Clemson University

TigerPrints

All Theses

Theses

8-2013

Statistical Regression Methods for GPGPU Design Space Exploration

Nimisha Raut
Clemson University, nim.raut12@gmail.com

Follow this and additional works at: https://tigerprints.clemson.edu/all_theses

Part of the Computer Engineering Commons

Recommended Citation

Raut, Nimisha, "Statistical Regression Methods for GPGPU Design Space Exploration" (2013). *All Theses*. 1701.
https://tigerprints.clemson.edu/all_theses/1701

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.

STATISTICAL REGRESSION METHODS FOR GPGPU DESIGN SPACE EXPLORATION

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
Nimisha Shirish Raut
August 2013

Accepted by:
Dr.Melissa Crawley Smith, Committee Chair
Dr.Stanley Birchfield
Dr.Walter Ligon

ABSTRACT

General Purpose Graphics Processing Units (GPGPUs) have leveraged the performance and power efficiency of today's heterogeneous systems to usher in a new era of innovation in high-performance scientific computing. These systems can offer significantly high performance for massively parallel applications; however, their resources may be wasted due to inefficient tuning strategies. Previous application tuning studies pre-dominantly employ low-level, architecture specific tuning which can make the performance modeling task difficult and less generic. In this research, we explore the GPGPU design space featuring the memory hierarchy for application tuning using regression-based performance prediction framework and rank the design space based on the runtime performance. The regression-based framework models the GPGPU device computations using algorithm characteristics such as the number of floating-point operations, total number of bytes, and hardware parameters pertaining to the GPGPU memory hierarchy as predictor variables. The computation component regression models are developed using several instrumented executions of the algorithms that include a range of FLOPS-to-Byte requirement. We validate our model with a *Synchronous Iterative Algorithm* (SIA) set that includes Spiking Neural Networks (SNNs) and Anisotropic Diffusion Filtering (ADF) for massive images. The highly parallel nature of the above mentioned algorithms, in addition to their wide range of communication-to-computation complexities, makes them good candidates for this study. A hierarchy of implementations for the SNNs and ADF is constructed and ranked using the regression-based framework. We further illustrate the *Synchronous Iterative GPGPU Execution*

(SIGE) model on the GPGPU-augmented Palmetto Cluster. The performance prediction framework maps appropriate design space implementation for 4 out of 5 case studies used in this research. The final goal of this research is to establish the efficacy of the regression-based framework to accurately predict the application kernel runtime, allowing developers to correctly rank their design space prior to the large-scale implementation.

DEDICATION

I dedicate this thesis to my family for their constant support and encouragement.

ACKNOWLEDGMENTS

First and foremost, I would like to express my sincere gratitude to my advisor Dr. Melissa C. Smith for her invaluable guidance and support throughout this thesis. I am especially grateful for her advice driven by her inspiring experience and knowledge, which has shown me the path to excel.

I thank Dr. Walter B. Ligon and Dr. Stanley Birchfield for being on my committee and reviewing my work.

I would also like to acknowledge the members of Future Computing Technology Lab of Clemson University for their support and cooperation. My special thanks to Vivek Pallipuram for the constant encouragement and valuable input that he provided me during the course of this research.

I am also thankful to my friends Deepthi and Praveen who have always stood by my side.

My gratitude towards my family, without them this thesis would not have been possible.

TABLE OF CONTENTS

Table of Contents (Continued)

Page

LIST OF TABLES

List of Tables (Continued)

LIST OF FIGURES

CHAPTER 1

INTRODUCTION

The High Performance Computing (HPC) community is tackling complex science and engineering problems using applications and simulators that demand high bandwidth and very high compute capabilities. For many years, these increasing performance demands relied on improving the single-core performance by increasing the clock rates and implementing execution optimizations such as instruction-level parallelism. Various limitations such as power consumption, memory wall, and clock wall left an opportunity in the HPC community for architecture alternatives to single-core processors. The search for alternatives led to the development of parallel computing architectures. Since the introduction of first multi-core processor by IBM in 2001, a surge of multi-core and many-core processors flooded the HPC community. Some of the multi-core processors are capable of achieving more than one trillion floating point operations per second (1 Teraflops) [1]. Despite of these advances, the continuing constraints on scalability and power in multi-core architectures and the continued demand for further performance improvement have led the HPC community to look at the various heterogeneous computing resources.

A heterogeneous computing system consists of a general-purpose multi-core processor and one or more accelerators such as a general-purpose graphics processing unit (GPGPU) or a field programmable gate array (FPGA). The inherent massively-parallel computing power of GPGPUs along with low cost and ease of programming make them the most widely used heterogeneous systems for parallel applications.

GPGPU computing offers unprecedented application performance by offloading compute-intensive portions of the application to the GPGPU device, while the serial computations, data movement, and management is done by the host processor. The landscape of HPC was changed with the introduction of the Fermi architecture by Nvidia in 2009 [2]. The Fermi-based GPGPUs coupled with the advent of general-purpose programming environments like Nvidia's Compute Unified Device Architecture (CUDA) [3] offered a tremendous performance leap compared to earlier GPGPUs. Further, with the introduction of the Tesla K20X GPGPU accelerators based on the Kepler architecture [4], GPGPUs continue to propel advances in mainstream energy-efficient computing by introducing features such as the *Next Generation Streaming Multiprocessor* (SMX), *Hyper-Q technology*, and *Dynamic Parallelism*. These radical features boost the application performance by nearly 10x as compared to the earlier architecture performance.

Heterogeneous computing has found its niche in high-performance computing, but most of its computing resources are under-utilized due to various limitations. Factors such as inefficient application mapping, load-balancing, and tuning in the existing parallel large-scale applications prevent complete utilization of the computing potential of the heterogeneous systems. These factors lead to poor application speed-up and sub-optimal scaling. The extraction of optimized performance from the heterogeneous system requires effective utilization of memory and bandwidth, and efficient load-balancing between the CPU host and the GPGPU accelerators. Additionally as the GPGPU architecture evolves, it is imperative that programs be tuned for specific GPGPU architectures to obtain maximum performance. Performance prediction models allow developers to employ design space exploration to optimize the application according to the various computing architecture features and to predict scalability and application runtime prior to large scale implementations [5, 6, 7 and 8]. Although several performance prediction models exist, most of them employ architecture specific tuning that can make the performance modeling task difficult. In addition, such an approach may vary from one architecture generation to another. With the above as motivation, we explore how the GPGPU design space featuring a memory hierarchy can be modeled to allow a developer to analyze and predict the algorithm performance with the given level of system abstraction.

In this research, we explore a performance framework of the GPGPU design space using a regression-based approach. The regression-based performance prediction framework developed using the *Synchronous Iterative GPGPU Execution* (SIGE) model proposed in [9], enables kernel runtime prediction prior to the actual large-scale implementation. Using the prediction framework, performance prediction can be achieved without detailed knowledge of the underlying computing architecture. The prediction framework aims to model the performance of the GPGPU device computations of *Synchronous Iterative Algorithms (SIAs)*, thus allowing the developer to rank the GPGPU architecture across the design space based on the predicted runtime performance. The regression-based framework can be broken into two primary components: the computation component that models the GPGPU and host computations; and the communication component that models the network-level communications. For the relevance of our research, we develop the regression models only for the computation component (device computations) using algorithm characteristics such as the number of floating-point operations and hardware parameters such as the amount of memory accessed for computations as predictor variables. The runtime data is collected using several small instrumented executions of the algorithm with a range of communication-to-computation requirements. Additionally, we illustrate the SIGE model [9] for multi-node GPGPU implementation and evaluate the strengths, weaknesses and opportunities for the model.

The case studies used in this research constitute a set of SIA algorithms that includes large-scale Spiking Neural Network (SNN) [10] and non-linear anisotropic diffusion filtering (ADF) [11] for massive images. The highly parallel nature of these algorithms, in addition to their wide range of communication-to-computation complexities, makes them good candidates for the GPGPU design space exploration study. The SNN models are used to construct a two-level character recognition network capable of recognizing 48 alpha-numeric characters [10]. ADF is the most widely used noise removal technique well known for preserving the sharp edges and finer details in an image [12]. Both algorithms are discussed in detail in later chapters. The heterogeneous Palmetto Cluster [13] is used for the implementation of the aforementioned applications. The performance prediction framework maps appropriate design space for 4 out of 5 case studies used in this research. The key contributions of this research are:

1) Application of the performance prediction framework to a single-node GPGPU problem on a GPGPU cluster for design space exploration.
2) Ranking the GPGPU design space, thereby mapping an implementation from the GPGPU design space to an application.
3) Illustration of the SIGE model on the Palmetto Cluster and study of the challenges involved in using the regression framework based on the SIGE model for other clusters and evaluating the Strengths, Weaknesses and Opportunities (SWO) for the model.

The remainder of the thesis is organized as follows. Chapter 2 provides the literature review and related work. Chapter 3 gives an overview of the GPGPU architecture, the programming model used in the study, and a background on the case studies and regression analysis theory used in this research. Chapter 4 presents the details of the experimental setup and various implementations used. Chapter 5 explains the regression model for the GPGPU design space exploration and presents the results and analysis. The thesis is concluded in Chapter 6 with conclusions and suggestions for future work.

CHAPTER 2

RELATED WORK

Researchers have conducted several design space exploratory studies for heterogeneous systems, such as GPGPU-based clusters. In this chapter, we discuss some of the prominent studies targeting GPGPU-based systems and further discuss some of the architecture studies using Spiking Neural Networks (SNN) and Anisotropic Diffusion Filters (ADF). The chapter is structured as follows. Section 2.1 highlights the performance modeling studies for GPGPU-based systems that enable design space exploration. Section 2.2 discusses architecture studies conducted using SNNs and ADF. The chapter is concluded with a summary in Section 2.3.

2.1 Performance Modeling Studies for GPGPU-Based Systems

In [7], the authors present a performance prediction model for GPGPU-based systems that incorporates various components of the GPGPU architecture such as scheduling, memory hierarchy, and pipelining. The model is developed with a combination of the BSP model of Valiant [14], the PRAM model of Fortune and Wyllie [15], and the extension to the PRAM model proposed by Gibbons et al. called the QRQW model [16]. The proposed model derives a relationship among the various components of the GPGPU architecture including the number of cores, effects of memory latency, memory access conflicts, computing cost, scheduling, and pipelining to analyze pseudo-code for a CUDA kernel and finally predict the performance of an application. Unlike the prediction framework used in this research, the model in [7] does not consider texture memory along with global and shared memory within the design space. Additionally, as the model is developed from three earlier models namely, the BSP model, PRAM model, and the QRQW model, it is necessary to have a thorough knowledge of these three models.

In [17], the authors use a micro-benchmark based approach to develop a throughput performance model for Nvidia GeForce 200-series GPGPU. The authors first design micro-benchmarks, observe the benchmark results, and then derive a simple throughput model for the instruction pipeline, shared memory, and global memory costs. Using real world matrix problems, the authors achieved prediction performance with 5–15% error rate. Unlike the prediction framework used in this thesis, the model in [17] focuses on identifying the performance bottlenecks for guiding programmers and architects for optimizations rather than large-scale performance predictions. Additionally, we consider a larger design space compared to the design space used in [17] that focuses only on three architecture components as mentioned earlier.

In [18], the authors propose an analytical model that estimates the execution time of GPGPU kernels by estimating the number of parallel memory requests (memory-warp parallelism) using the number of running threads, memory bandwidth, and memory bank parallelism, and the number of computations (computation-warp parallelism) that represents the number of warps that the Streaming Multiprocessor (SMP) can execute during one memory warp waiting period. The model anticipates the cost of memory accesses based on the degree of memory-warp parallelism and computation-warp parallelism, thereby estimating the overall execution time of a program. The geometric mean error rate of 5.4% is achieved for micro-benchmarks and 12.3% for GPGPU applications. Although the model provides good results, the model is specifically designed for the Nvidia Tesla architecture used in the GeForce-8 series, thereby not very useful for current GPGPU architectures. Additionally, computing the memory wraps for estimating the runtime prediction involves complex calculations.

In [19], the authors introduce an abstract interpretation of a GPGPU kernel, *work flow graph*, to estimate the GPGPU kernel time. The authors used micro-benchmarks to characterize GPGPU micro-architecture events such as incoherent memory accesses, shared memory bank conflicts, and control flow divergence. The authors used benchmarks such as dense matrix multiplication, Fast Fourier Transform, prefix sum scan, and sparse matrix-vector multiplication for validating the model. Although, the model is not tightly coupled to any specific GPGPU architecture, the model depends significantly on the GPGPU micro-architecture. Hence, it is imperative to have in-depth knowledge of the micro-architecture for accurate runtime predictions.

In [6], the authors propose a model to predict execution time for GPGPU applications by varying the number and configuration of the GPGPUs, and the size of the input data set. The authors determine the time it takes to compute a single element (smallest unit of computation involved with the problem being considered) of a problem by the reference GPGPU device and further estimate the algorithm execution time on M GPGPU devices, where M is the number of devices. Micro-benchmarked throughput values were used for modeling the PCI-Ex bus and network-level transactions. The authors used six scientific applications as case studies and achieved an average performance prediction error up to 11%, and 40% maximum error in a single-case. The authors report good prediction results for their multi-GPGPU implementation that is developed from a reference GPGPU implementation. The prediction framework used in our research can be employed for the aforementioned basic reference GPGPU implementation, as our prediction framework uses easily accessible algorithm parameters for prediction modeling.

In [5], the authors propose an automated GPGPU performance exploration tool based on stepwise regression modeling. The tool sparsely and randomly samples parameter values from the GPGPU design space and simulates regression designs. The automated tool then uses the earlier sampled simulations to build a performance estimator that identifies the most significant architectural parameters and their interactions for accurate application runtime prediction. The tool was used to evaluate the runtime for 11 GPU applications, with less than 1.1% average error. Although the model provides good prediction results, the proposed tool uses a complex methodology for runtime prediction when compared to the prediction framework used in this research that uses easily accessible algorithm parameters such as FLOPS and computational bytes.

In [20], the authors propose an integrated analytical and profile-based performance model to predict the CUDA kernel execution time for Sparse Matrix Vector Multiplication (SpMV). The modeling is divided into two phases: 1) profiling phase where benchmark matrices are generated based on the device hardware properties and 2) analytical model development phase where a relationship is established between the maximum number of rows the target GPGPU device can execute at a time, the number of non-zero elements per row in the target matrix, and execution times of the benchmark matrices obtained in phase 1. The authors report performance differences of less than 10% between actual and predicted runtime. Although the model predicts satisfactorily, the prediction approach is tightly-coupled to the SpMV application. Additionally, the benchmark matrices also must be regenerated as the GPGPU architecture changes.

We discussed some of the significant performance modeling studies for GPGPU-based systems. Although these models are sufficiently accurate, they present some limitations. The models discussed above require detailed knowledge of the GPGPU architecture for viable performance prediction. Additionally, the models are GPGPU architecture-specific, and thereby require modification with evolving GPGPU architectures. Additionally, several of the models employ complex methodology making the modeling task difficult. Unlike the previous modeling approaches, the prediction framework used in this research uses easily available application and hardware parameters, making the entire modeling task less complex. The regression-based framework used in this research is motivated by [9] and uses modeling concepts from [21, 8, and 22].

2.2 Architecture Studies for SNNs and ADF

2.2.1 SNNs

In this section, we discuss some of the prominent architecture studies conducted using large-scale SNN simulations. In [23], the author implemented a two-level character recognition network for SNNs using Nvidia's Tesla C870. The author also investigated an initial multi-GPU implementation to study the problem partitioning for simulating large-scale SNNs on a GPGPU-based cluster. In [24], the authors compared the performance of Nvidia’s Fermi architecture and AMD/ATi’s Radeon architecture; and CUDA and OpenCL programming models using SNN simulations. The authors presented various implementations, where they successively added optimization techniques associated with the two programming models and presented the affect of the network size scaling on the performance. The application speed-up reported was 1095× against a serial implementation. In [21], the authors analyzed the performance of various architecture such as Nvidia GPUs, and multi-core processors such as Intel Xeon, AMD Opteron, IBM’s Cell Broadband Engine using large-scale SNN simulations. The authors report a maximum speed-up of 574x for the GPGPU implementation. In [25], the authors investigated GPGPU cluster-based implementations of the Hodgkin-Huxley (HH) and Izhikevich SNN models using a two-level character recognition network. They reported GPGPU speed-ups of 24.6x and 177x for the Izhikevich and HH models, respectively.

2.2.2 ADF

There are several research studies in the literature that are conducted using anisotropic diffusion filtering in parallel computing. In [26], the authors implemented an anisotropic diffusion filter for parallel and distributed systems. The implemented filter used 30 iterations and a neighborhood factor of 15. A performance gain of 81.9% was achieved by their point-to-point and 93.8% by collective communication implementations when compared to the execution on a single compute node. The authors report collective communication efficiency of 21% over their point-to-point implementation. In [27], the authors illustrated the application of auto-tuning to a 27-point stencil on a wide range of cache-based multi-core architectures. The results showed that Intel's Nehalem architecture [28] delivered the best performance and achieved more than 6x speedup compared to the previous generation architectures. In [29], the authors implemented a GPGPU cluster-based implementation of the non-linear anisotropic diffusion filter. The implementation achieved a speed-up of 29x over an equivalent MPI-only implementation and exhibited reasonable scaling behavior that improved with the size of the images. In [30], the authors presented a hybrid parallel implementation of gradient domain processing for massive images using MPI, threading, and a CUDA-based GPGPU component. The authors used two GPGPU clusters and two data sets to demonstrate the performance and scalability of their implementation. The authors report good weak scalability results (efficiency above 80%) but the strong scalability performs well only up to 16-nodes for both the clusters.

2.3 Summary

In this chapter, we discussed some of the prominent performance modeling efforts targeting GPGPU-based heterogeneous systems that aide in design space exploratory studies. Further, we explained that unlike the performance models discussed from the literature, the prediction framework used in this research aims to provide easy and accurate runtime prediction, thereby guiding application developers when selecting a platform that best fits their application design space. Additionally, we discussed architecture studies conducted using SNNs and ADF.

CHAPTER 3

BACKGROUND

This chapter presents a background on the GPGPU architecture, Compute Unified Device Architecture (CUDA) programming model, the case studies and overview of the regression analysis theory used in this research. The chapter is structured as follows: Section 3.1 discusses the GPGPU Fermi Architecture [2] and CUDA framework [3]; Sections 3.2 and 3.3 provide background on the two case studies – Spiking Neural Networks (SNNs) [23] and the Non-Linear Anisotropic Diffusion Filter (ADF) [12]; The Regression Analysis Theory is described in Section 3.4; The chapter is concluded in Section 3.5 with a summary.

3.1 GPGPU Architecture

The introduction of fully programmable graphics card has radically changed the rate of evolution of the GPUs. The previous GPU architectures were designed with the concept of a fixed-function graphics pipeline used for 2-D or 3-D image rendering [31]. Nvidia introduced the GeForce 8 series in 2006, which revolutionized the GPU market, exposing the GPU architecture as a massively parallel processor for general-purpose computing. The G80 (GeForce 8800 GTX) [32] was the first GPGPU architecture to include a fully programmable unified processor (programmable shaders) called Streaming Processors (SPs). The SPs performed vertex transformations, pixel shading, and geometry computation. A group of SPs execute Single Instruction Multiple Data (SIMD) instructions, thereby providing massive parallelism. The G80 also introduced shared memory in each SP, a fast on-chip memory used for storing data and barrier synchronization for inter-thread communication. Several GPGPUs used for HPC today are in concurrence with the GeForce 8800 GTX architecture. A significant milestone by Nvidia in GPGPU computing was the introduction of the Fermi architecture in September 2009 [2], which drastically changed the face of the GPGPU computing as will be explored in the next subsection. Nvidia's latest GPU architecture, codenamed "Kepler" launched in 2012 [4] is built on the foundation established by the Fermi GPU architecture. The GK110 Kepler GPGPUs, have 5 GB of GDDR5 memory, 64 KB L1 cache/shared memory, 48KB read-only cache, 1536 KB L2 cache, and a quad warp scheduler. The Kepler GPGPU family introduces features such as the *Next Generation Streaming Multiprocessor* (SMX), that comprises of 192 CUDA cores, for a total of 1536 cores in the entire GPU, providing a tremendous performance boost at a lower power consumption when compared to the earlier GPGPUs. The Kepler GPGPUs also feature the *Dynamic Parallelism* that enables it to dynamically spawn new threads from the device kernel without going back to the host CPU. Furthermore, the *Hyper-Q* technology enables multiple CPU cores to launch work on a single GPU simultaneously, thereby radically increasing the GPU utilization and reducing the CPU idle time. For our experiments we have used the Fermi-based Tesla M2075.

3.1.1 Nvidia's Fermi Architecture

With the introduction of the Fermi GPU in 2009, Nvidia took a significant leap in the HPC industry, thereby helping to solve computationally intensive tasks efficiently.

The Fermi architecture consists of an array of streaming multiprocessors (SMPs), where each multiprocessor is comprised of a group of scalar processors, a double-precision (DP) unit, shared memory for thread cooperation, and texture addressing and texture fetch units. A *thread*, which is the basic unit of execution on the GPGPU device, is executed on the scalar processors within the SMPs. A group of threads, called a *thread block*, is executed on the multiprocessors. The blocks are further divided into SIMD groups of 32 threads called warps, which are further divided into groups of 16 threads called half-warps. The Fermi architecture consists of 16 SMPs made up of 32 cores each, making a total of 512 CUDA cores. Each SMP has an integer arithmetic logic unit (ALU) along with a floating point unit (FPU). The Fermi GPGPUs support a dual warp scheduler, capable of issuing and executing two warps concurrently. SMPs have 6 GB of GDDR5 DRAM memory, 16 load/store units, 4 special function registers (SFUs), a sizable register file, a configurable 64KB shared memory/L1 cache and the SMPs share L2 cache. Figure 3.1 shows the organization of SMPs on the Fermi architecture. The Fermi-based Tesla M2075 used in our research is discussed below.

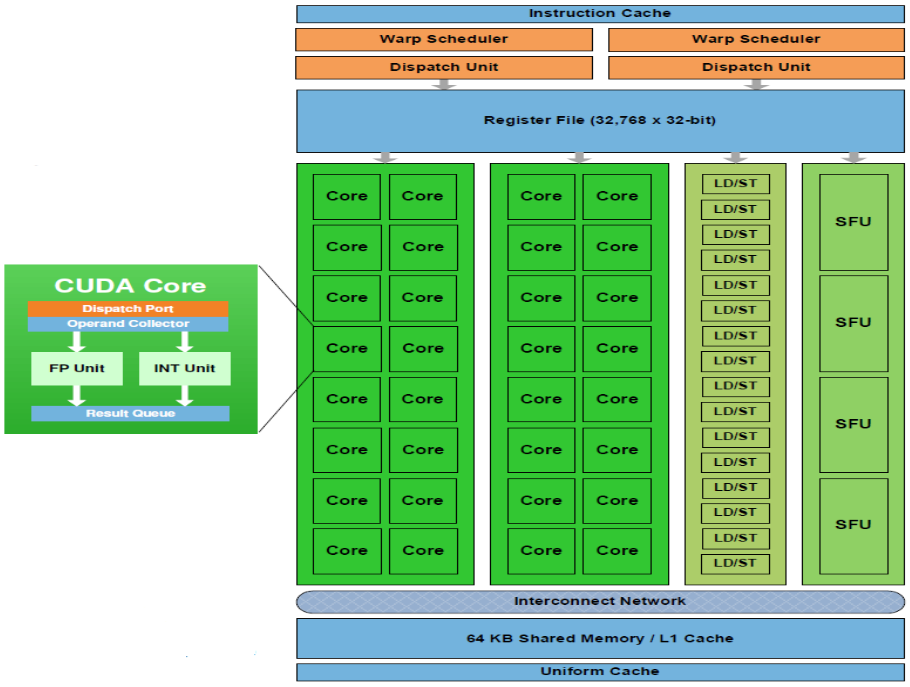

Figure 3.1: Fermi Streaming Multiprocessor (SMP) [3]

The Fermi-based Tesla M2075 used in this research belongs to Compute Capability 2.0 and comprises of 14 multiprocessors (448 cores), 6 GB of GDDR5 DRAM memory, 64 KB shared memory/L1 cache per multiprocessor, 768 KB L2 cache, 64 KB constant memory and operates at a clock rate of 1.15 GHz. The architecture can theoretically offer 1030 Gigaflops of single-precision floating-point performance and 515 Gigaflops of double-precision floating-point performance. The Tesla system's GDDR interface offers memory bandwidth up to 150 GB/s. More information on the Fermi GPGPU architecture and Tesla M2075 can be found in [2] and [33].

3.1.2 Nvidia CUDA Framework

The Compute Unified Device Architecture (CUDA) programming model leverages the power of GPGPUs by providing a C-like Application Programming

Interface (API) for various applications. In CUDA, the user-defined device functions called *kernels* are executed on the GPGPU device. Commonly, only one kernel can be executed on the GPGPU at a time, but more than one kernel can be executed sequentially. All the *threads* created in an application execute the kernel in parallel. The threads are accessed in kernel functions using built-in variables: *threadIdx*, *blockIdx*, and *blockDim*. The thread blocks can be arranged in one-dimensional, two-dimensional or three-dimensional *grid*. Figure 3.2 shows the CUDA thread hierarchy.

In the CUDA memory hierarchy each thread has its own local memory and a set of registers. The local memory is not located on the chip and resides in the external device memory. Threads in a block synchronize with each other using the shared memory and the shared memory is local to that block. All threads have access to a global memory that resides in off-chip DRAM. The constant memory and texture memory are off-chip, cached, and read-only memories. The texture cache is usually bound to either, pitch memory or CUDA arrays or to the global memory itself.

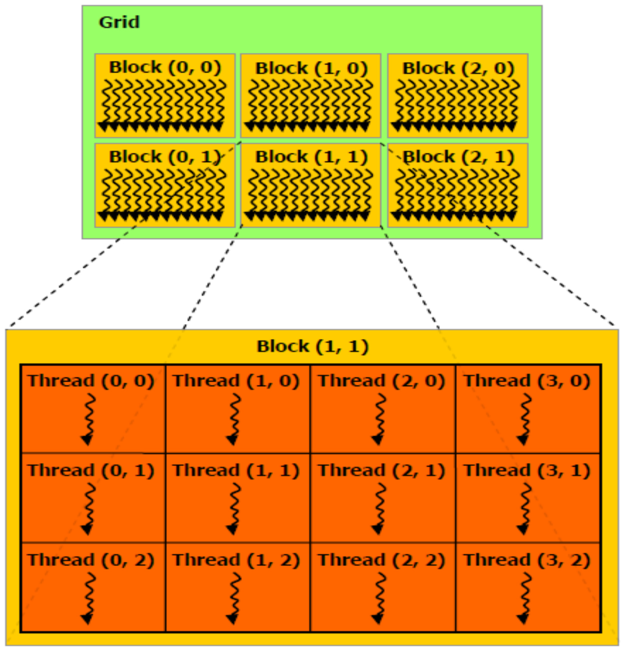

Figure 3.2: Grid of thread blocks in CUDA [3]

Various optimization strategies offered by CUDA can be found in [3]. The three primary optimization strategies offered by CUDA that are used in this research are Memory Optimization, Execution Configuration Optimization, and Instruction Optimization. Memory optimizations aim at reducing the bottleneck presented due to the large amount of data transfer between the device and the host over the relatively low bandwidth PCI-Ex bus. One way to resolve this bottleneck is by transferring the relevant data to the device memory for processing. Once all of the operations are finished, the final output is transferred back to the host. Another technique to reduce global memory latency is the use of cache and on-chip shared memory. With the introduction of L1 and L2 caches in the recent GPGPU architectures, the user can configure the amount of L1 cache and shared memory. Furthermore, the data in the global memory can be cached either in L2 only, or both L1 and L2 caches. The on-chip shared memory also assists in thread synchronization in a block, allowing coordination amongst the threads. The use of registers can also assist in reducing the frequent global memory accesses, in addition, avoiding the bank conflicts that occur with the shared memory accesses. The cached texture memory can also provide performance improvements by taking advantage of the data locality in the application.

The Execution Configuration optimization is related to the number of threads per block as well the dimensions of the thread block. The optimization manages the multiprocessor occupancy of the application. Multiprocessor occupancy is defined as the ratio of the number of warps running on the SMP to the maximum number of warps that can physically run on the SMP. Appropriately selecting the number of threads per block or the dimension of the *threadblock*, is an effective way to hide the memory latency in the kernels. Additionally, it is important that the number of threads be high enough to keep the hardware busy and efficiently utilize the memory or the compute bandwidth. Lastly, keeping the number of threads a multiple of 32 aids the coalescing of memory accesses. Coalescing of memory accesses enables all threads in a warp to complete the data access in one or more transactions.

Instruction optimization techniques used in this study consist of fast math functions and Reduced Conditional Statements (RCS). Fast math functions are capable of improving the performance at the cost of accuracy. Applications that require high accuracy should use fast math functions with caution. During execution of any algorithm, divergent paths taken within a warp are serialized that adversely affects the performance.

RCS optimizations reduce these divergent paths by avoiding branching instructions at compile time.

3.2 Spiking Neural Network (SNN)

The Synchronous Iterative Algorithm (SIA) used in this research satisfies two basic properties: 1) The synchronous property that implies that computations in an algorithm can occur simultaneously on multiple computing devices; and 2) The iterative property that implies that a single hardware operation or a combination of hardware operations specific to the algorithm can be repeated multiple times as required by the algorithm. In this sub-section, we discuss the Spiking Neural Network (SNN) models and two-level character recognition network used as the SIA case studies in this research. SNNs are highly biologically accurate models used to simulate a mammalian brain for capturing its functional and inference capabilities. The research presented in this thesis uses a two-level character recognition network that can recognize 48 alpha-numeric characters: English characters (A-Z), 10 numerals (0-9), 8 Greek letters and 4 symbols as mentioned in [10]. A *spiking* neuron fires an electric pulse, commonly referred to as *spike*, at certain time intervals, whose timing is a function of the input and hence this form of time encoding is used for processing information. Out of the several models proposed in [30], we use the following four models in this research. The Hodgkin-Huxley (HH) model [35], Morris-Lecar model [36], Wilson model [37], and the Izhikevich model [38]. These models were chosen as they encompass a spectrum of computation-to-communication requirements. The four models are described briefly below.

3.2.1. Four SNN Models

The Hodgkin-Huxley (HH) model [35] is considered to be the most accurate and the most important model in the neuroscience community till date. The model involves four equations and ten parameters describing neuron current activation and deactivation. The model takes 1200 FLOPS per millisecond for the complete neuron update. In our research, we have used 0.01 milliseconds time-step for the neuron update.

The Morris-Lecar (ML) model [37] is another biophysically meaningful model, replicating almost all of the spiking neuron properties. The ML equations include hyperbolic functions, making this model more complex than the two models mentioned later. The model takes 600 FLOPS per millisecond time-step for the neuron update. For our experiments, we have used 0.01 milliseconds time-step for the neuron update.

The Wilson model [38] attempts to model cortical neurons with a system of polynomial equations. The model in general takes 180 FLOPS per millisecond for the neuron update. The time-step of 0.01 milliseconds was used to evaluate the polynomial equations describing neuron dynamics.

In [36], Izhikevich developed a simple and very computationally efficient spiking neuron model that is almost as accurate as the HH model. Izhikevich was successful in reducing the complex HH model equations to a 2-D system of ordinary equations. Izhikevich's model requires only 13 FLOPS per neuron update and still sufficiently reproduces a majority of neuronal properties. In our research, we have used a 1 millisecond time-step (13 FLOPS per millisecond) for neuronal dynamics update.

A more detailed description of the four SNN models can be found in [23]. In Table 3.1, we summarize the FLOPS/Byte ratio for the four SNN models, which provides an algorithmic analysis of the aforementioned SNN models used in this study. The FLOPS/Byte ratio is an algorithm specific value and is defined as the ratio of the number of floating-point operations required for a complete neuron update (level-1 and level-2 of the two-level network) to the overall bytes requested (all model parameters and supporting data structures) for all of the neuron updates [23].

Table 3.1 FLOPS/Byte Ratio for SNN Models

Model	**FLOPS required for the complete neuron update**	**Bytes required for the complete neuron update**	**FLOPS/Byte Ratio**
HH	246	25	9.84
ML	147	17	8.65
Wilson	38	25	1.52
Izhikevich	13	13	1

3.2.2 The Two-Level Network

The SNN models discussed in the previous section are used for the large-scale SNN simulations using a two-level character recognition network based on [34]. The task of the network is to identify images from a training data set of 48 images (English characters (A-Z), 10 numerals (0-9), 8 Greek letters and 4 symbols). The level-1 neurons act as an input collection layer and the level-2 neurons act as output collection layer. The total number of neurons in the input level is equal to the total number of pixels in the test image as each neuron in level-1 corresponds to a pixel in the input image. Therefore, the level-1 is the most computationally intensive layer of the two-level network. The total number of neurons in the output layer is equal to the number of images in the database (that in 48 in our case), making level-2 less computationally dense. When an input image is presented to level-1, each neuron evaluates its membrane potential based on the pixel level presented and the neuron model chosen. This process is referred to as the *evaluation of neuron dynamics*. If the pixel is “on,” a constant current is supplied to the neuron for membrane potential evaluation. The input current equation for a level-2 neuron is:

$$I_j = \sum w_{ij} * f_i \tag{3.1}$$

In Equation 3.1, I_j is the net input current to the neuron j in level-2, w_{ij} is the weight of the synapse connecting neuron i in level-1 with the neuron j in level-2. A neuron in any level is said to have “fired” if its membrane potential crosses the threshold value for the selected neuron model. In our research, we accelerate the recognition phase of the network by implementing all of the level-1 neurons on the GPGPU device since this level is highly compute-intensive, while the less computationally dense level-2 neurons (input current accumulation and dynamics) are implemented on the host processor. Figure 3.3 illustrates the two-level character recognition network.

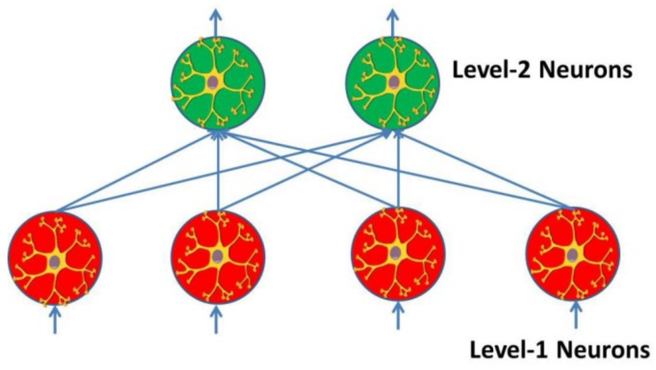

Figure 3.3: Two-Level Character Recognition Network

3.3 Non-Linear Anisotropic Diffusion Filter (ADF)

Images represent significant data in various image processing applications such as surveillance, medical imaging, etc. Although various advances have been made to capture and process these images in the most sophisticated ways, these applications are still prone to the surrounding noise signals. Various noise removal techniques have been tried and tested to eliminate different types of noise. Some of the techniques such as the median filtering and hybrid median filtering (bidirectional linear median filter) retain edge information but cause streaking and blotching effects in the processed image [39]. While a few techniques are computationally efficient and prone to boundary errors, others require an excessively large number of iterations. Thus each technique has its own advantages and disadvantages. The literature reports [40, 41 and 42] anisotropic diffusion filtering produces superior results compared to other noise reduction algorithms. Anisotropic diffusion filtering uses piecewise smoothing and immediate localization to reduce noise in an image and improves the overall signal-to-noise ratio. In piecewise smoothing of an image, the intra-region smoothing is preferred over inter-region smoothing. The immediate localization property causes regions along the boundaries of an image to be sharp and aligned with semantically meaningful boundaries at a given resolution. These properties of anisotropic diffusion filtering preserve the sharp edges and fine details in an image, making it a viable candidate for use in numerous image processing applications. Further details on anisotropic diffusion filtering can be found in [12].

In [11], the author implements a novel non-linear anisotropic diffusion filter based on the statistic-local open system proposed by Wu and Liu in [43]. As mentioned in [43], the order-statistic filters have two shortcomings. First, the order-statistic filters tend to ignore edge texture information. Second, the order-statistic filters cannot efficiently filter out the impulse noise from high-level noised images. The proposed filter in [11] overcomes the first shortcoming by processing only the estimated noised pixels in a single iteration, thereby only allowing for local diffusion. The value of the center pixel is then compared with the pixel value after the order-statistic filtering. If the difference in the values is above a threshold level K_{noise}, only then will the pixel be declared a noised pixel, otherwise it is declared a pure pixel. The noise estimated image for the nth iteration is represented by sgn^n and is given by Equation 3.2

$$\mathrm{sgn}^n = \begin{cases} 0 & if \left| med\left(u\right) - u \right| \geq K_{noise} \\ 1 & otherwise \end{cases} \tag{3.2}$$

where, u represents a pixel in the input image, $med(u)$ represents the value of the pixel after applying a median filter, and K_{noise} is a constant threshold.

The second shortcoming is addressed by using anisotropic diffusion filtering based on a local open system, where some pixels are labeled *convergences* and other pixels are labeled *origins*. The *convergence* pixels represent the energy flowing in, whereas the *origin* pixels represent the energy flowing out. The neighbors of noised pixels are declared as either *convergences* or *origin*s and their values remain unchanged. The authors claim that the image details are well preserved if the above two labels are chosen carefully. The authors also propose a new conduction coefficient c_i^n, to avoid any energy effects from the neighboring noise pixels as shown in Equation 3.3

$$c_i^n = \frac{1}{1+(\frac{\|\nabla u_i^n\|}{K})^2} \tag{3.3}$$

where, u_i^n represents the pixel in the i^{th} direction (i= N,S,E,W), ∇u_i^n represents the gradient in the direction i, and K is a constant. Equation 3.4 represents the proposed filter model in its iterative form

$$u^{n+1} = u^n + \lambda \left[\sum_{i=N,S,E,W} \mathrm{sgn}_i^n . c_i^n . \nabla u_i^n \right] \tag{3.4}$$

where, sgn_i^n represents the pixel value of the noise estimated image for n^{th} iteration in the direction i.

30 iterations are used in the implementation based on the iteration scheme used in [44]. The Peak Signal-to-Noise Ratio (PSNR) is used to evaluate the quality of an image and can be calculated with Equation 3.5

$$PSNR = 10 * \log_{10}(255^2 / MSE) \tag{3.5}$$

$$MSE = \sum\sum(u(x,y) - v(x,y))^2 / 256^2 \tag{3.6}$$

where u refers to the original noise free image, v represents the filtered output image, and MSE stands for mean squared error given by Equation 3.6

The anisotropic diffusion filtering scheme used in this research is summarized as follows:

1) Estimate the noised pixels. If the difference between the real center pixel value and the value of the pixel after the order-statistic filtering is above a threshold K_{noise}, the pixel is labeled as a noised pixel and will be processed. The threshold K_{noise} for our implementation is 40 [11].

2) Evaluate the new conduction coefficient using Equation 3.3.

3) Perform the anisotropic diffusion filtering using Equation 3.4.

4) Repeat steps 1 through 3 for 30 iterations.

3.4 Multiple Regression Analysis Theory

Multiple regression analysis is a statistical technique used to estimate the relation between the dependent variable and a set of independent variables [45]. Using multiple regression analysis, we obtain a *regression function* or *predictor equation* that relates the response, y, with a set of independent variables, x_i. A multiple regression model can either be linear with respect to the independent variables or may involve interaction and higher-order terms as shown in Equation 3.7:

$$y = \alpha_0 + \alpha_1 x_1 + \alpha_2 x_2 + \alpha_3 x_3 + \alpha_4 x_1 x_2 + \varepsilon \tag{3.7}$$

In Equation 3.7, α_0 represents the constant term, the coefficients α_i represent the estimates of the model parameters, ε represents the error due to the difference between the actual response and the estimated response, and the term x_1x_2 represents the interaction between independent variables x_1 and x_2, respectively. The *least square method* is the most commonly used estimation criterion. The estimation criterion includes the following two important conditions: 1) The sum of errors must be zero and 2) the sum of the squares of errors is the minimum. As described in [45], the error ε must satisfy the following four conditions for reliable prediction: 1) the mean of the probability distribution (PD) of ε is zero, 2) the variance of PD is constant irrespective of x, 3) the PD of ε is normal and 4) the errors associated with any two observations are independent.

With the aforementioned criterion, we obtain a regression model that best fits the input data for deterministic *Synchronous Iterative Algorithms* (SIAs). Some examples of SIA include: neural network simulations (SNNs), stencil-based image processing (e.g. ADF) and bio-molecular dynamics [46]. To evaluate the validity of the models obtained we use the R-squared and p-values of the regression model and also the p-values of the individual estimates, and visual inspection of the standardized residual plots. Typically, a model is considered reliable if the R-squared value is greater than 0.95 and p-values are less than 0.05. Further details on the regression theory can be found in [45]. In this research, we use the statistical package R [47] to perform all regression analysis.

3.5 Summary

In this chapter, we discussed the Nvidia's Fermi GPGPU architecture and CUDA framework for general purpose graphics computing. We also provided an overview of the four SNN models, the two-level character recognition network for large-scale simulations, and the anisotropic diffusion filter (ADF) for massive images. We also discussed the Multiple Regression Theory, which forms the basis of this research.

CHAPTER 4

EXPERIMENTAL-SETUP, MAPPING AND IMPLEMENTATION

This chapter presents the experimental set-up, SNN-SIA and ADF-SIA mapping methodology, and the GPGPU design space implementations explored in this research. Further, the chapter also provides an overview of the *Synchronous Iterative GPGPU Execution* (SIGE) model [9] and explains the SNN mapping methodology for multi-node GPGPU implementation. The chapter is structured as follows. Section 4.1 describes the Palmetto Cluster [13]. The SNN-ADF mapping used in this study is explained in Section 4.2. Section 4.3 describes the GPGPU design space for the SNN-ADF SIAs. Section 4.4 provides an overview of the SIGE model [9] along with the multi-node GPGPU SNN mapping methodology. The chapter is concluded in Section 4.5 with a summary.

4.1 Palmetto Cluster

We use the GPGPU augmented Palmetto Cluster at Clemson University [13] for the SNN-ADF SIA implementations and GPGPU design space exploratory studies performed in this research. The Palmetto Cluster includes 12 GPGPU HP SL250 servers, with each server connected to two Fermi-based Nvidia Tesla M2075 GPUs via Peripheral Component Interconnect Express (PCI-Ex) bus. Each server is composed of two 2.4 GHz Intel E5-2665 processors with 8 cores each and 64 GB RAM. The servers are connected via Infiniband. For our implementations, we used CUDA 4.2 and MPI version 2.2 on Scientific Linux 6. Additional details on the Palmetto Cluster can be found in [13].

4.2 Network Mapping

This sub-section provides the details of the network mapping for single-node GPGPU SNN and ADF implementations. We explain how the computation tasks are assigned to the CPU core and GPGPU device for optimal performance. In addition, we also discuss the various optimization techniques employed to improve the overall performance of an implementation.

4.2.1 Single-Node GPGPU SNN Mapping

As discussed in Chapter 3, the SNN models are used to implement the two-level character recognition network shown in Figure 3.3. The level-1 neurons act as an input collection layer and the level-2 neurons act as an output collection layer. The total number of neurons in the input level is equal to the total number of pixels in the test image, as each neuron in level-1 corresponds to a pixel in the input image. Therefore, level-1 is the most compute-intensive layer of the network and hence suitable for a GPGPU implementation, whereas the level-2 computations are performed by the CPU-host processor as these computations constitute only 5% of the total computations. The dynamics of a single level-1 neuron is evaluated by a single GPGPU thread. After the GPGPU device finishes level-1 computations, it provides the CPU-host processor with the level-1 firing information in the form of a global firing vector. The host processor uses the global firing vector to evaluate the level-2 neuron current and dynamics.

Several memory-level, instruction-level, and execution configuration-level optimizations were performed for the SNN implementation. These optimizations are explained in Section 3.1.2. To further reduce the data transfers between the CPU-host and GPGPU device, the *block firing vector* concept introduced in [23] was implemented. The block firing vector is implemented in the device shared memory to avoid transferring the global firing vector in each algorithmic time-step. The block firing vector acts as a collection of flags for thread blocks and it is *blocksize* (number of threads in a block) magnitude smaller than the global firing vector. Due to its nominal size, the block firing vector can be transferred from the GPGPU device to the CPU-host in each time-step instead of transferring the entire global firing vector in each time-step. If at any time-step, the block firing vector contains information of a firing event, only then will the entire global firing vector be transferred from the GPGPU device to CPU-host. Figure 4.1 illustrates the concept of the block firing vector.

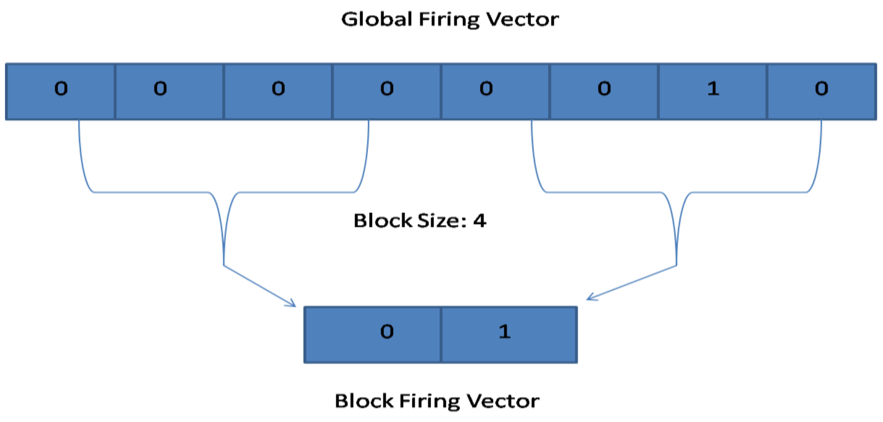

Figure 4.1: Concept of block firing vector

4.2.2 Single-Node GPGPU ADF Mapping

The details of the ADF algorithm are described in Chapter 3. As mentioned in Chapter 3, the highly data-parallel, compute-intensive tasks, namely the median filtering and the partial differential equation evaluation, are performed on the GPGPU device. Whereas, the CPU-host performs the serial computation and data transfer. Two separate GPGPU kernels, namely the *median_kernel* and *PDE_kernel*, are used for the computationally intensive tasks in the ADF algorithm. In each of the GPGPU kernels, a single CUDA thread operates on a single pixel. Therefore, the number of threads created for each kernel is equal to the number of pixels in the input image.

Similar to the SNN implementation, various optimizations including execution configuration optimization, memory optimization, and reduced conditional statements (RCS) were used for the ADF implementation. For the execution configuration optimization, an optimal thread-block configuration was selected to maximize the multiprocessor occupancy, the ratio of the number of warps (a group of 32 concurrent threads) running on the multi-processor to the maximum number of warps that can physically run on the multi-processor. To reduce frequent incoherent accesses to the device global memory, the GPGPU register file was used for pre-fetching the neighboring pixels. Additionally, the conditional statements were replaced with ternary operators to reduce the number of divergent branches. Divergent branches are serialized, thereby impeding the kernel performance [48].

4.3 GPGPU Design Space Implementations for SNN-ADF SIAs

The GPGPU design space exploration aims to analyze the performance of several functionally equivalent implementations of an algorithm, thereby ranking the GPGPU design space. This ranking enables developers to choose the best implementation for optimal algorithm performance on GPGPU-based systems. GPGPUs have a specialized architecture with a memory hierarchy comprising of global, local, shared, constant, and texture memories, each with distinct properties that influence application performance, thereby requiring prudent use of these memories. In our research, we explore the GPGPU design space featuring the above mentioned GPGPU memory hierarchy for optimal application performance. In what follows, we discuss the three GPGPU design space implementations studied in this research.

4.3.1 Implementation 1: Global Memory

For Implementation 1, we use the GPGPU device DRAM that is the *global memory*, to store the entire input data pertaining to an application. The GPGPU device fetches the data from the global memory for computations; once all of the computations are finished, the GPGPU device writes the output back to the global memory for reading by the host processor. The size of the global memory is in the range of Gigabytes, thereby allowing the GPGPU device to access more data for computations. As the global memory is off-chip memory, frequent accesses result in higher memory latency, thereby impeding the overall application performance. All memory accesses for the SNN and ADF implementations use the global memory. We chose a constant thread block configuration of 256 threads per block to maximize the multiprocessor occupancy for the SNN and ADF implementations using the global memory.

4.3.2 Implementation 2: Shared Memory

For Implementation 2, we use *shared memory,* which is an on-chip read/write memory local to a given thread block. All the threads in a thread block have access to the same shared memory, thereby enabling synchronization of the threads within a thread block. Additionally, being an on-chip memory, the use of shared memory reduces the frequent accesses to the off-chip global memory, improving the application performance. For our SNN-ADF SIAs, the size of the shared memory depends on the *blocksize* (number of threads in a block). Therefore, to obtain the kernel runtimes using various *blocksizes*, we vary the *blocksize* parameter in the kernel from 32 threads to 1024 threads. Additionally, for our SNN models, the Implementation 1 is equivalent to Implementation 2 using a *blocksize* of 256, as they have same number of global memory accesses; whereas, for our ADF algorithm, the neighboring pixels in the noised image are fetched from the shared memory.

4.3.3 Implementation 3: Texture Memory

For Implementation 3, we use the *texture memory,* designed for high speed data reading. As described in Chapter 3, texture memory is cached and therefore allows for faster accesses to the data, reducing the frequent high latency accesses to the global memory. The CUDA framework provides techniques for using 1D, 2D, or 3D textures. We use the read-only 1D texture memory to read the level-1 currents for the SNN implementation. For the ADF implementation, we use the read-only 2D texture memory to fetch the neighboring pixels in the noised image.

4.4 Illustration of the SIGE Model

In our research, we study the SIGE model proposed in [9] for Strengths, Weaknesses, and Opportunities (SWO) analysis. The SWO analysis is a sub-set of Strengths, Weaknesses, Opportunities and Threats (SWOT) analysis, as we do not consider Threats for the SIGE model. SWO analysis enables one to identify the positive and negative attributes of a framework, opening avenues for further refinement and improvement. In [9], the authors developed a regression-based framework using the SIGE model for performance analysis of SIAs on the NCSA Forge Cluster [49]. We use the SIGE model and regression-based performance prediction framework to predict the overall execution time of the multi-node GPGPU implementation of the four SNN models on the Palmetto Cluster [13]. In Section 4.4.1, we briefly explain how the single-node GPGPU SNN implementation is extended to a multi-node GPGPU implementation. The detailed description can be found in [9].

4.4.1 Multi-Node GPGPU SNN Mapping

As explained in [9], for the multi-node GPGPU implementation, the MPI ranks were assigned in node-packing fashion. The nodes were configured with a maximum of two MPI processes per node allowing for a 1:1 CPU-core/GPGPU-device ratio at each node thereby reducing the long distance inter-node communication.

The multi-node GPGPU implementation follows the Master-Worker Paradigm as shown in Figure 4.2. The master process, commonly the MPI rank 0, scatters the level-1 neuron inputs to all the other processes. At each MPI process, the level-1 neuron parameters are initialized to constant values that are specific to the SNN model, and hence require no MPI communication. Each CPU-GPGPU pair works as an independent unit where the GPGPU device evaluates the partial level-1 neuron dynamics and the CPU processor evaluates the partial level-2 currents using the firing vector obtained from its corresponding GPGPU device. The partial level-2 currents from each MPI process are then accumulated at MPI rank 0 where the complete level-2 neuron dynamics are evaluated and the character recognition decision is made.

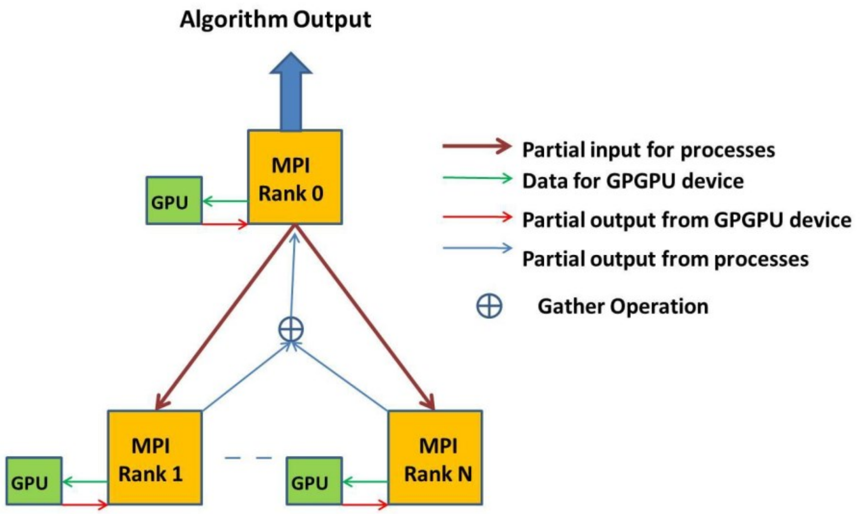

Figure 4.2 Multi-GPGPU Orchestration using Master-Worker Paradigm

4.5 Summary

In this chapter, we discussed the network mapping for the SNN and ADF implementations. Further we explained the design space implementations used in this research. Lastly, we provided a brief overview of the SWO analysis, which will be discussed in detail in Chapter 6, and further explained the SNN mapping for multi-node GPGPU implementations on the Palmetto Cluster.

CHAPTER 5

REGRESSION FRAMEWORK AND PREDICTION MODELS

This chapter presents the regression-based framework used for the GPGPU design space exploration and further demonstrates the use of regression-based framework for runtime prediction of large-scale SNN simulations on the GPGPU augmented Palmetto cluster. The chapter is structured as follows. The use of regression-based framework for the GPGPU design space exploration is illustrated in Section 5.1. Section 5.2 explains the development of regression equations for the multi-node GPGPU implementation of SNN-SIA on the Palmetto Cluster. This study enables a Strength, Weakness, and Opportunity, (SWO) analysis for the SIGE model and regression-based framework, opening further avenues for improvement. The chapter is concluded in Section 5.3 with a summary.

5.1 Performance Prediction Framework

In this sub-section, we explain the regression-based performance prediction framework introduced in [9] that targets Synchronous Iterative Algorithms (SIAs) on GPGPU-based systems. We employ the regression-based framework to predict the kernel execution time of the three SNN and ADF implementations. The low-level design space abstraction is explained in Section 5.1.1. Sections 5.1.2, 5.1.3, and 5.1.4 explain the development of the regression equations for the three GPGPU design space implementations of SNN-ADF SIAs.

5.1.1 Low-Level Design Space Abstraction

As mentioned in Chapter 4, the GPGPU design space constitutes a specialized memory hierarchy comprising of global, local, shared, constant, and texture memories, each with distinct properties that influence the application performance. Similar to low-level abstraction [9], the low-level design space abstraction aims to statistically abstract the characteristics of the system architecture that influence the performance of the aforementioned memories, thereby enabling the kernel runtime prediction using limited implementation details and system information. The regression-based analysis enables the formulation of mathematical models that assist in the kernel runtime prediction for a particular GPGPU architecture with a certain degree of confidence [50]. In our regression-based analysis, the kernel runtime satisfactorily typifies the dependent variable for regression analysis. The choice of independent variables depends on the algorithm studied and the implementation selected from the design space. Additionally, the choice of independent variables can be adjusted by adding or removing parameters based on their statistical significance (contribution to the overall regression model).

The regression-based framework used in this research focuses on the computation component that models the GPGPU device computations using common algorithm characteristics such as the number of floating-point operations and hardware parameters such as the amount of memory accessed for computations as predictor variables. The regression models for the computation component are trained using several small, instrumented executions of an SIA set with a range of computation-to-communication requirements. To perform the regression analysis, we choose a set of nominal test sizes as samples to characterize the behavior of the entire population that includes larger input sizes. These regression models were selected based on their high R^2 values (greater than 0.95) and low p-values (less than 0.05).

5.1.2 Regression Models for Implementation 1:

For Implementation 1, we group the four SNN models either as computation-bound or communication-bound SNN models based on the FLOPS/Byte ratio values mentioned in Table 3.1. As seen in Table 3.1, the HH and ML models have high FLOPS/Bytes ratios, hence they are grouped as computation-bound models, whereas the Izhikevich and Wilson models have low FLOPS/Byte ratios, consequently they are grouped as communication-bound models. Additionally, to obtain prediction models for the algorithms that have FLOPS/Byte ratios between the ML and Wilson models, we present a case where both the models are moderately computation-bound and communication-bound with moderate FLOPS and bytes requirements. The GPGPU kernel regression models are developed separately for the computation-bound, communication-bound, and moderately computation-bound and communication-bound SNN models. These regression models use algorithm characteristics such as the number of floating-point operations, *MFLOPs* (in megaflops) and the number of computational bytes, *MBYTES* (in megabytes) as predictor variables. For each of the SNN models, we perform several instrumented runs of the GPGPU kernel using several network sizes to construct the regression models for the aforementioned bounds. The SNN regression models for all of the aforementioned bounds are shown in Equations 5.1, 5.2, and 5.3.

Computation-Bound:

$$T_{GPU-Kernel} = 20.970927 + 0.029189 * MFLOPS - 0.255117 * MBYTES \qquad (5.1)$$

Communication-Bound:

$$T_{GPU-Kernel} = 2.203181 - 0.035948 * MFLOPS + 0.063823 * MBYTES \qquad (5.2)$$

Moderately Computation- and Communication-Bound:

$$T_{GPU-Kernel} = 2.628 - 0.0005626 * MFLOPS + 0.009957 * MBYTES \qquad (5.3)$$

We now explain the development of the GPGPU kernel runtime regression model for the ADF algorithm. Table 5.1 shows the FLOPS-to-Byte and FLOPS/Byte ratio information per data element for the ADF algorithm and the Izhikevich SNN model. As seen in Table 5.1, both the Izhikevich SNN and ADF algorithms have similar FLOPS-to-Byte requirements with FLOPS/Byte ratio close to 1, therefore we group them together as communication-bound algorithms with a common regression model for the GPGPU device computations, given by Equation 5.4.

Communication-Bound:

$$T_{GPU-Kernel} = -1.20271 + 0.29242 * MFLOPS - 0.24918 * MBYTES \qquad (5.4)$$

Table 5.1 FLOPS/Bytes Ratio for Izhikevich SNN Model and ADF Algorithm

Algorithm	**FLOPS**	**Bytes**	**FLOPS/Byte Ratio**
Izhikevich SNN	13	13	1
ADF	16	12	1.33

5.1.3 Regression Models for Implementation 2:

As mentioned in Chapter 4, shared memory takes the advantage of locality to reduce the frequent accesses to the global memory. Similar to Implementation 1, we group the HH and ML models together as computation-bound models, the Izhikevich and

Wilson models as communication-bound, and ML and Wilson as moderately computation-bound and communication-bound SNN models. Similarly, we also group the ADF and Izhikevich as communication-bound algorithms. As shared memory is allocated per thread block and all threads in the block have access to the same shared memory, we consider the hardware parameter, *BLOCKSIZE* (number of threads in a thread block) as one of the independent variables for developing the GPGPU kernel runtime regression model, in addition to *MFLOPs* and *MBYTES*. For obtaining the regression equations, we considered p-values of the *BLOCKSIZE* up to 0.2 which is in acceptable range. The regression models for all the aforementioned bounds for Implementation 2 are shown in Equations 5.5, 5.6, 5.7, and 5.8.

Computation-Bound: (5.5)

$$T_{GPU-Kernel} = 7.92565 + 0.029034 * MFLOPS - 0.248 * MBYTES + 0.012873 * BLOCKSIZE$$

Communication-Bound: (5.6)

$$T_{GPU-Kernel} = 2.38273 - 0.0403443 * MFLOPS + 0.0710828 * MBYTES + 0.0012492 * BLOCKSIZE$$

Moderately Computation- and Communication-Bound: (5.7)

$$T_{GPU-Kernel} = 2.738 - 5.501e-04 * MFLOPS + 1.050e-02 * MBYTES + 1.491e-03 * BLOCKSIZE$$

Communication-Bound (ADF and Izhikevich): (5.8)

$$T_{GPU-Kernel} = 19.27251 + 0.12649 * MFLOPS - 0.1114 * MBYTES - 0.03378 * BLOCKSIZE$$

5.1.4 Regression Models for Implementation 3:

Texture memory is a fast, read-only cache between the GPGPU Streaming Multiprocessors (SMPs) and device memory that provides high effective bandwidth by reducing memory requests to the off-chip global memory. The four SNN models and ADF algorithm represent a wide-range of computation requirements. The amount of texture memory and global memory accessed therefore varies for each of the four SNN models and ADF algorithm. We model the kernel runtime of the four SNN models and ADF algorithm individually. For the kernel runtime regression equations for the four SNN models and ADF algorithm, a significant collinearity is observed between the predictor variables: global memory (*GLOBAL)* and the texture memory (*TEXTURE)*. To mitigate the collinearity between the predictor variables, we use the texture memory as an indicator variable for developing the kernel runtime regression models. The indicator variables are commonly used to incorporate the categorical effects of variables in the regression analysis. An indicator variable can assume values 0 or 1 to indicate the absence or presence of the categorical effect. The predictor variables used for the kernel runtime regression model are the number of floating-point operations (*MFLOPs)* and the number of bytes accessed from the global memory (*GLOBAL)* as quantitative variables; and the texture memory (*TEXTURE)* as an indicator variable. The regression models for the HH, ML, and Wilson SNN models are shown in Equations 5.9, 5.10, and 5.11.

HH: (5.9)

$$T_{GPU-Kernel} = 46.70 + 0.003125 * MFLOPS + 0.01447 * (GLOBAL * TEXTURE)$$

Morris-Lecar: (5.10)

$$T_{GPU-Kernel} = 2.182 + 8.814e-04 * MFLOPS - 9.629e-03 * GLOBAL + 0.1782 * TEXTURE$$

Wilson: (5.11)

$$T_{GPU-Kernel} = 3.288 + 5.848e-03 * MFLOPS + 5.912e-03 * (GLOBAL * TEXTURE)$$

The Izhikevich model is a sparse-computation SNN model (see Table 3.1). The use of texture memory as quantitative or qualitative variable for accessing the level-1 current information does not contribute significantly to the overall kernel time prediction when compared to the global memory. We observed a p-value of 0.5 that renders the predictor variable (*TEXTURE)* statistically less significant. Therefore, we do not consider the texture memory as predictor variable and use the number of floating-point operations (*MFLOPs)*, and the number of bytes accessed from the global memory, (*GLOBAL)* as predictor variables. Equation 4.12 provides the regression model for Izhikevich model for Implementation 3.

Izhikevich: (5.12)

$$T_{GPU-Kernel} = 1.3184132 + 0.0519219 * MFLOPS - 0.0804248 * GLOBAL$$

For the ADF algorithm, all of the computations involve texture memory accesses as the entire noised image is bound to the texture memory. Therefore, for the ADF algorithm, we consider the texture memory as a quantitative variable for kernel runtime prediction. Equation 5.13 gives the regression model for the ADF algorithm.

ADF: (5.13)

$$T_{GPU-Kernel} = 1.2523 + 0.3536 * MFLOPS - 18.7048 * TEXTURE$$

5.2 Regression-based Framework for multi-node GPGPU implementation

The regression-based framework using the SIGE model proposed in [9] is broken into two primary components: computation and communication. The computation component models the CPU-host and GPGPU device computations using algorithm characteristics such as the number of floating-point operations and computational bytes as predictor variables. Similar to the single-node case, the computation component of the multi-node regression models is trained using several small, instrumented executions of an SIA set with a range of computation-to-communication requirements. The communication component of the regression-based framework is further divided into two sub-components: 1) inter-node communication over the network (Infiniband) and 2) CPU-host/GPGPU-device (host-device) communication over the PCI-Ex bus. The regression models for the communication component are developed using micro-benchmarks that measure transaction throughput and employ data transfer size and processor count as predictor variables. Equation 5.14, adapted from the SIGE model, provides the intermediate equations used for the application runtime prediction on GPGPU-based systems.

$$
\begin{aligned}
&T_{execution-time} = T_{computation} + T_{communication} \\
&T_{computation} = T_{GPU} + T_{CPU} \\
&T_{communication} = T_{inter-node} + T_{PCI-Ex} \qquad (5.14) \\
&T_{inter-node} = T_{scatter} + T_{reduce} \\
&T_{PCI-Ex} = T_{H2D} + T_{D2H}
\end{aligned}
$$

5.2.1 Regression-based Framework for the Computation Component

The computation component of the regression-based framework aims to model the CPU-host and GPGPU device computations. The CPU-host regression model uses the following predictor variables: the number of processors, *P* and the total number of computational bytes, *MBYTES*. The regression models for the CPU-host are elucidated in Equations 5.15, 5.16, 5.17, and 5.18. Similar to the single-node implementation, these regression models were selected based on their high R^2 values (greater than 0.95) and low p-values (less than 0.05).

HH: (5.15)

$$T_{CPU} = -7.781 + 2.344 * P + 5.351e - 04 * MBYTES$$

Morris-Lecar: (5.16)

$$T_{CPU} = 18.66 + 6.25 * P + 0.00217 * MBYTES$$

Wilson: (5.17)

$$T_{CPU} = -13.83 + 5.841 * P + 0.001867 * MBYTES$$

Izhikevich: (5.18)

$$T_{CPU} = -5.6285 + 1.779 * P + 0.0187 * MBYTES$$

The GPGPU computations for the SNN-ADF SIAs significantly depend on the number of floating-point operations (*MFLOPs)* and the number of computational bytes (*MBYTES)* that increase with the problem size. Similar to the single-node implementation, the HH and ML SNN models are grouped into computation-bound

SNNs, whereas the Izhikevich and Wilson models are grouped into communication-bound SNNs. Additionally, the ML and Wilson models are grouped into moderately computation-bound and communication-bound. The regression equations are identical to those in the single-node implementation.

5.2.2 Regression-based Framework for the Communication Component

The communication component of the regression-based framework is broken into the following two sub-components: 1) Inter-node communication over Infiniband and 2) CPU-host/GPGPU-device communication over PCI-Ex bus.

The inter-node communication comprises of the network-level transactions such as scatter, gather, reduce, etc. Each network-level transaction is modeled separately with the message size, *MBYTES* (message size in megabytes) and the number of processors, *P* as predictor variables. Micro-benchmarks were performed on the aforementioned network-level transactions using a typical data-size range (2 KB - 100 MB) to obtain an initial sketch of the transaction throughput. As proposed in [9], we perform separate regression analysis for the network-level transactions at all node configurations due to the irregular behavior of the network-level transactions at various node configurations, as can be seen from Figure 5.1. The log-transformation (log of the problem-size) best fits the graphs shown in Figure 5.1.

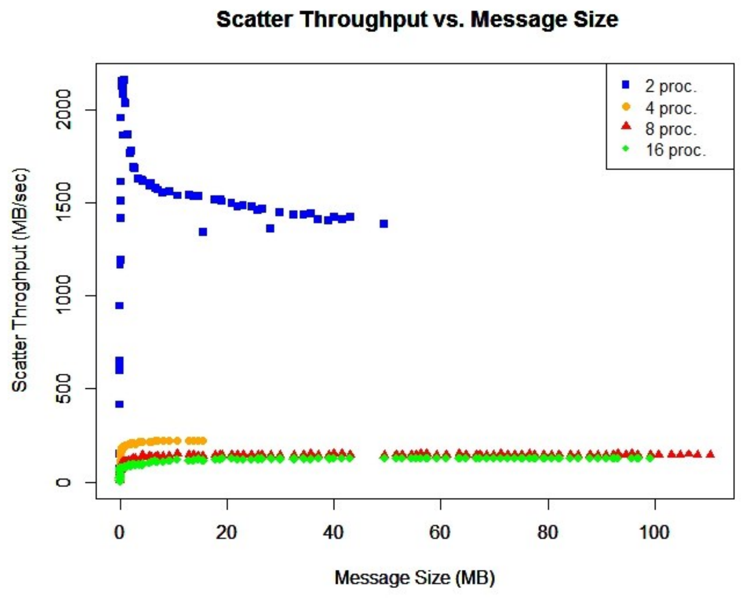

Figure 5.1: Scatter Throughput vs. Message size (megabytes)

For the node configuration of 16, we use Michaelis-Menten kinetics [51] as it provides better performance prediction for the given network sizes. The equation for the Michaelis-Menten kinetics is:

$$v = \frac{V_{\max}[S]}{K_m + [S]} \tag{5.19}$$

where, v represents the reaction rate, V_{max} represents the maximum rate achieved by the system, and K_m represents the substrate concentration at which the reaction rate is half of V_{max} [51]. For the scatter throughput over Infiniband, v and S correspond to the scatter throughput and message size in megabytes, respectively. The terms K_m and V_{max} for the scatter throughput, expressed in megabytes and MB/sec respectively, are obtained by performing non-linear regression analysis on the training dataset. Figure 5.2 shows an example of predicted scatter throughput for 16-node configuration using the Michaelis-

Menten kinetics. The regression models for the inter-node scatter operation are given by Equations 5.20, 5.21, 5.22, and 5.23.

2-Processors:

$$T_{scatter-throughput} = 1402.55 + 75.29 * \log(message-size) \tag{5.20}$$

4-Processors:

$$T_{scatter-throughput} = (223.314 * message-size)/(0.0957 + message-size) \tag{5.21}$$

8-Processors:

$$T_{scatter-throughput} = (149.087 * message-size)/(0.3545 + message-size) \tag{5.22}$$

16-Processors:

$$T_{scatter-throughput} = (130.5677 * message-size)/(0.715 + message-size) \tag{5.23}$$

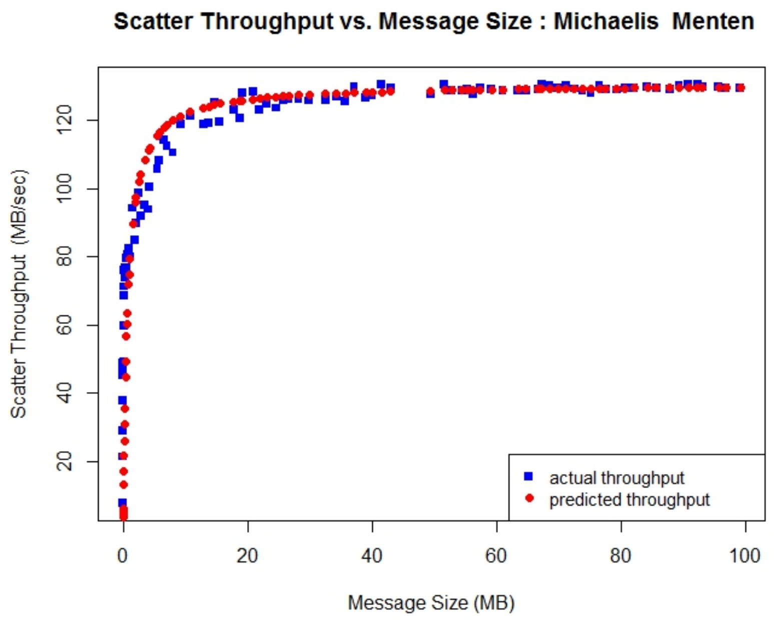

Figure 5.2: Scatter Throughput Prediction for 16-node Configuration using Michaelis-Menten Kinetics

The reduce operation is performed on the level-2 currents in SNN implementations. The size of the level-2 currents is equal to the size of training data set (equal to 48) at all node configurations. Therefore for the reduce operation, as observed in Figure 5.3, the behavior of the transaction is nearly invariant across the SNN network sizes. Therefore, for larger SNN network sizes, we use the average value of the sample space to predict the performance of the reduce transaction at each node configuration.

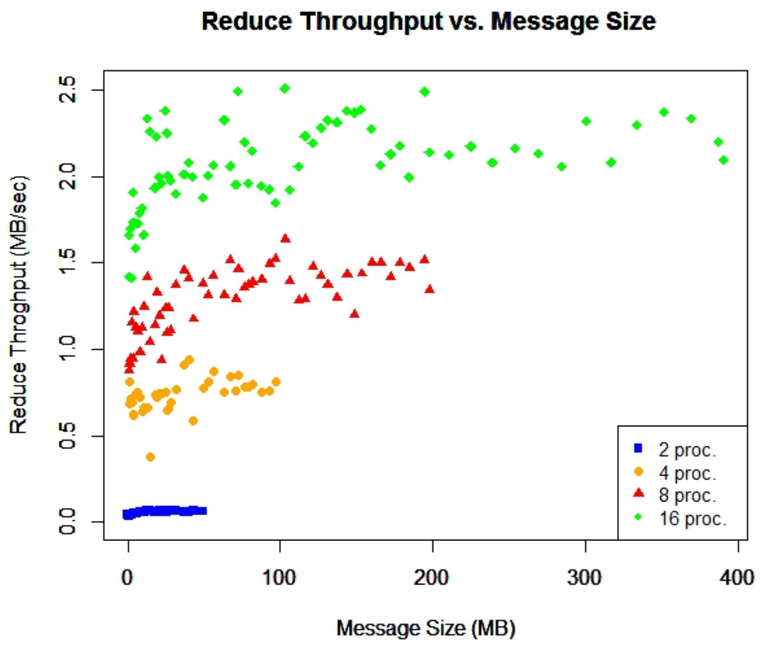

Figure 5.3: Reduce Throughput vs. Message size (megabytes)

For the PCI-Ex bus, we model the host-device transfers using read-back and download throughputs. Figures 5.4, 5.5.a, and 5.5.b show the download and read-back throughput curves for different per-server host-device pair configurations. For the download throughput, the PCI-Ex bus performance resembles the Michaelis-Menten kinetics. There are two data vectors for the SNN implementation that are transferred from the device to host: the *block firing vector* and the *global firing vector*. As the block firing vector is transferred to the host in every time-step of the SNN algorithm; and the global firing vector is transferred only when there is a firing instance, each of these vectors are modeled separately for better analysis. Based on the graphs obtained from the read-back throughput (Figures 5.5.a and 5.5.b), we model the device-to-host transfers either by using Michaelis-Menten kinetics, log transformation method or intuitively fitting a mathematical function for the throughput as a function of message size.

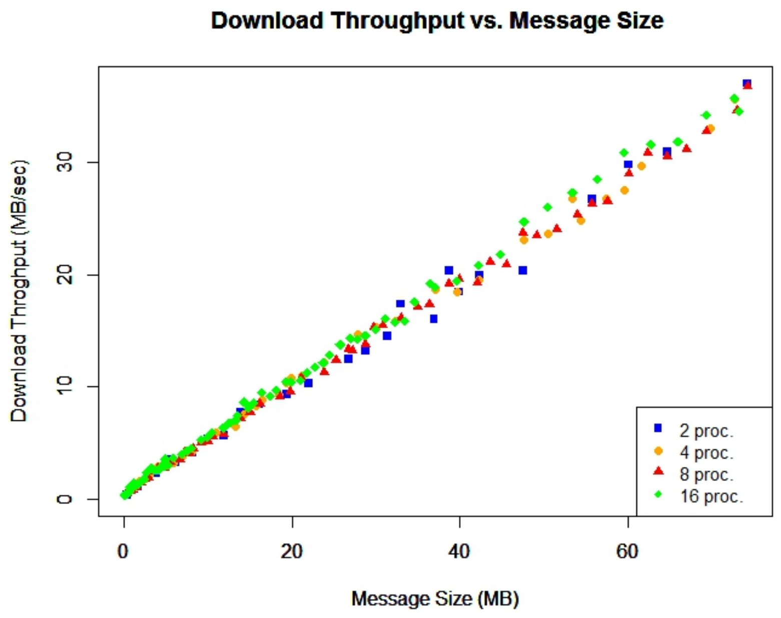

Figure 5.4: Download Throughput vs. Message size (megabytes)

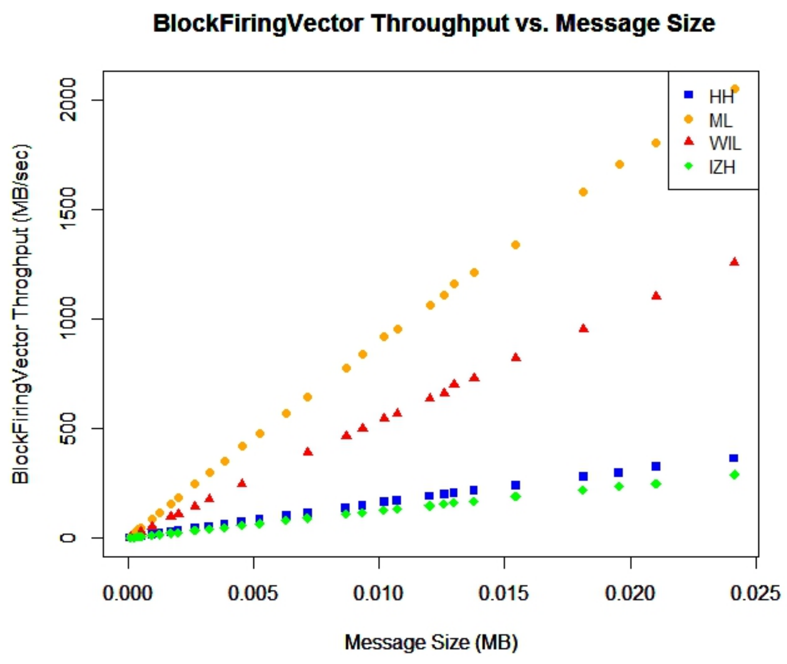

Figure 5.5.a: Block Firing Vector Throughput vs. Message size (megabytes)

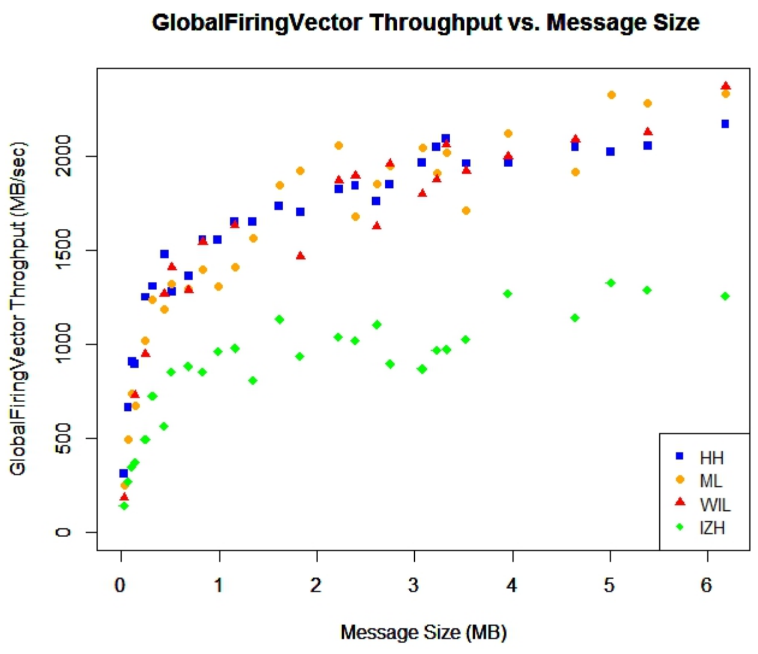

Figure 5.5.b: Global Firing Vector Throughput vs. Message size (megabytes)

The regression models for the host-device transfer times were selected based on high R^2 and low p-values for reliable prediction. Equations 5.24 to 5.28 elucidate the regression models for read-back and download throughputs for the four SNN models.

Download Throughput:

$$T_{download} = (2334.26 * message-size)/(message-size + 0.9444) \tag{5.24}$$

Read-back Throughput:

HH: (5.25)

$$T_{blockfiringvector} = 3.235 + 1.531e + 04 * message-size$$

$$T_{globalfiringvector} = 1569.91 + 320.42 * \log(message-size)$$

Morris-Lecar: (5.26)

$$T_{blockfiringvector} = 17.98 + 85.960e + 03 * message-size$$

$$T_{globalfiringvector} = 1526.14 + 380.66 * \log(message-size)$$

Wilson: (5.27)

$$T_{blockfiringvector} = 7.813 + 52.225e + 03 * message-size$$

$$T_{globalfiringvector} = 1505.26 + 370.54 * \log(message-size)$$

Izhikevich: (5.28)

$$T_{blockfiringvector} = 2.601 + 12.050e + 03 * message-size$$

$$T_{globalfiringvector} = 852.49 + 206.64 * \log(message-size)$$

5.3 Summary

In this chapter, we explained the prediction models obtained for the three design space implementations. Further, we explained the development of the regression models for the multi-node GPGPU augmented Palmetto Cluster, using the SIGE model proposed in [9].

CHAPTER 6

RESULTS AND ANALYSIS

In this chapter, we present the results and analysis for the design space exploratory studies conducted using the regression-based performance prediction framework. As mentioned in Chapter 4, these studies were conducted using SNN-ADF SIAs on the GPGPU augmented Palmetto Cluster. Further, we provide the prediction results for the multi-node GPGPU implementation using the SIGE model [4] and present the SWO analysis. The chapter is structured as follows. Section 6.1 provides the design space exploration results using the SNN models and ADF algorithm. Section 6.2 presents the results for the multi-node GPGPU implementation of the four SNN models. The SWO analysis is presented in Section 6.3. The chapter concludes with a summary provided in Section 6.4.

6.1 Design Space Exploration

In this section, we present the results for the GPGPU design space exploration using the SNN-ADF SIAs as described in Chapter 4. We discuss the kernel runtime values and the prediction error rates for the four SNN models and ADF algorithm. To compare the implementations in the design space, we use the intermediate SNN network sizes: 2400x2400, 3120x3120, and 3600x3600. Similarly, we use the image sizes: 6400x6400, 7680x7680 and 8192x8192 for the ADF algorithm. In addition to the above mentioned test sizes, we also present the results for the largest data size validated using the prediction framework for Implementation 1 and Implementation 3.

6.1.1 Results for Implementation 1

Implementation 1 uses global memory for all data accesses on a single host-device pair. Table 6.1 presents the observed statistical-average kernel runtime values, predicted kernel runtime values, and the prediction error rate obtained using Equations 5.1, 5.2, 5.3, and 5.4 for the four SNN models and the ADF algorithm. For the compute-intensive HH model, the prediction framework gives an error rate of 4.05% for the largest test data size, with the overall prediction error rates below 5% for all the other test data sizes. Similarly, the ML and Wilson models give an error rate of 2.4% and 1.79% for the largest test data size respectively and the overall prediction rates are below 5% for all other test data sizes. For the Izhikevich model, we observe a prediction error rate of 6.09% for the largest data size; whereas the overall prediction error rate is below 10%. The prediction model for the ADF algorithm gives error rates below 5% for all tested image sizes. Additionally, for the ADF algorithm, we validated the prediction model for image size as large as 12800x12800 (156 mega-pixels) and observed a 0.7% prediction error rate.

Table 6.1 Observed and Predicted Values for Implementation 1

Algorithms	Test Data Size	Observed Time (ms)	Predicted Time (ms)	Error Rate (%)
HH	2400x2400	1040.63	1063.615	-2.20879
	3120x3120	1729.446	1783.039	-3.09889
	3600x3600	2274.846	2366.92	-4.04748
ML	2400x2400	30.87975	29.97072	2.943764
	3120x3120	50.44851	48.8372	3.193953
	3600x3600	65.76706	64.14913	2.460092
Wilson	2400x2400	79.60812	83.24969	-4.57437
	3120x3120	135.5998	138.8786	-2.41803
	3600x3600	180.7912	184.0268	-1.78971
Izhikevich	2400x2400	14.4879	14.14669	2.35509
	3120x3120	23.71966	22.38772	5.615374
	3600x3600	30.96388	29.07608	6.096778
ADF	6400x6400	901.936035	910.6197	-0.96278
	7680x7680	1410.561279	1423.52	-0.91868
	8192x8192	1664.515503	1619.815	2.685503
	12800x12800	3927.704	3956.36	-0.72957

6.1.2 Blocksize Scaling Analysis

Prior to presenting the results for Implementation 2, we first study the performance of an algorithm with varying *blocksizes* (number of threads in a block) since it influences the shared memory performance. The best performing *blocksize* is then used for the rest of the analysis. We present the results for the intermediate *blocksizes*: 32, 64, 128, 256, 512, 768, and 1024. It should be noted that when deriving the regression equations, a larger set of *blocksizes* was used to obtain statistically significant prediction equations. For the SNN models, the execution configuration parameter, grid dimension (number of blocks in a grid), depends on the *blocksize* used. A single grid uses *nbx* blocks, where *nbx* is given by Equation 6.1.

$$nbx = {}^{Ne}\!/_{blocksize} + {}^{Ne}\!/_{blocksize} == 0?0:1 \tag{6.1}$$

where, *Ne* is the total number of input neurons. For larger test data sizes such as 3600x3600 and smaller *blocksizes* such as 32 and 64, the kernel gives the error “invalid configuration argument”, meaning the grid dimension used exceeds the maximum grid dimension (65536) permissible by the Nvidia Tesla M2075 specifications [3]. Therefore, the *blocksize*s 32 and 64 were found to yield incorrect simulation results for larger test sizes and were not considered for regression analysis. We first present the results for the HH SNN model, followed by the ML, Wilson, and Izhikevich models, and finally the ADF algorithm.

6.1.2.1 HH SNN Model

Table 6.2 presents the observed statistical-average kernel runtime values for the intermediate test network sizes for the HH model with varying *blocksizes*. As shown in Table 6.2, we obtain kernel runtime values for the test network size up to 1440x1440. Beyond this network size, the kernel execution fails giving a “segmentation fault”. As explained in Chapter 4, shared memory is used to store the firing information for the two-level SNN network. The authors speculate that network sizes beyond 1440 x 1440 result in insufficient shared memory allocation that results in a kernel failure; this limitation is under further investigation and reserved for future work. As seen from Table 6.2, a *blocksize* of 256 generally performs the best (lowest kernel runtime). Another observation made from the kernel runtime values for the HH model, is that the model performs better with *blocksizes* in the range of 128 to 512 as compared to *blocksize*s above 512. This behavior of the HH model kernel is attributed to the high multiprocessor occupancy achieved for *blocksizes* in the range of 128 to 512, which hide the memory latency thereby improving the performance of the system [2]. Beyond a *blocksize* of 512, the streaming multiprocessors become saturated and assigning more threads only decreases occupancy and hence the overall performance.

Table 6.2 Observed Kernel Runtime Values for HH Model (ms)

Blocksize	**Test Data Sizes**		
	1200x1200	1220x1220	1440x1440
32	498.6	498.131	695.1105
64	324.4827	318.1602	465.1345
128	292.6853	289.8708	408.2941
256	289.3785	290.5333	403.4989
512	297.9401	295.4466	407.5135
768	350.564087	352.7624	513.226929
1024	333.3528	353.4338	474.5684

6.1.2.2 ML SNN Model

Table 6.3 shows the observed statistical-average kernel runtime values for the intermediate network sizes for the ML model with varying *blocksizes*. For the ML model, we also include the kernel runtime values obtained using a *blocksize* of 192 while presenting the results because this *blocksize* generally perform the best as observed from Table 6.3. For the ML SNN model, *blocksizes* between 128 and 768 performs better when compared to higher *blocksizes* due to the high multiprocessor occupancy as explained in Section 6.1.2.1.

Table 6.3 Observed Kernel Runtime Values for ML Model (ms)

Blocksize	**Test Data Sizes**		
	2880x2880	3120x3120	3600X3600
32	NA	NA	NA
64	61.32263	NA	NA
128	48.9747	51.794647	69.444992
192	42.2898	50.01535	66.01018
256	44.75959	50.17936	65.89087
512	42.94045	52.60608	69.61125
768	47.904594	57.151424	73.708122
1024	58.097061	68.81992	90.87206

6.1.2.3 Wilson SNN Model

Table 6.4 provides the observed statistical-average kernel runtime values for the intermediate network sizes for the Wilson model with varying *blocksizes*. A *blocksize* of 256 generally performs better when compared to the other *blocksizes*. Similar to the HH SNN model, a *blocksize* within the range of 128 to 512 performs better when compared to higher *blocksizes* for the Wilson SNN model.

Table 6.4 Observed Kernel Runtime Values for Wilson Model (ms)

BlockSize	**Test Data Sizes**		
	2880x2880	3120x3120	3600X3600
32	NA	NA	NA
64	124.7512	NA	NA
128	117.1114	136.8663	178.0053
256	117.1636	136.3236	180.7753
512	119.8014	139.3572	184.254
768	142.3379	165.1318	219.633
1024	132.3825	155.5816	204.29

6.1.2.4 Izhikevich SNN Model

Table 6.5 presents the observed statistical-average kernel runtime values for the intermediate network sizes for the Izhikevich model with varying block configuration. A *blocksize* of 256 generally performs better when compared to the other *blocksizes*. For the Izhikevich SNN model, *blocksize*s between 128 to 768 yields better performance for the given data sizes.

Table 6.5 Observed Kernel Runtime Values for Izhikevich Model (ms)

BlockSize	**Test Data Sizes**		
	2880x2880	3120x3120	3600X3600
32	NA	NA	NA
64	33.88435	NA	NA
128	22.08983	25.83462	34.12537
256	19.73694	22.20939	31.48434
512	20.83513	24.69626	30.8084
768	21.248667	25.03208	33.272076
1024	26.6039	28.88834	39.70522

6.1.2.5 ADF Algorithm

Table 6.6 provides the observed statistical-average kernel runtime values for the intermediate test sizes of the ADF algorithm. As observed in Table 6.6, the ADF implementation with a *blocksize* of 128 provides the lowest kernel runtime values. Similar to the HH and Wilson SNN models, *blocksizes* from 128 to 512 provide better performance results when compared to higher *blocksizes*.

Table 6.6 Observed Kernel Runtime Values for ADF Algorithm (ms)

BlockSize	**Test Data Sizes**		
	6400x6400	7168x7168	7680x7680
32	1394.071	1740.234	2011.201
64	848.225	1058.855	1205.337
128	655.476	821.038	939.456
256	687.573	865.677	988.824
512	665.657	870.361	996.026
768	770.011	971.868	1155.382
1024	837.172	1096.574	1176.612

6.1.3 Results for Implementation 2

For Implementation 2, we use the best performing *blocksizes* from Section 6.1.2 for each of the four SNN models and the ADF algorithm and present the observed statistical-average kernel runtime values, predicted kernel runtime values, and prediction error rate in Table 6.7. The predicted kernel runtime values are obtained using Equations 5.5, 5.6, 5.7, and 5.8. As discussed in Chapter 5, we use slightly higher p-values (up to 0.2) for developing the regression models for Implementation 2, thereby giving slightly higher error rates when compared to other implementations. The regression models provide predictions with maximum error rate of 17.23% for the HH SNN model. As mentioned in Section 6.1.2.1, the sample data points for modeling the HH model are limited due to the insufficient shared memory resource allocation. Additionally, the regression model for the HH SNN model is grouped with the ML SNN model since both SNN models have similar FLOPS/Bytes ratio. We have considered a larger number of samples for the ML SNN model verses the HH SNN model for developing the regression model equations. Therefore, the prediction model obtained is inherently biased toward the ML SNN model; the prediction framework performs well with error rates below 12%. For the Wilson SNN model, the error rates are below 10% for all *blocksizes*. For the Izhikevich SNN model, the prediction error rates are below 11% for the test data sizes. Additionally, we observe from Table 6.7, the prediction model yielded slightly higher prediction error rates for the ADF algorithm. The reason for this behavior is under investigation and is considered future work for this research. It should be noted that for the ADF algorithm, the prediction model provides satisfactory results (less than 10% error rates) for larger image sizes using larger *blocksizes* such as 1024 as observed from Table 6.7.

Table 6.7 Observed and Predicted Values for Implementation 2

Algorithms	Blocksize	Test Data Size	Observed Time (ms)	Predicted Time (ms)	Error Rate (%)
HH	256	1200x1200	289.3785	330.039	-13.5976
		1320x1320	337.896	396.1389	-17.2369
		1440x1440	403.4989	468.5341	-16.1178
ML	192	2400x2400	30.72581	33.86306	-10.2105
		3120x3120	50.01535	55.14182	-10.2498
		3600x3600	66.01018	72.41154	-9.69754
Wilson	256	2400x2400	81.67606	88.71942	-8.62354
		3120x3120	136.3236	147.7832	-8.40617
		3600x3600	180.7753	195.7191	-8.26651
Izhikevich	256	2400x2400	14.313102	15.87395	-10.905
		3120x3120	23.290216	24.96155	-7.1761
		3600x3600	31.175144	32.33698	-3.72682
ADF	128	6400x6400	604.952	756.9519	-25.1259
		7680x7680	709.022	885.7719	-24.9287
		8192x8192	765.192	954.0465	-24.6807
	1024	6912x6192	905.216	923.7796	-2.05074
		7168x7168	1096.574	994.6306	9.296537
		7680x7680	1176.612	1144.062	2.766435

6.1.3 Results for Implementation 3

Implementation 3 uses the texture memory as discussed in Chapter 4. Table 6.8 presents the observed statistical-average kernel runtime values, predicted kernel runtime values and the error rate obtained using the Equations 5.9, 5.10, 5.11, and 5.13 for the SNN-ADF SIAs. As seen in Table 6.8, the regression models provide good predictions for the tested problem sizes on the single host-device pair with a maximum error rate of 1.9% for the HH SNN model. The prediction error rates are below 5% for all of the SNN models and below 2% for the ADF algorithm. As discussed in Chapter 5, we do not develop a regression equation for the Izhikevich model for Implementation 3 because the texture memory predictor (TEXTURE) does not contribute significantly to the overall kernel time when compared to the global memory. For Implementation 3, the largest data size used to validate the prediction framework for the HH model is 5420x5420 with an observed error rate of 0.9%. For the ML model, the largest data size used to validate the framework is 5420x5420 with error rate 1.557%. Finally, for the ADF algorithm the largest image size used for validation was 15360x15360 with an error rate of 0.0755%.

Table 6.8 Observed and Predicted Values for Implementation 3

Algorithms	Test Data Size	Observed Time (ms)	Predicted Time (ms)	Error Rate (%)
HH	2400x2400	1148.419312	1137.091	0.986438
	3120x3120	1912.344727	1889.461	1.196654
	3600x3600	2513.027588	2500.079	0.51524
	5420x5420	5658.714355	5607.769	0.900296
ML	2400x2400	34.313698	33.88377	1.252921
	3120x3120	55.747215	55.63504	0.201219
	3600x3600	70.594208	73.28824	-3.81623
	5420x5420	165.714279	163.1326	1.557898
Wilson	2400x2400	89.43698	91.80581	-2.6486
	3120x3120	152.4633	152.8831	-0.27535
	3600x3600	203.1309	202.4531	0.33369
Izhikevich	2400x2400	17.2002	NA	NA
	3120x3120	28.88623	NA	NA
	3600x3600	36.8613	NA	NA
ADF	6400x6400	719.63	720.9702	-0.18623
	7680x7680	1126.374	1126.908	-0.04739
	8192x8192	1284.132	1282.31	0.141891
	15360x15360	4516.057	4512.644	0.075578

6.1.4 Design Space Exploration: Comparing Implementations

Sections 6.1.1, 6.1.3, and 6.1.4 provide the kernel runtime values for the three design space implementations. In this sub-section, we first compare the observed kernel runtime values of the three design space implementations in Table 6.9, followed by the predicted kernel runtime values comparison in Table 6.10. We discuss the comparison results for the four SNN models first and then discuss the results for the ADF algorithm.

As mentioned in Section 6.1.2, we use a *blocksize* of 256 for the HH, Wilson, and Izhikevich SNN models and a *blocksize* of 192 for the ML SNN model, based on the observed kernel runtime values. It should be noted that, for the four SNN models, the design space Implementation 1 and design space Implementation 2 using a *blocksize* of 256 are equivalent since they use the same block configuration and employ equal number of global memory accesses. As observed in Table 6.9, for the given test data sizes, design space Implementation 2 performs best for the HH, ML, and Izhikevich SNN models; whereas Implementation 1 performs slightly better than Implementation 2 for the Wilson SNN model. As mentioned in Chapter 5, Implementation 1 uses the global memory for all the data accesses. Frequent access to the off-chip DRAM global memory results in high memory latency reducing the overall application performance. Similarly, for the four SNN models, the off-chip texture memory is used to read level-1 currents, whereas the remaining data transfers uses the global memory, thereby any performance improvement provided by the use of texture memory is amortized by the global memory accesses executed in the algorithm. On the other hand, the on-chip shared memory has much higher bandwidth and lower latency than the global and texture memory. Therefore, Implementation 2, which uses shared memory, performs better for 3 out of the 4 SNN models when compared to Implementations 1 and 3. As observed from Table 6.9, for the Wilson SNN model, although Implementation 1 performs the best, the difference in the kernel timing when compared to Implementation 2 is nominal. Additionally, Implementation 2 performs better than Implementation 3, due to the use of shared memory as explained earlier.

Table 6.10 shows that the prediction framework for Implementation 1, gives the best prediction results for the four SNN models. Similar to the results obtained from the observed kernel runtime values, the prediction framework for the Wilson SNN model ranks Implementation 1 as the best implementation. Additionally, the global memory implementation is equivalent to Implementation 2 using a *blocksize* of 256, therefore the prediction framework gives the expected results for the HH and Izhikevich SNN models.

For the ML SNN model, we use a *blocksize* of 192 for the observed design space ranking. Although, the prediction framework deviates from the observed design space implementation in this case, the difference in the kernel runtime values of the observed design space Implementation 2 and the predicted design space Implementation 1 is small and gives a prediction error rate below 3% for the tested data sizes. Therefore, the prediction framework maps the appropriate design space implementations and gives expected prediction results for all of the SNN models.

Unlike the SNN implementations, Implementations 1 and 2 for the ADF algorithm are distinct implementations as they use the global memory and shared memory, respectively for fetching the neighboring pixels in an image. Additionally, we use 2D read-only texture memory for fetching the neighboring pixels for Implementation 3. For the ADF algorithm, the cached texture memory takes advantage of 2D spatial locality, thereby performing better than global memory implementation as seen from Table 6.9. When compared to the shared memory implementation, texture memory does not provide a performance gain over shared memory. As explained above, the use of shared memory reduces the memory latency of an application when compared to the global and texture memories, thereby leading to better performance as seen in Table 6.9. As shown in Table 6.10, the regression-based framework predicts that the design space Implementation 2 will perform better than other implementations for the given test image sizes. The prediction framework selects the design space implementation corresponding to the observed design space implementation, thereby giving suitable prediction results for the ADF algorithm.

Table 6.9 Observed Kernel Runtime Values for Three Design Space Implementations

Algorithms	Test Data Size	Implementation 1	Implementation 2	Implementation 3	Best Implementation
HH	1200x1200	289.2258	289.3785	317.5417	Implementation 2
	1320x1320	353.8927	337.896	382.6518	
	1440x1440	404.5053	403.4989	438.265	
ML	2400x2400	30.87975	30.72581	34.313698	Implementation 2
	3120x3120	50.44851	50.01535	55.747215	
	3600x3600	65.76706	66.01018	70.594208	
Wilson	2400x2400	79.60812	81.67606	89.43698	Implementation 1
	3120x3120	135.5998	136.3236	152.4633	
	3600x3600	180.7912	180.7753	203.1309	
Izhikevich	2400x2400	14.4879	14.313102	17.2002	Implementation 2
	3120x3120	23.71966	23.290216	28.88623	
	3600x3600	30.96388	31.175144	36.8613	
ADF	6400x6400	901.93604	604.952	719.63	Implementation 2
	7680x7680	1410.5613	709.022	1126.374	
	8192x8192	1664.5155	765.192	1284.132	

Table 6.10 Predicted Kernel Runtime Values for Three Design Space Implementations

Algorithms	Test Data Size	Implementation 1	Implementation 2	Implementation 3	Best Implementation
HH	1200x1200	281.632	330.039	319.2977	Implementation 1
	1320x1320	336.3708	396.1389	376.5423	
	1440x1440	396.3228	468.5341	439.2407	
ML	2400x2400	29.97072	33.86306	33.88377	Implementation 1
	3120x3120	48.8372	55.14182	55.63504	
	3600x3600	64.14913	72.41154	73.28824	
Wilson	2400x2400	83.24969	88.71942	91.80581	Implementation 1
	3120x3120	138.8786	147.7832	152.8831	
	3600x3600	184.0268	195.7191	202.4531	
Izhikevich	2400x2400	14.14669	15.87395	NA	Implementation 1
	3120x3120	22.38772	24.96155	NA	
	3600x3600	29.07608	32.33698	NA	
ADF	6400x6400	910.6197	756.9519	720.9702	Implementation 2
	7680x7680	1423.52	885.7719	1126.908	
	8192x8192	1619.815	954.0465	1282.31	

6.2 Results for Multi-node GPGPU Implementation

In this section, we present the validation results for the regression-based framework developed using the SIGE model, for the four SNN models studied. We present the application runtime values in terms of computation time and communication time for node configurations varying from 2- to 16-nodes using a set of selected SNN network sizes at each node configuration. First, we discuss the results for the computation component that includes the GPGPU-device time and CPU time. Lastly, we discuss the communication component that includes host-device transfer times and the inter-node communication times.

6.2.1 Computation Component

The computation component of the regression-based framework consists of the GPGPU device runtime and CPU-host runtime. First, we discuss the computationally intensive HH SNN model, followed by the ML, Wilson, and the Izhikevich SNN models.

6.2.1.1 HH SNN Model

Table 6.11 shows the observed and predicted computation times for the HH SNN model computation component with node configurations varying from 2- to 16-nodes using intermediate network sizes. As shown in Equation 5.14, the computation time, $T_{computation}$, is the sum of CPU computation time, T_{CPU} and GPGPU computation time, T_{GPU}. Equations 5.1 and 5.15 give the regression models for the computation component of the HH model. The prediction error rates generally are below 3% for all node configurations. As seen in Table 6.11, the computation component regression models provide good prediction results for the tested node configurations and SNN network sizes with maximum error rate of 1.04%.

Table 6.11 HH: Observed and Predicted Values for Computation Component (ms)

Configuration	**Computation Component**					
	Network Size	**Predicted T_{CPU}**	**Predicted T_{GPU}**	**Predicted $T_{computation}$**	**Observed $T_{computation}$**	**Error in $T_{computation}$ (%)**
2-Node	3360x3360	258.512	2064.553	2323.065	2301.663	0.92983
	3600x3600	297.2188	2366.92	2664.139	2636.65	1.04256
4-Node	4940x4940	284.3376	2163.943	2448.281	2432.95	0.63014
	5040x5040	295.9006	2251.553	2547.453	2536.071	0.44881
8-Node	5040x5040	158.1238	1136.613	1294.737	1296.107	0.105687
	5280x5280	172.472	1245.326	1417.798	1420.925	0.220059
16-Node	5040x5040	103.2994	579.1439	682.4433	680.0791	0.34763
	5200x5200	108.0451	615.1006	723.1457	718.5106	0.6451

6.2.1.2 ML SNN Model

Table 6.12 shows the observed and predicted computation times for the ML SNN model computation component using intermediate network sizes. Similar to the HH model, the computation time, $T_{computation}$, for ML model is the sum of CPU computation time, T_{CPU}, and GPGPU computation time, T_{GPU}. Equations 5.3 and 5.16 give the regression models for the computation component of the ML model. The prediction error rates generally are below 6% for all the node configurations. As seen in Table 6.12, the computation component regression models provide good prediction results for the tested node configurations and SNN network sizes with maximum error rate of 0.4%.

Table 6.12 ML: Observed and Predicted Values for Computation Component (ms)

Configuration	**Computation Component**					
	Network Size	**Predicted T_{CPU}**	**Predicted T_{GPU}**	**Predicted $T_{computation}$**	**Observed $T_{computation}$**	**Error in $T_{computation}$ (%)**
2-Node	3360x3360	252.561	57.82797	310.389	300.1474	-3.41219
	3600x3600	290.7598	65.99753	356.7574	341.6671	-4.41667
4-Node	4800x4800	270.8798	58.9548	329.8346	342.792	3.779933
	5040x5040	297.8824	64.72984	362.6122	371.3796	2.360754
8-Node	6960x6960	310.0138	61.84232	372.3235	378.3466	1.591968
	7200x7200	329.9098	65.99753	395.9074	403.858	1.968671
16-Node	9840x9840	362.5165	61.80711	424.3236	429.2847	1.155659
	10080x10080	376.1824	64.72984	440.9123	448.6839	1.732099

6.2.1.3 Wilson SNN Model

Table 6.13 shows the observed and the predicted computation times for the Wilson SNN model computation component. We use Equations 5.3 and 5.17 to predict the computation component of the ML model. The prediction error rates are generally below 5% for all node configurations. As seen in Table 6.13, the computation component regression models have a maximum error rate of 0.9%.

Table 6.13 Wilson: Observed and Predicted Values for Computation Component (ms)

Configuration	**Computation Component**					
	Network Size	**Predicted T_{CPU}**	**Predicted T_{GPU}**	**Predicted $T_{computation}$**	**Observed $T_{computation}$**	**Error in $T_{computation}$ (%)**
2-Node	3360x3360	246.7858	158.6492	405.435	393.6342	-2.99792
	3600x3600	283.6179	181.7915	465.4094	452.3376	-2.88983
4-Node	4800x4800	263.5481	161.8412	425.3893	434.3684	2.067164
	5040x5040	289.5845	178.2005	467.785	477.2485	1.982923
8-Node	7140x7140	313.9209	178.8114	492.7323	500.0447	1.462341
	7200x7200	318.6639	181.7915	500.4554	508.0566	1.49614
16-Node	9840x9840	346.4996	169.9211	516.4207	519.2619	0.547178
	10080x10080	359.6765	178.2005	537.877	545.5909	1.413857

6.2.1.4 Izhikevich SNN Model

Table 6.14 shows the observed and the predicted runtimes for the Izhikevich SNN model computation component. Equations 5.2 and 5.18 give the regression models for the computation component of the Izhikevich model. As seen in Table 6.14, the prediction error values for the computation component of the Izhikevich model are higher for the given test data sizes compared to the other SNN models. The Izhikevich model has a relatively short execution time, which consequently results in higher error rates because the slightest deviation results in a larger percentage of the overall runtime [50]. Table

5.14 provides the observed and predicted runtime values along with the overall prediction error rates for all of the node configurations; the maximum error is 6.9%.

Table 6.14 Izhikevich: Observed and Predicted Values for Computation Component (ms)

Configuration	**Computation Component**					
	Network Size	**Predicted T_{CPU}**	**Predicted T_{GPU}**	**Predicted $T_{computation}$**	**Observed $T_{computation}$**	**Error in $T_{computation}$ (%)**
2-Node	3360x3360	230.0167	26.91656	256.9333	240.1918	-6.97005
	3600x3600	264.3562	30.56783	294.924	276.6639	-6.60011
4-Node	4940x4940	252.3274	28.91051	281.2379	288.4484	2.499752
	5040x5040	262.5857	30.00125	292.587	302.591	3.306141
8-Node	6960x6960	257.5645	28.71072	286.2752	289.5344	1.125647
	7200x7200	275.0302	30.56783	305.598	310.064	1.440332
16-Node	10080x10080	283.9337	30.00125	313.935	301.4971	-4.12537
	10120x10120	286.0101	30.22203	316.2321	303.8529	-4.07407

6.2.2 Communication Component

The communication component of the regression-based framework consists of the host-device transfer and inter-node communication times. We use Equations 5.20 to 5.23 to predict the scatter collective runtime and averages of the sample space for predicting the reduce collective runtime as mentioned in Chapter 4. Similarly, Equation 5.24 gives the prediction model for the download throughput for the four SNN models and Equations 5.25, 5.26, 5.27, and 5.28 provide the regression equations for the read-back throughput for the HH, ML, Wilson, and Izhikevich SNN models, respectively. We first present the results for HH model followed by ML, Wilson, and Izhikevich models.

6.2.2.1 HH SNN Model

Table 6.15 provides the inter-node communication times including collective communications such as scatter and reduce operations for the HH model. From Table 6.15, we see that the scatter operation contributes significantly to the overall inter-node communication time versus the reduce operation. The prediction framework has small error for large compute-node configurations with error rates as low as 0.07% for 16-node configuration and a data size of 5200x5200. There are few outliers with an error rate as high as 24% for the largest network size for a 2 compute-node configuration. Table 6.16 shows the PCI-Ex communication times and the error rates for HH model. As seen from Table 6.16, the PCI-Ex component prediction model predicts satisfactorily with error rates less than 8% for all tested node-configurations.

Table 6.15 HH: Observed and Predicted Values for Inter-Node Communication (ms)

Configuration	**Inter-Node Communication Component**					
	Network Size	**Predicted $T_{scatter}$**	**Predicted T_{reduce}**	**Predicted $T_{inter-node}$**	**Observed $T_{inter-node}$**	**Error in $T_{inter-node}$ (%)**
2-Node	3360x3360	25.54857	1.239811	26.78838	35.38488	24.29428
	3600x3600	29.14918	1.239811	30.38899	39.99505	24.01812
4-Node	4940x4940	417.2961	20.67513	437.9712	465.1806	5.849213
	5040x5040	434.3441	20.67513	455.0193	482.9928	5.7917
8-Node	5280x5280	715.7046	29.03013	744.7347	760.2685	2.043195
	5420x5420	754.034	29.03013	783.0641	796.8723	1.732793
16-Node	5040x5040	747.6164	43.03274	790.6491	789.3775	-0.16109
	5200x5200	795.4843	43.03274	838.517	837.8881	-0.07506

Table 6.16 HH: Observed and Predicted Values for PCI-Ex Communication (ms)

Configuration	**PCI-Ex Communication Component**					
	Network Size	**Predicted T_{H2D}**	**Predicted T_{D2H}**	**Predicted T_{PCI-Ex}**	**Observed T_{PCI-Ex}**	**Error in T_{PCI-Ex} (%)**
2-Node	3360x3360	55.75369	33.07544	88.82913	91.62046	3.04662
	3600x3600	63.9431	33.43218	97.37528	104.2714	6.613663
4-Node	4940x4940	60.2259	33.27224	93.49814	99.95006	6.455145
	5040x5040	62.67233	33.2783	95.95063	99.44494	3.513813
8-Node	5200x5200	33.54659	31.97824	65.52483	69.56421	5.806686
	5420x5420	34.57419	32.03614	66.61033	70.39621	5.377955
16-Node	5040x5040	15.97152	30.65526	46.62678	48.92074	4.689136
	5200x5200	16.97559	30.76396	47.73955	49.70574	3.955662

6.2.2.2 ML SNN Model

Table 6.17 provides the inter-node communication times and error rates for the ML model. The regression models for the inter-node communication components yield satisfactory results with high prediction accuracy at larger compute node-configurations with error rate as low as 0.35% for a data size of 10120x10120 as seen in Table 6.17. The error rate for the inter-node communication times is as high as 15% for largest test size for 2 compute-node configurations. Table 6.18 provides the PCI-Ex communication times and error rates for the ML model. The regression model for PCI-Ex component gives good prediction results (< 15%) for all node-configurations with error rates up to 13% for the 4-node configuration and as low as 3% for the 2-node configuration.

Table 6.17 ML: Observed and Predicted Values for Inter-Node Communication (ms)

Configuration	**Inter-Node Communication Component**					
	Network Size	**Predicted $T_{scatter}$**	**Predicted T_{reduce}**	**Predicted $T_{inter-node}$**	**Observed $T_{inter-node}$**	**Error in $T_{inter-node}$ (%)**
2-Node	3360x3360	23.83364	0.0631	23.89674	21.00727	-13.7546
	3600x3600	25.54857	0.0631	25.61167	22.26549	-15.0285
4-Node	4940x4940	417.2961	0.955101	418.2512	437.202	4.334563
	5040x5040	434.3441	0.955101	435.2992	456.1608	4.573299
8-Node	7140x7140	1306.796	1.375033	1308.171	1321.042	0.974299
	7200x7200	1328.811	1.375033	1330.186	1338.526	0.623055
16-Node	10080x10080	2974.034	2.334394	2976.368	2988.011	0.389645
	10120x10120	2997.641	2.334394	2999.975	3010.69	0.355876

Table 6.18 ML: Observed and Predicted Values for PCI-Ex Communication (ms)

Configuration	**PCI-Ex Communication Component**					
	Network Size	**Predicted T_{H2D}**	**Predicted T_{D2H}**	**Predicted T_{PCI-Ex}**	**Observed T_{PCI-Ex}**	**Error in T_{PCI-Ex} (%)**
2-Node	3360x3360	37.30399	3.80944	41.11343	42.62878	3.554751
	3600x3600	42.7636	4.111294	46.87489	50.41451	7.021023
4-Node	4800x4800	38.05704	3.851448	41.90849	48.24216	13.12891
	5040x5040	41.91642	4.064848	45.98127	51.87661	11.36417
8-Node	7140x7140	42.06055	4.07276	46.13331	50.51756	8.678658
	7200x7200	42.7636	4.111294	46.87489	49.76978	5.816544
16-Node	10080x10080	41.91642	4.064848	45.98127	51.04411	9.918568
	10120x10120	42.24653	4.082962	46.32949	50.7329	8.679594

6.2.2.3 Wilson SNN Model

Table 6.19 provides the inter-node communication times and error rates for the Wilson model. From Table 6.19, we observe that the regression models for the inter-node communication components yielded higher error values with few outliers using the 2 compute-node configuration but predicts satisfactorily for larger compute-node configurations. As seen from Table 6.19, the prediction error rate is as low as 0.2% for the largest data size validated on the 16-node configuration. Table 6.20 shows the PCI-Ex communication times. The prediction model for PCI-Ex component predicts satisfactorily with error rates less than 10% for the 2-node, 4-node, and 8-node configurations and below 15% for the data sizes using the 16-node configuration.

Table 6.19 Wilson: Observed and Predicted Values for Inter-Node Communication (ms)

Configuration	**Inter-Node Communication Component**					
	Network Size	**Predicted $T_{scatter}$**	**Predicted T_{reduce}**	**Predicted $T_{inter\text{-}node}$**	**Observed $T_{inter\text{-}node}$**	**Error in $T_{inter\text{-}node}$ (%)**
2-Node	3360x3360	25.54857	0.132766	25.68134	20.4826	-25.3812
	3600x3600	29.14918	0.132766	29.28195	23.81837	-22.9385
4-Node	4940x4940	417.2961	1.646746	418.9428	437.439	4.228272
	5040x5040	434.3441	1.646746	435.9908	456.8487	4.565591
8-Node	7140x7140	1306.796	2.464926	1309.261	1315.881	0.503067
	7200x7200	1328.811	2.464926	1331.276	1347.637	1.214064
16-Node	10080x10080	2974.034	4.228	2978.262	3000.041	0.72595
	10120x10120	2997.641	4.228	3001.869	3010.218	0.277349

Table 6.20 Wilson: Observed and Predicted Values for PCI-Ex Communication (ms)

Configuration	**PCI-Ex Communication Component**					
	Network Size	**Predicted T_{H2D}**	**Predicted T_{D2H}**	**Predicted $T_{PCI\text{-}Ex}$**	**Observed $T_{PCI\text{-}Ex}$**	**Error in $T_{PCI\text{-}Ex}$ (%)**
2-Node	3360x3360	55.75369	4.999086	60.75278	63.90211	4.928365
	3600x3600	63.9431	5.305225	69.24833	73.26241	5.47905
4-Node	4800x4800	56.88327	5.041662	61.92493	66.64434	7.081485
	5040x5040	62.67233	5.25809	67.93042	70.45292	3.580409
8-Node	7140x7140	62.88854	5.266119	68.15466	75.3673	9.569992
	7200x7200	63.9431	5.305225	69.24833	75.67147	8.488201
16-Node	9840x9840	59.7425	5.148916	64.89142	74.21862	12.5672
	10080x10080	62.67233	5.25809	67.93042	75.44286	9.957788

6.2.2.4 Izhikevich SNN Model

Table 6.21 provides the inter-node communication times and Table 6.22 provides the PCI-EX communication times and error rates for the Izhikevich model. From Table

6.21, we see that the regression models for the inter-node communication components yielded higher error values (approximately 20%) for the 2 compute-node configuration but predicts satisfactorily for larger compute-node configurations (prediction error rates below 5%). The PCI-Ex prediction model gives good prediction results with error rates less than 16%.

Table 6.21 Izhikevich: Observed and Predicted Values for Inter-Node Communication (ms)

Configuration	**Inter-Node Communication Component**					
	Network Size	**Predicted $T_{scatter}$**	**Predicted T_{reduce}**	**Predicted $T_{inter-node}$**	**Observed $T_{inter-node}$**	**Error in $T_{inter-node}$ (%)**
2-Node	3240x3240	23.83364	0.044721	23.87836	21.86058	-9.23023
	3600x3600	29.14918	0.044721	29.1939	24.25903	-20.3424
4-Node	4940x4940	417.2961	0.670245	417.9663	442.0012	5.437726
	5040x5040	434.3441	0.670245	435.0143	462.2796	5.897995
8-Node	7140x7140	1306.796	0.988747	1307.785	1302.538	-0.40277
	7200x7200	1328.811	0.988747	1329.8	1337.555	0.579795
16-Node	10080x10080	2974.034	1.892673	2975.927	2993.83	0.598012
	10120x10120	2997.641	1.892673	2999.534	3017.063	0.580995

Table 6.22 Izhikevich: Observed and Predicted Values for PCI-Ex Communication (ms)

Configuration	**PCI-Ex Communication Component**					
	Network Size	**Predicted T_{H2D}**	**Predicted T_{D2H}**	**Predicted T_{PCI-Ex}**	**Observed T_{PCI-Ex}**	**Error in T_{PCI-Ex} (%)**
2-Node	3360x3360	28.07914	5.389269	33.46841	37.54222	10.85127
	3600x3600	32.17384	5.93472	38.10856	45.19944	15.68798
4-Node	4940x4940	30.31524	5.705096	36.02034	39.51047	8.833439
	5040x5040	31.53846	5.870386	37.40885	42.95592	12.91342
8-Node	6840x6840	29.07634	5.53677	34.61311	37.82094	8.481622
	6960x6960	30.09119	5.674724	35.76591	39.64856	9.792662
16-Node	10080x10080	31.53846	5.870386	37.40885	42.89471	12.78913
	10120x10120	31.78604	5.903736	37.68978	41.85537	9.952361

6.3 SWO Analysis of the Regression Framework based on SIGE Model

In this sub-section, we use the results from Section 6.2 and perform the Strengths, Weaknesses, and Opportunities (SWO) analysis of the SIGE model proposed in [4]. As mentioned in Chapter 4, the SWO analysis enables one to study a framework, discussing its strengths and weaknesses for further improvements. We first provide the predicted overall runtime, observed runtime, and overall error rate for the HH, ML, Wilson and Izhikevich models in Tables 6.23, 6.24, 6.25 and 6.26, respectively using Equation 5.14 and further perform the SWO analysis.

Table 6.23 HH: Observed and Predicted Values for Total Execution Time (ms)

Configuration	$T_{execution-time}=T_{computation}+T_{communication}$			
	Network Size	**Predicted** $T_{execution-time}$	**Observed** $T_{execution-time}$	**Error in** $T_{execution-time}$ **(%)**
2-Node	3360x3360	2377.918	2375.387	-0.10657
	3600x3600	2722.044	2688.843	-1.23476
4-Node	4940x4940	2979.75	2998.08	0.61139
	5040x5040	3098.522	3118.508	0.64088
8-Node	5200x5200	2163.648	2218.991	2.494091
	5280x5280	2227.842	2251.59	1.054721
16-Node	5040x5040	1519.719	1518.377	-0.08836
	5200x5200	1609.402	1606.104	-0.20533

Table 6.24 ML: Observed and Predicted Values for Total Execution Time (ms)

Configuration	$T_{execution-time}=T_{computation}+T_{communication}$			
	Network Size	**Predicted** $T_{execution-time}$	**Observed** $T_{execution-time}$	**Error in** $T_{execution-time}$ **(%)**
2-Node	3360x3360	377.1141	363.7835	-3.66445
	3600x3600	432.8445	414.3471	-4.46424
4-Node	4800x4800	766.701	803.5798	4.589314
	5040x5040	843.8928	879.417	4.039525
8-Node	6960x6960	1659.5	1678.35	1.123139
	7200x7200	1772.969	1792.154	1.070493
16-Node	10080x10080	3463.262	3487.739	0.701802
	10120x10120	3490.021	3722.942	6.256389

Table 6.25 Wilson: Observed and Predicted Values for Total Execution Time (ms)

Configuration	$T_{execution\text{-}time}=T_{computation}+T_{communication}$			
	Network Size	**Predicted** $T_{execution\text{-}time}$	**Observed** $T_{execution\text{-}time}$	**Error in** $T_{execution\text{-}time}$ **(%)**
2-Node	3360x3360	491.8691	478.0189	-2.89743
	3600x3600	563.9396	549.4184	-2.64303
4-Node	4800x4800	882.9638	914.9215	3.492943
	5040x5040	971.7063	1004.55	3.269501
8-Node	7140x7140	1870.148	1891.293	1.117991
	7200x7200	1900.98	1931.365	1.573249
16-Node	9840x9840	3419.897	3444.12	0.703318
	10080x10080	3584.07	3621.075	1.021934

Table 6.26 Izhikevich: Observed and Predicted Values for Total Execution Time (ms)

Configuration	$T_{execution\text{-}time}=T_{computation}+T_{communication}$			
	Network Size	**Predicted** $T_{execution\text{-}time}$	**Observed** $T_{execution\text{-}time}$	**Error in** $T_{execution\text{-}time}$ **(%)**
2-Node	3360x3360	315.995	296.9556	-6.41154
	3600x3600	362.2265	346.1223	-4.65272
4-Node	4940x4940	735.2246	769.96	4.511331
	5040x5040	765.0101	807.8265	5.300197
8-Node	6960x6960	1564.886	1574.407	0.604699
	7200x7200	1673.528	1692.217	1.104404
16-Node	10080x10080	3327.271	3338.222	0.328058
	10120x10120	3353.455	3362.771	0.277022

Strengths – In [4], the authors proposed the SIGE model for developing the regression-based framework for predicting runtimes of Synchronous Iterative Algorithms (SIAs) on multi-node GPGPU systems. The authors used the Forge GPGPU cluster at the National Center for Super-Computing Applications (NCSA) [47] that consists of Fermi-based Tesla M2070 GPGPUs for implementing the SNN SIA. For our research, we use the GPGPU nodes in the Palmetto Cluster where each GPGPU-enabled-node consists of 2 Fermi-based Tesla M2075 [20]. From Tables 6.23 – 6.26, we observe that the prediction framework developed using the SIGE model gives good prediction results with very low error rates and few outliers. The HH model yields a prediction error rate below 3% for all test data sizes and all node configurations. The ML model provides an overall prediction error rate below 5%. The Wilson model also yields a prediction error rate below 5%. The Izhikevich model gives error rates up to 7% for the given test data sizes and all node configurations. The regression-based framework developed using the SIGE model is deemed satisfactory for runtime prediction for other clusters consisting of other GPGPU architectures, thereby enabling application to architecture mapping. The regression-based framework enables runtime prediction for SIAs without actual large-scale implementations; therefore the framework can be used for obtaining runtime values for larger-node configurations and larger data sizes.

Weaknesses – The regression-based framework is broken into two components: computation and communication. Although this component division provides sufficient insight into the algorithm performance, the behavior of the individual components may vary across computing systems. Although, the regression-based framework provides satisfactory prediction results for the scatter component, we observed a few outliers as seen in Tables 6.15, 6.17, 6.19, and 6.21. These outliers are attributed to the missing predictor variables in the regression equations, for instance, network protocol changes.

Opportunities – Considering the weaknesses mentioned above, other predictor variables, in addition to the ones used in this research, could be employed to obtain better prediction results. Other GPGPUs such as the AMD's Radeon or NVIDIA's Kepler should be explored to further validate the accuracy of the SIGE model.

6.4 Summary

In this chapter, we explained the prediction results for the three design space implementations explored in this research. We also discussed the *blocksize* scaling results for design space Implementation 2, used to obtain the best results for Implementation 2. We then explored the three design space implementations based on their runtime performance. Further we provided the results for illustrating the SIGE model. Lastly, we performed the SWO analysis of the SIGE model based on the results obtained from the SIGE model illustration on the GPGPU nodes of the Palmetto Cluster.

CHAPTER 7

CONCLUSIONS AND FUTURE WORK

In this chapter, we summarize the research presented in this thesis and provide conclusions based on the results obtained. We also suggest additional ideas for future work and thorough exploration of the GPGPU design space.

7.1 Summary

We presented the use of the regression-based performance prediction framework for exploring the GPGPU design space featuring a memory hierarchy using the Synchronous Iterative GPGPU Execution (SIGE) model. Additionally, we illustrated the functionality of the SIGE model on the GPGPU nodes of the Palmetto Cluster and performed a Strengths, Weaknesses, and Opportunities (SWO) analysis. In Chapter 1, we presented the motivation for our research. We discussed the role of heterogeneous systems, such as GPGPU-based clusters, in High Performance Computing. Several factors including inefficient application mapping, load-balancing, and tuning in the heterogeneous systems lead to poor application speed-up and overall performance degradation. To optimize the application performance, developers use performance prediction models, thereby predicting scalability and application runtime prior to large-scale implementations. Earlier prediction models employ architecture-specific tuning, rendering them inadequate for performance modeling on current architectures. The research presented in this thesis aims to explore the GPGPU design space featuring a memory hierarchy, thereby allowing a developer to analyze and predict the algorithm performance with the given level of system abstraction.

Chapter 2 provides the literature review of some of the prominent design space exploratory studies on GPGPU-based systems. Although these techniques provide satisfactory prediction results, they require thorough architecture knowledge for accurate prediction. Additionally, the techniques employed by the previous models for prediction is not trivial, thereby making the modeling task difficult. To overcome the limitations of the aforementioned models, we employ a performance prediction framework using the SIGE model that uses easily available application parameters for straightforward performance modeling, thereby enabling developers to map their application to the appropriate architecture. In addition, the chapter also introduces the architecture studies conducted using the SNNs and ADF.

Chapter 3 presented Nvidia's GPGPU architecture along with the CUDA framework used in this research. Further, the chapter presented background on the case studies used in this research: Spiking Neural Networks (SNNs) and non-linear Anisotropic Diffusion Filter (ADF). Additionally, the chapter provided an overview of the multiple regression analysis theory that is used in this research for obtaining the predictor equations.

Chapter 4 provided an overview of the Palmetto Cluster used in this research. The chapter also presented details of the network mapping for single-node GPGPU SNN and ADF implementations on the GPGPU-augmented Palmetto Cluster. The chapter also explains the various GPGPU design space implementations explored. Finally, the mapping methodology for the multi-node GPGPU implementation of the SNNs was presented to illustrate the SIGE model.

In Chapter 5, we explained the regression-based performance prediction framework used for the GPGPU design space exploration. We introduced the low-level design space abstraction that enables kernel runtime prediction with limited implementation details. Further, we described the development of the regression models for the three design space implementations explored in this research. The prediction framework uses algorithm parameters such as the number of floating-point operations, and the number of computational bytes, along with hardware parameters such as number of threads in a block to develop the regression models. The chapter also provided details on the regression-based framework for the multi-node GPGPU SNN implementation. The regression-based framework includes the computation and communication components that are modeled using algorithm parameters as mentioned earlier along with the number of processors and message size.

In Chapter 6, we presented results for the GPGPU design space explored in this research using SNN-ADF SIAs as case studies on the GPGPU-enabled nodes of the Palmetto Cluster. The *blocksize* (number of threads in a block) scaling analysis investigates the behavior of an algorithm while varying the block configuration. The results show that the *blocksize* of 256 performs optimally for the HH, Wilson, and Izhikevich models, whereas the *blocksize* of 192 performs the best for the ML model. Similarly, the ADF algorithm performs best with a *blocksize* of 128. It is asserted that high multiprocessor occupancy is observed for the above block configurations, thereby improving the performance of the algorithm. Further, we rank the design space implementations based on the observed and predicted kernel runtime values. For the HH, ML, and Izhikevich model, Implementation 1 performs best according to the predicted kernel runtimes but Implementation 2 is the best observed kernel runtime. For the Wilson model, both the predicted and observed kernel runtime values rank Implementation 1 as the best implementation. Similarly, for the ADF algorithm both the predicted and observed kernel runtime values rank the design space Implementation 2 as the best implementation.

Chapter 6 also provided an illustration of the use of SIGE model on the Palmetto Cluster. The chapter presented the results for the computation and communication components of the regression-based framework using the four SNN models. The regression models for the computation component predicted the runtimes with high accuracy (less than 7% error rate) for the larger data sizes for all node configurations. For a given data size and node configuration, the Izhikevich SNN model observed slightly higher error rates when compared to the other SNN models due to the short execution times as explained in the chapter. The regression models for the communication component provided slightly higher error rates (approximately 12% for the HH SNN model) due to the error-prone scatter runtime prediction values, but in general predicted well for larger node configurations. Considering the overall runtime, the prediction framework provides good prediction results (below 10% error) barring a few outlier test cases. Further, we use this study to perform a SWO analysis of the regression framework based on the SIGE model. The SIGE model provides a prediction framework that is not tightly-coupled with a specific cluster as validated in our research. In [4], the authors used the Forge GPGPU cluster at the National Center for Super-Computing Applications, whereas our research uses the Palmetto Cluster at the Clemson University and provides good prediction results. Additionally, the regression-based framework can be used for obtaining runtime values for larger-node configurations and larger data sizes without actual large-scale implementations. One of the weaknesses of the regression-based framework reported was, although the framework enables satisfactory algorithm performance analysis, the behavior of the individual components may vary across computing systems resulting in larger prediction errors. We suggested including other predictor variables to overcome the aforementioned weakness. Furthermore, we also proposed the use of other GPGPU architectures to validate the accuracy of the SIGE model and the regression framework.

7.2 Conclusions

From the above summary we draw the following conclusions:

1. The *blocksize* scaling analysis enables the study of algorithm behavior with varying block configurations. The multiprocessor occupancy contributes significantly to the kernel runtime performance, thereby affecting the overall application performance. As seen in these studies, selected *blocksizes* give best kernel runtime performance for the four SNN models and the ADF algorithm because a high multiprocessor occupancy is achieved at that *blocksize*. Additionally, after selecting the appropriate *blocksize*, we observe a drop in performance, as the streaming multiprocessors are saturated, which

results in performance degradation. Additionally, *blocksizes* below a particular range for each algorithm mentioned in Chapter 6, give low occupancy, which interferes with the ability of the SMPs to hide memory latency and therefore results in poor application performance.

2. From the design space exploration results based on the observed kernel runtime, we conclude that the shared memory implementation performs best in most of the case studies used in this research. Shared memory is an on-chip memory that provides faster data access when compared to the off-chip global and texture memory, thereby providing much higher bandwidth and lower latency leading to an overall performance gain. Additionally, between the texture memory and the global memory implementation, the performance varies depending on how data is accessed from the texture memory and how many global memory accesses are executed in an algorithm. For the four SNN models, the global memory implementation performs better than texture memory because the 1D texture memory is used to read level-1 currents only and the remaining data transfers use global memory. Whereas, in case of the ADF algorithm, texture memory takes advantage of the 2D spatial locality, thereby giving better performance than the global memory ADF implementation.

3. The predicted kernel runtime ranks the global memory implementation as the best implementation for the four SNN models and the shared memory implementation as the best implementation for the ADF algorithm. Regression analysis depends on many factors including but not limited to, the training data set used for developing the regression models, the R^2 value, and the p-value that defines the significance of a

predictor variable. Global memory uses a more stable training data set when compared to the shared and texture memory implementations, thereby making the p-values for each predictor variable below 0.05. As mentioned in Chapter 5, the shared memory implementation uses predictor variables with slightly higher p-values (up to 0.2), thus giving slightly higher prediction errors (but still below 25% error). Additionally, the global memory implementation and shared memory implementation using a *blocksize* of 256, are equivalent as they employ an equal number of global memory accesses. Therefore, our prediction framework appropriately ranks the design space implementations for 4 out of 5 case studies. Although there is a deviation in the predicted and observed design space ranking for the 5th case study, the observed and predicted kernel runtime for design space implementation 2 differ only marginally, giving an error rate below 3%. Therefore, our prediction framework ranks the best design space implementation for an application as expected for 4 out 5 cases and acceptable results in the 5th case.

4. With SWO analysis, we are able to analyze the strengths and weaknesses of the regression framework based on the SIGE model and provide suggestions for how it can be further explored for better performance prediction and application to architecture mapping.

7.3 Future Work

The research in this thesis assists developers in choosing the best implementation from the GPGPU design space for their application prior to actual large-scale implementation. This work can further be extended by enlarging the GPGPU design space under consideration. Other GPGPU memories such as the local and constant memory can be included in the design space exploration along with the effect of cache. As mentioned in Chapter 7, a shared memory implementation presents some limitations on prediction modeling. Specifically, for the HH SNN model, the sample points obtained were few and this resulted in the regression model yielding high prediction error. Similarly, the shared memory implementation for the ADF algorithm does not give high prediction accuracy for selected block configurations. This issue can be thoroughly investigated in future studies. Additionally, the shared memory implementation in this research uses the same number of global memory accesses as that of the global memory implementation. The shared memory implementation can be further explored with reduced accesses to the global memory to study the changes in the performance of the application. As discussed in the SWO analysis, other predictor variables, such as network protocols can be studied to further explore the communication component performance modeling. In addition, other GPGPU architectures such as AMD's Radeon and Nvidia's Kepler can be used to analyze the behavior of the SIGE model and further validate the modeling framework.

REFERENCES

1. Intel Teraflop Research Chip Overview. http://download.intel.com/pressroom/kits/Teraflops/Teraflops_Research_Chip_Overview.pdf

2. Nvidia's Next Generation CUDA Compute architecture: Fermi. http://www.nvidia.com/content/PDF/fermi_white_papers/NVIDIAFermiComputeArchitectureWhitepaper.pdf

3. Nvidia CUDA Programming Guide v2.5.0. http://developer.download.nvidia.com/GPU_Programming_Guide/GPU_Programming_Guide.pdf

4. Nvidia's Next Generation CUDA Compute Architecture: Kepler GK110 - Whitepaper http://www.nvidia.com/content/PDF/kepler/NVIDIA-Kepler-GK110-Architecture-Whitepaper.pdf

5. W. Jia, K.A. Shaw, M. Martonosi. Stargazer: Automated Regression-based GPU Design Space Exploration. In Proceedings of the IEEE International Symposium on Performance Analysis of Systems and Software, pp. 2-13, 2012

6. D. Schaa, D. Kaeli. Exploring the Multiple-GPU Design Space. In Proceedings of International Symposium on Parallel and Distributed Processing, pp. 1-12, 2009

7. K. Kothapalli, R. Mukherjee, M.S. Rehman, S. Patidar, P.J. Narayanan, K. Srinathan. A Performance Prediction Model for the CUDA GPGPU Platform. In Proceedings of the IEEE International Conference for High Performance Computing, pp. 463-472, 2009

8. B.J. Barnes, B. Rountree, D.K. Lowenthal, J. Reeves, B.D. Supinski, M. Schulz. A Regression-Based Approach to Scalability Prediction. In Proceedings of the 22nd Annual International Conference on Supercomputing, pp. 368-377, 2008

9. V.K. Pallipuram, M.C. Smith, N. Raut, X. Ren. A Regression-Based Heterogeneous Performance Prediction Framework for GPGPU Clusters. In Proceedings of Concurrency and Computation: Practice and Experience, pp. 27, 2012

10. M.A. Bhuiyan, T.M. Taha, R. Jalasutram. Character Recognition with Two Spiking Neural Network Models on Multicore Architecture. Symposium on Computational Intelligence for Multimedia Signal and Vision Processing, pp. 29-34, 2009

11. S. Naik. Connecting Architecture, Fitness, Optimizations and Performance using an Anisotropic Diffusion Filter. Master's Thesis, Clemson University, 2012

12. J. Weickert. Theoretical Foundations of Anisotropic Diffusion in Image Processing. In Proceedings of the 7th Theoretical Foundations of Computer Vision, 1996.

13. Palmetto Cluster User Guide http://desktop2petascale.org/resources/175/download/Palmetto.Cluster.Users.Guide.pdf

14. L. G. Valiant. A Bridging Model for Parallel Computation. Communications of the ACM, vol. 33(8), pp. 103-111, 1990

15. S. Fortune, J. Wyllie. Parallelism in Random Access Machines. In Proceedings of the 10th annual ACM symposium on Theory of computing, pp. 114–118, 1978

16. P. B. Gibbons, Y. Matias, V. Ramachandran. The Queue-Read Queue-Write PRAM Model: Accounting for Contention in Parallel Algorithms. SIAM Journal on Computing, vol. 28(2), pp. 733-769, 1999

17. Y. Zhang, J.D. Owens. A Quantitative Performance Analysis Model for GPU Architectures. In Proceedings of the 17th International Symposium on High Performance Computer Architecture, pp. 382-393, 2011

18. S. Hong, H. Kim. An Analytical Model for a GPU Architecture with Memory-Level and Thread-Level Parallelism Awareness. In Proceedings of the 36th International Symposium on Computer Architecture, vol. 37(3), pp. 152-163, 2009

19. S.S. Baghsorkhi, M. Delahaye, S.J. Patel, W.D. Gropp, W.W. Hwu. An Adaptive Performance Modeling Tool for GPU Architectures. In Proceedings of the 15th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, vol. 45(5), pp. 105-114, 2011

20. P. Guo, L. Wang. Accurate CUDA Performance Modeling for Sparse Matrix-Vector Multiplication. In Proceedings of the International Conference on High Performance Computing and Simulation, pp. 496-502, 2012

21. M.A. Bhuiyan, M.C. Smith, V.K. Pallipuram. Performance, Optimization, and Fitness: Connecting Applications to Accelerators. Concurrency and Computation: Practice and Experience, vol. 23(10), pp. 1066-1100, 2010

22. M.A. Bhuiyan. Performance Analysis and Fitness of GPGPU and Multi-Core Architectures for Scientific Applications. Ph.D. Dissertation, Clemson University, 2011

23. V.K. Pallipuram. Acceleration of Spiking Neural Networks on Single-GPU and Multi-GPU Systems. Master's Thesis, Clemson University, 2010

24. V.K. Pallipuram, M.A. Bhuiyan, M.C. Smith. A Comparative Study of GPU Programming Models and Architectures Using Neural Networks. Journal of Super Computing, vol. 61(3), pp. 673-718, 2011

25. B. Han, T.M. Taha. Neuromorphic Models on GPGPU Cluster. In Proceedings of the International Joint Conference on Neural Networks, pp. 1-8, 2010

26. A.C. Sobieranski, L. Coser, M.A.R. Dantas, A. Wangenheim, E. Comunello. An Anisotropic Diffusion Filtering Implementation to Execute in Parallel Distributed Systems. In Proceedings of the 11th International Conference on Computational Science and Engineering Workshops, pp. 182-187, 2008
27. K. Datta, S. William, V. Volkov, J. Carter, L. Oliker, J. Shalf, K. Yelick. Auto-tuning the 27-point Stencil for Multicore. The 4th International Workshop on Automatic Performance Tuning, 2009

28. White Paper; Intel Next Generation Microarchitecture (Nehalem) http://www.intel.com/pressroom/archive/reference/whitepaper_nehalem.pdf

29. V.K. Pallipuram, N. Raut, X. Ren, M.C. Smith, S. Naik. A Multi-Node GPGPU Implementation of Non-Linear Anisotropic Diffusion Filter. In Proceedings of the Symposium on Application Accelerators for High-Performance Computing, pp. 11-18, 2012

30. S. Philip, B. Summa, V. Pascucci, P.T. Bremer. Hybrid CPU-GPU Solver for Gradient Domain Processing of Massive Images. In Proceedings of the 17th International Conference of Parallel and Distributed Systems, pp. 244-251, 2011

31. Chris McClanahan. History and Evolution of GPU Architecture. A Paper Survey http://mcclanahoochie.com/blog/wp-content/uploads/2011/03/gpu-hist-paper.pdf

32. Technical Brief: NVIDIA GeForce 8800 GPU Architecture. http://www.nvidia.com/page/8800_tech_briefs.html

33. Nvidia's Tesla M2075 - GPU Computing Processor. http://www.nvidia.com/docs/IO/105880/DS-Tesla-M-Class-Aug11.pdf

34. A. Gupta, L. Long. Character Recognition Using Spiking Neural Networks. In Proceedings of the International Joint Conference on Neural Networks, pp. 53-58, 2007

35. A. L. Hodgkin, A. F. Huxley. A Quantitative Description of Membrane Current and Application to Conduction and Excitation in Nerve. Journal of Physiology, vol. 117, pp. 500-544, 1952

36. E. M. Izhikevich. Simple Model to Use for Cortical Spiking Neurons. IEEE transactions on Neural Networks, vol. 14(5), pp. 1569-1572, 2003

37. C. Morris, H. Lecar. Voltage Oscillations in the Barnacle Giant Muscle Fiber, Biophysical Journal, vol. 35, pp. 193-213, 1981

38. H. R. Wilson. Simplified Dynamics of Human and Mammalian Neocortical Neurons. Journal of Theoretical Biology, vol. 200(4), pp. 375-388, 1999

39. D. Tsai, W. Chiu, W. Li. Anisotropic Diffusion-based Detail-preserving Smoothing for Image Restoration. In Proceedings of 17th IEEE International Conference on Image Processing, pp. 4145-4148, 2010

40. K. Hildebrandt, K. Polthier. Anisotropic filtering of Non-linear Surface Features. Computer Graphics Forum, vol. 23(3), pp. 391-400, 2004

41. J. Weickert. Anisotropic diffusion in Image Processing. B.G. Teubner, Stuttgart, 1998

42. A.C. Sobieranski, L. Coser, MAR. Dantas, A. Wangheim, E. Comunello. An Anisotropic Diffusion Filtering Implementation to Execute in Parallel Distributed Systems. In Proceedings of the 11th International Conference on Computational Science and Engineering Workshops, 2008

43. W. Wu, H. Liu. Noise Removal using Non-linear Diffusion Filtering based on Statistic-local Open System. Congress on Image and Signal Processing, vol. 3, pp. 372-378, 2008

44. P. Perona, J. Malik. Scale Space and Edge Detection using Anisotropic Diffusion. IEEE Transactions on Pattern Analysis and Machine Intelligence, vol. 12, pp. 629-639, 1990

45. W. Mendenhall, T. Sincich. A Second Course in Statistics: Regression Analysis. 6th Edition, Pearson Education, 2003

46. S.S. Hampton, S.R. Alam, P.S. Crozier, P.K. Agarwal. Optimal Utilization of Heterogeneous Resources for Biomolecular Simulations. In Proceedings of International Conference for High Performance Computing, Networking, Storage and Analysis, pp. 1-11, 2010

47. The R manual.
http://cran.r-project.org/manuals.html

48. T.D. Han, T.S. Abdelrehman. Reducing Branch Divergence in GPU Programs. In Proceedings of the 4th Workshop on General Purpose Processing on Graphical Processing Units, 2011

49. Dell Nvidia Linux Cluster Forge.
http://www.ncsa.illinois.edu/UserInfo/Resources/Hardware/DellNVIDIACluster/

50. D.G. Kleinbaum, L.L. Kupper, K.E. Muller, A. Nizam. Applied Regression Analysis and Other Multivariable Methods. 3rd Edition, Duxbury Press, 1998

51. L. Michaelis, M.L. Menten. The Kinetics of Invertase Action. Biochem. Z, vol. 49, pp. 333-369, 1913
Translated by Roger S. Goody, Kenneth A. Johnson
http://path.upmc.edu/divisions/chp/PDF/Michaelis-Menten_Kinetik.pdf