_WP\\_190.pdf](http://www.mellanox.com/pdf/whitepapers/IB_Intro_WP_190.pdf)
- [5] Wilen, Adam, Justin Schade, and Ron Thornburg. *Introduction to PCI Express*. Santa Clara: Intel Press, 2003.
- [6] MacQueen, James. "Some methods for classification and analysis of multivariate observations." Proceedings of the fifth Berkeley symposium on mathematical statistics and probability. Vol. 1. No. 14. 1967.
- [7] Izhikevich, Eugene M. "Simple model of spiking neurons." *IEEE Transactions on neural networks* 14.6 (2003): 1569-1572.
- [8] Chvatal, Václav, and David Sankoff. "Longest common subsequences of two random sequences." *Journal of Applied Probability* (1975): 306-315.
- [9] Knuth, Donald E., James H. Morris, Jr, and Vaughn R. Pratt. "Fast pattern matching in strings." *SLAM journal of computing* 6.2 (1977): 323-350.
- [10] Farivar, Reza, et. Al. "A Parallel Implementation of K-Means Clustering on GPUs." *PDPTA*. 2008.

- [11] Zechner, Mario, and Michael Granitzer. "Accelerating k-means on the graphics processor via cuda." *Intensive Applications and Services, 2009. INTENSIVE'09. First International Conference on*. IEEE, 2009.
- [12] Hong-Tao, Bai, et al. "K-means on commodity gpus with cuda." *Computer Science and Information Engineering, 2009 WRI World Congress on*. Vol. 3. IEEE, 2009.
- [13] Li, You, et al. "Speeding up k-means algorithm by gpus." *Computer and Information Technology (CIT), 2010 IEEE 10<sup>th</sup> International Conference on*. IEEE, 2010.
- [14] Izhikevich, Eugene M. "Which model to use for cortical spiking neurons?." *IEEE transactions on neural networks* 15.5 (2004): 1063-1070.
- [15] Morris, Catherine, and Harold Lecar. "Voltage oscillations in the barnacle giant muscle fiber." *Biophysical journal* 35.1 (1981): 193-213.
- [16] Wilson, Hugh T. "Simplified dynamics of human and mammalian neocortical neurons." *Journal of theoretical biology* 200.4 (1999): 375-388.
- [17] Hodgkin, Alan L., and Andrew F. Huxley. "A quantitative description of membrane current and its application to conduction and excitation in nerve." *The Journal of physiology* 117.4 (1952): 500.
- [18] Fidjeland, Andreas K., and Murray P. Shanahan. "Accelerated simulation of spiking neural networks using GPUs." *Neural Networks (IJCNN), The 2010 International Joint Conference on*. IEEE, 2010.
- [19] Gupta, Ankur, and Lyle N. Long. "Character recognition using spiking neural networks." *Neural Networks, 2007. IJCNN 2007. International Joint Conference on*. IEEE, 2007.
- [20] Han, Bing, and Tarek M. Taha. "Acceleration of spiking neural network based pattern recognition on NVIDIA graphics processors." *Applied Optics* 49.10 (2010): B83-B91.
- [21] Bhuiyan, Mohammad A., Vivek K. Pallipuram, and Melissa C. Smith. "Acceleration of spiking neural networks in emerging multi-core and GPU architectures." *Parallel & Distributed Processing, Workshops and Phd Forum (IPDPSW), 2010 IEEE International Symposium on*. IEEE, 2010.

- [22] Pallipuram, Vivek K., Mohammed A. Bhuiyan, and Melissa C. Smith. "Evaluation of GPU architectures using spiking neural networks." *Application Accelerators in High-Performance Computing (SAAHPC), 2011 Symposium on*. IEEE, 2011.
- [23] Izhikevich, Eugene M. "Polychronization: computation with spikes." *Neural computation* 18.2 (2006): 245-282.
- [24] Khajeh-Saeed, Ali, Stephen Poole, and J. Blair Perot. "Acceleration of the Smith-Waterman algorithm using single and multiple graphics processors." *Journal of Computational Physics* 229.11 (2010): 4247-4258.
- [25] McGuiness, Timothy, et al. "High Performance Computing on GPU Clusters." *Submitted to the SLAM Journal of Scientific Computing* (2010).
- [26] Steinbrecher, Johann, Cesar J. Philippidis, and Weijia Shang. "A Case Study of Implementing Supernode Transformations." *International Journal of Parallel Programming* 42.2 (2014): 320-342.
- [27] Ray, Siddheswar, and Rose H. Turi. "Determination of number of clusters in k-means clustering and application in colour image segmentation." *Proceedings of the 4<sup>th</sup> international conference on advances in pattern recognition and digital techniques*. 1999.
- [28] Huang, Zhexue. "Extensions to the k-means algorithm for clustering large data sets with categorical values." *Data mining and knowledge discovery* 2.3 (1998): 283-304.
- [29] Oyelade, I. J., O. O. Oladipupo, and I. C. Obagbuwa. "Applications of k Means Clustering algorithm for prediction of Students Academic Performance." *arXiv preprint arXiv: 1002.2425* (2010).
- [30] Haykin, Simon. "Neural networks: a comprehensive foundation 2<sup>nd</sup> edition." *Upper Saddle River, NJ, the US: Prentice Hall* (1999).
- [31] Pallipuram, Vivek K. "Acceleration of spiking neural networks on single-GPU and multi-GPU systems." *Master's Thesis, Clemson University* (2010).
- [32] Ghosh-Dastidar, Samanwoy, and Hojjat Adeli. "A new supervised learning algorithm for multiple spiking neural networks with application in epilepsy and seizure detection." *Neural Networks* 22.10 (2009): 1419-1431.

- [33] Bereg, Sergey, and Binhai Zhu. “RNA multiple structural alignment with longest common subsequences.” *Computing and Combinatorics*. Springer Berlin Heidelberg, 2005. 32-41.
- [34] Iyer, Srikanth K., and Barkha Saxena. “Improved genetic algorithm for the permutation flowshop scheduling problem.” *Computers & Operations Research* 31.4 (2004): 593-606.
- [35] Message Passing Interface (MPI)  
<https://computing.llnl.gov/tutorials/mpi>
- [36] Palmetto User’s Guide (Clemson University Palmetto Cluster)  
<http://citi.clemson.edu/palmetto/pages/userguide.html>
- [37] AMD Phenom II Processors  
<http://www.amd.com/en-us/products/processors/desktop/phenom-ii>
- [38] Intel Core Duo  
<http://www.intel.com/products/processor/core2duo/specifications.htm>
- [39] Intel Xeon Processor E5-2600 Product Family  
[http://download.intel.com/newsroom/kits/xeon/e5/pdfs/Intel\\_Xeon\\_E5\\_Factsheet.pdf](http://download.intel.com/newsroom/kits/xeon/e5/pdfs/Intel_Xeon_E5_Factsheet.pdf)
- [40] McClanahan, Chris. “History and Evolution of GPU Architecture.” *A Paper Survey* <http://mcclanahoochie.com/blog/wpcontent/uploads/2011/03/gpu-hist-paper.pdf> (2010).
- [41] GeForce, NVIDIA. “8800 GPU Architecture overview.” *Technical Brief TB-02787-001 v0.9* (2006).
- [42] NVIDIA’s Next Generation CUDA Compute Architecture: Fermi – Whitepaper  
[http://www.nvidia.com/content/PDF/fermi\\_white\\_papers/NVIDIA\\_Fermi\\_Compute\\_Architecture\\_Whitepaper.pdf](http://www.nvidia.com/content/PDF/fermi_white_papers/NVIDIA_Fermi_Compute_Architecture_Whitepaper.pdf)
- [43] NVIDIA’s Next Generation CUDA Compute Architecture: Kepler GK110 – Whitepaper  
<http://www.nvidia.com/content/PDF/kepler/NVIDIA-Kepler-GK110-Architecture-Whitepaper.pdf>
- [44] NVIDIA HyperQ Documentation  
[http://docs.nvidia.com/cuda/samples/6\\_Advanced/simpleHyperQ/doc/HyperQ.pdf](http://docs.nvidia.com/cuda/samples/6_Advanced/simpleHyperQ/doc/HyperQ.pdf)

- [45] NVIDIA CUDA Compiler Driver NVCC  
<http://docs.nvidia.com/cuda/cuda-compiler-driver-nvcc>
- [46] CUDA Toolkit  
<https://developer.nvidia.com/cuda-toolkit>
- [47] Zhang, Yao, and John D. Owens. “A quantitative performance analysis model for GPU architectures.” *High Performance Computer Architecture (HPCA), 2011 IEEE 17<sup>th</sup> International Symposium on*. IEEE, 2011.
- [48] Bhuiyan, M. A. “Performance Analysis and Fitness of GPGPU and Multi-Core Architectures for Scientific Applications.” *Ph.D. Dissertation, Clemson University* (2011).
- [49] Pallipuram, V. K. “Exploring Multiple Levels of Performance Modeling for Heterogeneous Systems.” *Ph.D. Dissertation, Clemson University* (2013).