12-2011

PERFORMANCE ANALYSIS AND FITNESS OF GPGPU AND MULTICORE ARCHITECTURES FOR SCIENTIFIC APPLICATIONS

Mohammad Bhuiyan
Clemson University, mbhuiya@g.clemson.edu

PERFORMANCE ANALYSIS AND FITNESS OF GPGPU AND MULTICORE ARCHITECTURES FOR SCIENTIFIC APPLICATIONS

A Dissertation
Presented to
the Graduate School of
Clemson University

In Partial Fulfillment
of the Requirements for the Degree
Doctor of Philosophy
Computer Engineering

---

by
Mohammad Ashraf Uddin Bhuiyan
December 2011

Accepted by:
Dr. Melissa C. Smith, Committee Chair
Dr. Walter B. Ligon, III
Dr. Haying (Helen) Shen
Dr. Jan Medlock
Dr. James von Oehsen

ABSTRACT

Recent trends in computing architecture development have focused on exploiting task- and data-level parallelism from applications. Major hardware vendors are experimenting with novel parallel architectures, such as the Many Integrated Core (MIC) from Intel that integrates 50 or more x86 processors on a single chip, the Accelerated Processing Unit from AMD that integrates a multicore x86 processor with a graphical processing unit (GPU), and many other initiatives from other hardware vendors that are underway.

Therefore, various types of architectures are available to developers for accelerating an application. A performance model that predicts the suitability of the architecture for accelerating an application would be very helpful prior to implementation. Thus, in this research, a *Fitness* model that ranks the potential performance of accelerators for an application is proposed. Then the *Fitness* model is extended using statistical multiple regression to model both the runtime performance of accelerators and the impact of programming models on accelerator performance with high degree of accuracy. We have validated both performance models for all the case studies. The error rate of these models, calculated using the experimental performance data, is tolerable in the high-performance computing field.

In this research, to develop and validate the two performance models we have also analyzed the performance of several multicore CPUs and GPGPU architectures and the corresponding programming models using multiple case studies. The first case study used

in this research is a matrix-matrix multiplication algorithm. By varying the size of the matrix from a small size to a very large size, the performance of the multicore and GPGPU architectures are studied. The second case study used in this research is a biological spiking neural network (SNN), implemented with four neuron models that have varying requirements for communication and computation making them useful for performance analysis of the hardware platforms. We report and analyze the performance variation of the four popular accelerators (Intel Xeon, AMD Opteron, Nvidia Fermi, and IBM PS3) and four advanced CPU architectures (Intel 32 core, AMD 32 core, IBM 16 core, and SUN 32 core) with problem size (matrix and network size) scaling, available optimization techniques and execution configuration. This thorough analysis provides insight regarding how the performance of an accelerator is affected by problem size, optimization techniques, and accelerator configuration.

We have analyzed the performance impact of four popular multicore parallel programming models, POSIX-threading, Open Multi-Processing (OpenMP), Open Computing Language (OpenCL), and Concurrency Runtime on an Intel i7 multicore architecture; and, two GPGPU programming models, Compute Unified Device Architecture (CUDA) and OpenCL, on a NVIDIA GPGPU. With the broad study conducted using a wide range of application complexity, multiple optimizations, and varying problem size, it was found that according to their achievable performance, the programming models for the x86 processor cannot be ranked across all applications, whereas the programming models for GPGPU can be ranked conclusively. We also have

qualitatively and quantitatively ranked all the six programming models in terms of their perceived programming effort.

The results and analysis in this research indicate and are supported by the proposed performance models that for a given hardware system, the best performance for an application is obtained with a proper match of programming model and architecture.

DEDICATION

This dissertation is dedicated to my wife, Selina, for her support and understanding in the course of my doctoral research, and my son, Ian, for the peace and joy in my life.

ACKNOWLEDGMENTS

I would like to acknowledge my advisor, Melissa C. Smith, for her focused guidance toward the goals of this research. Without her inspiration and guidance I could not complete this dissertation. I would also like to thank to other members of my committee, Walter B. Ligon III, Haying (Helen) Shen, Jan Medlock, and James von Oehsen, who played very important roles for completing my thesis. Their comments and suggestions certainly improved my research. I would also like to thank my colleagues in our research group, specially, Vivek Pallipuram and Scott Gibson for their constructive suggestions towards my research goal. I would also like to thank Arctic Region Supercomputing Center (ARSC), Nvidia, and Cyberinfrastructure Technology Integration (CITI) group of Clemson University for their help in use of the computing nodes. Finally, this research was supported in part by the National Science Foundation (NSF) under grant No. CCF-0916387.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

Chapter 1

Introduction

Recent developments in multicore processors and General-Purpose Graphical Processing Units (GPU) mainly focus on exploiting task- and thread-level parallelism from applications. In this research, we analyze various aspects of the performance of these leading architectures including one of the most advanced GPGPUs from NVIDIA and multicore processors from Intel, AMD, IBM, and SUN. This research investigates multiple architectures, the largest being a 32-core multicore (Intel and AMD) and 512-core GPGPU (Nvidia Fermi). These processors offer various types and levels of parallelism through its parallel architecture. Thus, to exploit the offered parallelism, various parallel programming models available to exploit the parallelism including POSIX-threading, OpenMP, OpenCL, and Concurrency Runtime for multicore processors as well as CUDA and OpenCL for GPGPUs. These programming models are not only different in terms of the programming effort required by the developers, but the performance of applications developed with these programming models also varies for a given combination of application and architecture.

The first case study used in this research is a matrix-matrix multiplication of square size matrix. The problem size, i.e., matrix size is varied to collect performance data of four architectures.The second case study used in this research is a biological spiking

neural network (SNN), implemented with the Izhikevich, Wilson, Morris-Lecar, and Hodgkin-Huxley neuron models. The four SNN models have varying requirements for communication and computation making them useful for performance analysis of the hardware platforms. Of these, the Hodgkin-Huxley (HH) model is the most compute intensive, while the Izhikevich model is the most computationally efficient.

We report and analyze the performance variations with network (problem size) scaling, available optimization techniques and execution configuration. Based on the performance analysis of various architectures, a *Fitness* performance model, that predicts the suitability of the single node architecture for accelerating an application is proposed and verified with the matrix-matrix multiplication and SNN implementation results. The Fitness model is extended to model the performance of the accelerator more accurately by utilizing statistical regression. The effect of programming models, such as POSIX threading, OpenMP, Concurrency Runtime, OpenCL, and CUDA on the performance of accelerators is demonstrated with the implementation of the SNN and modeled using statistical regression. The Roofline model, an existing performance model, has also been used in this research to verify the hardware bottleneck(s) and attainable peak performance of the architectures.

Our parallel implementations were successful in attaining application speed-up for the most computationally intensive HH model as high as 976x for CUDA and 878x for OpenCL. For the communication intensive Izhikevich model 17x speedup was achieved for CUDA and 12x for OpenCL on Nvidia's Tesla C2050 over a serial implementation on an Intel Core 2 Quad 2.66GHz system using all compiler optimizations. Using an

Intel i7 multicore platform, a speedup of 27x was achieved for the HH model with OpenCL and 28x for the Izhikevich model with p-threading compared to the same serial implementation mentioned above. Application speed-up values were found to be dependent on the communication and computation requirements of each model and the way they were mapped onto the architecture utilizing the parallelization and optimizations techniques offered by the programming models. The thorough performance analysis produced the *Fitness* model and extension with multiple regression introduced in Chapter 7 which conclude that a proper match of architecture and programming model with algorithm complexity provides the best performance.

1.1 Application Accelerators

The history of relying on the principles of Moore's Law for increasing performance from a single core processor is no longer plausible due to physical limiting factors including the memory and clock walls. The performance/power ratio is another important limiting factor, as it is one of the most important considerations for building future supercomputers. To overcome these limitations, the industry trends have moved to multicore or many-core processors, increasing the number of processing cores per processing chip (CPU). For the last couple of years, 4 to 6 core processors were available from major vendors such as Intel, IBM, AMD, and Oracle-SUN. Currently these companies are presenting prototypes of up to 80 cores in each CPU. Theoretically, more than 1 Tera FLOPs can be achieved from this processor [1]. The current state-of-the-art

is a 100-core processor available for general-purpose and high-performance computing [2].

In addition to multicore CPUs, GPGPUs have been very effective and popular in high-performance computing for the last four years. This paradigm uses the CPU and GPGPU together in a heterogeneous co-processing computing model. In this model, the serial part of the application is run on the CPU whereas the compute intensive parallel part is run on the GPGPU so that the parallel portion is accelerated exploiting the massively parallel architecture of the GPGPU. More than 1 tera FLOPs peak performance is claimed for NVIDIA's and AMD/ATI's advanced GPGPUs [3][4].

The current and future generations of multicore processors and GPGPUs are discussed in the following sub-sections.

1.1.1 Multicore Processors

The number of cores per processor has been increasing during the last decade. The most popular multicore architectures used in high-performance computing and supercomputers are Intel Xeon, AMD Opteron, IBM Cell, and SUN T2+ UltraSparc. These processors predominantly have 4 to 8 cores. At the SuperComputing 2010 (SC10) Conference [5], IBM presented the "heptadecacore" BlueGene/Q processor, which has 18 cores per CPU; though only 16 cores are available for application acceleration. At 1.6 GHz clock speed, the processor would manage 205 GFLOPs. This year Intel plans to announce the next-generation Itanium processor, code-named Poulson, the follow-on processor to Tukwila [6]. According to [6], the Poulson processor is a "32nm, 3.1 billion

transistor, 12-Wide-Issue Itanium processor for mission-critical servers." The processor has 8 cores (each supports multi-threading), a ring-based system interface, and the combined cache on die is 50MB. The communication link supports 128 GB/s of bandwidth between the processors and 45 GB/s of memory bandwidth. AMD revealed its latest multicore processor, the 16-core Interlagos at SC10. Interlagos executes 64 double-precision floating-point operations per clock, for 224 Gflops at 3.5 GHz [7]. This is a competitive performance with Intel's planned 8-core Poulson processor, which will achieve similar theoretical peak flops value [7].

1.1.2 GPGPU Processors

General Purpose Graphical Processing Units (GPGPUs) have attracted attention from researchers in the HPC community for accelerating numerous data parallel algorithms. The introduction of CUDA by the NVIDIA Corporation in November 2006 [8] has completely changed the face of the graphical computing. Several research groups have explored NVIDIA GPGPUs with CUDA to accelerate large data-parallel algorithms such as the Smith Waterman algorithm [9], NAMD [10], LAMMPS [11], OpenMM [12] and others of significant concern in the biological and physical sciences communities.

1.2Programming models

Given the growing interest of researchers in multicore and GPGPUs, several programming models have emerged, establishing their niche in the HPC community. While a few multicore models like POSIX-threading [81] and OpenMP [82] have existed for several years, newer programming models have emerged with their own strengths,

such as Open Computing Language (OpenCL) [84], Concurrency Runtime [83], and Thread Building Block (TBB) [117]. These programming models are not only different in terms of the programming effort required, applications performance also varies significantly for a given combination of architecture and application. Thus a thorough study is required to investigate application performance for a correlation of architectures with programming models.

While there are many programming models available for multicore processors as discussed earlier, only two programming models are currently available for GPGPU: CUDA and OpenCL. While CUDA coupled with NVIDIA's GPGPUs has the major share of the HPC GPGPU market, the Open Computing Language (OpenCL) [84] is a growing standard that allows for portability across architectures. Unlike CUDA, which is solely dedicated for computing with NVIDIA GPGPUs, OpenCL was conceived to support a variety of architectures such as GPGPUs from different vendors, DSPs, Cell processors and multicore architectures.

1.3 Motivation And Organization

Scientific application developers need the capability to estimate and understand architecture performance to minimize development time and produce an optimized implementation. Benchmark results across various accelerators are one of the indicators used to determine if an accelerator is suitable for an application. But application characteristics can vary widely in terms of computation, communication, patterns of

instructions. Thus a specific benchmark result alone is not adequate for a developer to estimate the potential performance of an accelerator for an application.

Along with the benchmark results, understanding the performance characteristics and limitations is very important for developing parallel software. Thus some case study applications having varying computational and communication requirements can provide insight on various aspects of accelerator performance. The result and the analysis of the study not only aid the developer in performance estimation but can also help with optimization of the application for the target accelerator. Also it will be beneficial for the developers to know what software optimizations have the greatest impact on performance so that they can prioritize their optimization effort.

There are broad ranges of multiprocessor architectures available and new accelerators are continuously evolving. The programming nuances, implementation strategies, hardware bottlenecks, and software optimizations are very different from one architecture to the next. Substantial development time is required to build an optimized and accelerated implementation of an application for each platform. Therefore, implementing an application on all available architectures to determine which performs the best is unreasonable. For this reason, a *Fitness* performance model is proposed, the first contribution of this research, as a useful tool for a developer to determine the top architecture matches for an application from a pool of candidate platforms. To predict the performance of accelerator in a high degree of accuracy, the Fitness model is extended with multiple statistical regression. The multiple regression model for the accelerators and programming models is able to predict the actual runtime performance.

We have applied both of the proposed performance models to the implementations of SNNs and matrix-matrix multiplications on the hardware architectures to project the best match and the actual performance.

The second contribution of this research is a rigorous study of several leading accelerators, and using case studies, various aspects of accelerator performance are investigated. As the leading accelerators, we have studied Intel multicore, AMD multicore, IBM PowerXCell 8i and PS3, SUN T2 UltraSparc, NVIDIA Tesla C870, and Fermi, and AMD/ATI Radeon. Additionally some experimental prototypes, such as AMD 32 core and the Intel 32 accelerator are also studied. As the first case study, a single precision matrix-matrix multiplication algorithm is used to study the accelerator performance over a range of problem sizes. For the second case study, we have selected four computation models used in Spiking Neural Networks, namely Hodgkin-Huxley, Morris-Lecar, Wilosn, and Izhikevich model. Of these, the Hodgkin-Huxley (HH) model is the most compute intensive, while the Izhikevich model is the most computationally efficient.

In summary, the contributions of this research are:

1) Introduction of a *Fitness* performance model for accelerators and its validation
2) Extension of the *Fitness* model with statistical multiple regression for accelerators and programming models and its validation
3) Performance analysis of several leading accelerators with the variation of problem size, optimization technique, accelerator configuration

4) Investigation and quantification of the effect of programming models on the application performance and programming effort for a given architecture

1.3.1 *Fitness* Performance Model Introduction

There are various types of multiprocessors available having different hardware bottlenecks and software optimizations. Thus, it is very time consuming for a developer to implement an application on all of these multiprocessors to determine the best one for that application and/or problem size. Therefore a performance *Fitness* model is introduced that ranks the performance of architectures for an application so that the best match of architecture and application is determined. The *Fitness* model is validated using the data collected from the performance analysis of architectures.

1.3.2 Extended Fitness Model

To predict the runtime performance in higher accuracy and to include the impact of accelerator configurations and programming models on the architecture performance, a multiple statistical regression model is developed as an extension to the Fitness Model. Multiple regression models are developed for eight accelerators and six programming models. These models are validated using the performance dataset of accelerators and programming models.

1.3.3 Architecture Performance analysis

A scaling study of any application is an important aspect for understanding accelerator performance. In some cases, the performance increases with an increase in

problem size whereas in others, the performance decreases. It is also important to know which accelerators can accommodate larger problems sizes. Thus in this research we investigate these aspects for the set of leading accelerators using the SNN case study. Additionally, optimization techniques may have varying effect on the performance. We investigate and analyze the effect of optimization techniques on the application performance across a set of accelerators.

It is also important to understand how the addition of cores increases or decreases the performance. In this research, the performance as cores are added is thoroughly studied. The recent multicores from all the major vendors support hyper threading techniques meaning more threads than cores can be launched. Using more threads than the number of cores may not always increases performance. Similarly for the GPPGU, threads per block can be controlled by the developer and the ratio has a significant effect on performance. Thus we analyzed the effect of hyper threading for multicore and thread per block on the GPGPU for inclusion in the *Fitness* model.

1.3.4 Programming Model Investigation

There are various programming models that have been developed to exploit the parallel architectures. For example, POSIX-threading, OpenMP, Concurrency Runtime, OpenCL are available for multicore architectures whereas CUDA and OpenCL are available for GPGPUs. It is found in our studies, the performance of an application-architecture pair depends on the programming model used. For example, for a given combination of architecture and application, the resulting performance varies from one

programming model to the next. We investigate and quantify the programming model effect on application performance.

1.4 Method of Study

The effect of problem sizes, several software-level optimizations, accelerator configuration, programming models on the accelerator performance are analyzed. Based on the above performance analysis, we introduce a *Fitness* performance model to predict the theoretical best accelerator, or group of accelerators, for an application. The *Fitness* model is validated using the matrix-matrix multiplication and SNN implementations on the accelerators. Then the *Fitness* model is extended to predict the runtime performance with high degree of accuracy and include the effect of accelerator configurations using statistical regression. Finally, statistical regression is used to model the effect of programming models on the application performance on accelerators.

1.5 Dissertation Organization

In the next chapter, we present an overview of the related work in accelerator performance, programming models, performance modeling, and SNNs. Chapter 3 details the case studies, matrix-matrix multiplications and four SNN models. Chapter 4 explains the experimental systems, which include the accelerators, programming models, implementations, and optimization techniques. Chapter 5 and 6 details the experimental results on the accelerators performance and programming models respectively. A multiple regression model for accelerators and programming models are also developed in this chapter. Chapter 7 develops the *Fitness* model and multiple regression models for

accelerators and programming models. Chapter 8 presents the validation of *Fitness* model and the multiple regression models using the selected case studies. In Chapter 9, we summarize all the results and derived models and describe the implication of this research in the HPC paradigm. The conclusions and suggestions for future work based on the current results are also described in this chapter.

Chapter 2

Background And Related Work

In this chapter we will discuss the trends in high-performance computing related to this research. In recent years, vendors have been releasing more complex and advanced computing architectures that are proving to be increasingly challenging for researchers to use. These challenges, such as methods for exploiting the memory hierarchy are placing more emphasis on performance analysis of new architectures for scientific and commercial applications. These advanced multicore processors and many-core graphic processors have garnered attention from the HPC community because of the theoretical performance but the actual performance for a given application can be dramatically lower. In this chapter, we focus our discussion on the existing body of performance analysis research for multicore processors, many-core graphic processors, and various proposed performance models for these architectures. Additionally, matrix-matrix multiplication algorithm and four SNN models are used as case studies in this research and we will discuss the relevant work on accelerating these case studies.

2.1 Performance Study of Multicore

Recent performance analysis research has been focused on emerging multicore architectures using different case studies. Aparna Chandramowlishwaran, et al. [28] used the Fast Multipole Method (FMM) as a case study to analyze several performance

metrics including speedup on emerging multicore architectures: Intel's Nahelam, AMD's Barcelona, and Sun's Victoria Falls. They consider several performance enhancements including low-level tuning, numerical approximation, data structure transformations, OpenMP parallelization, and algorithmic tuning. The authors report significant speedups, for example, 25x with Intel's Nahelam. Sam Williams, et al. [29] utilize the Lattice Boltzmann Computation algorithm to study the performance of the Intel Xeon E5345 (Clovertown), AMD Opteron 2214 (Santa Rosa), AMD Opteron 2356 (Barcelona), Sun T5140 T2+ (Victoria Falls), and QS20 IBM Cell Blade. They also studied the impact of various memory, instruction, and data structure optimizations on speedup and throughput performance. Kaushik Datta, et al. [30] presents various optimization techniques of stencil computations on the Intel Itanium2, AMD Opteron, and IBM Power5 processors. They were able to achieve 88% of algorithmic peak on the Cell BE and 54% on the cache-based processors. Our research includes a rigorous performance study of several leading advanced multicore processors from Intel, AMD, IBM, and SUN and GPGPUs from NVIDIA and AMD/ATI considering various architecture and software level factors that affect their performance.

2.2 Performance Study of GPGPU

In addition to traditional multicore architectures, several groups are analyzing the performance of GPGPUs to accelerate scientific and commercial applications. Araya-Polo, et al. [31] accelerated the Reverse Time Migration (RTM) algorithm used in advanced seismic imaging techniques on the IBM Cell/BE system (QS22 blade), the NVIDIA Tesla C1060, and an FPGA-based system (SGI RASC RC100 platform having

two Virtex 4 FPGAs). A maximum speedup of 26x was achieved with NVIDIA Tesla over the Intel Harpertown processor. Ali Khajeh-Saeed, et al. [32] accelerated the Smith Waterman algorithm on NVIDIA GPGPUs and discuss the application of optimization and parallelization techniques. Using the SSCA#1 (Bioinformatics) benchmark, they were able to achieve a speedup of 45x and they show linear scaling up to 4 GPGPUs. Nageswaran, et al. [33] implemented a large-scale spiking neural network using the Izhikevich model on a GPGPU. They were able to simulate 100K neurons with 50 million synapses achieving a speedup of 27x. In this research, we use four different SNN neuron models having a range of computational complexities as one of the case studies for performance analysis of the multicore and GPGPU architectures.

2.3 Programming Models

Along with the development of parallel architectures, diversified efforts have been made to develop parallel programming models to exploit the parallelism present in applications and offered by hardware. Programming models play a significant role in software parallelization and many parallel models have been explored over the last few decades.

In 1970, a Parallel Random Access Machine (PRAM) [89] was proposed as a shared memory Single Instruction Multiple Data (SIMD) model. Using PRAM, many parallel algorithms were developed by the computational science community for mathematical problems over the next 20 years. Though it was very successful on a shared memory architecture, it could not be used in real-world distributed systems. In 1989, Valiant proposed the Bulk-Synchronous Parallel [91] style for writing efficient parallel programs

on distributed memory machines. In 1994, the Message Passing Interface (MPI) [90] standards were released; the APIs are callable from C, C++ and FORTRAN. So far, MPI has been the most successful in distributed memory clusters. In [93], Parallel FORTRAN Preprocessor (PFP) and Parallel C Preprocessor (PCP) are proposed as new parallel programming models. Contrary to the fork-join models, they allow the programmer to view the entire program as executing in parallel. The processors are split into teams that execute sections of the code in parallel or serial where needed. In [86], the Parallel Cellular Programming model is proposed, where the number of parallel processes in a program is much larger than the number of processors in a machine. In this model, the authors present a computation model where many processes execute on a single processor efficiently. They also present a virtual machine that runs processes according to this model. The programming model is well-suited to the design of massively parallel fine-grained applications, such as automata, partial differential equations, and finite element methods.

The Parallel Phase Model [88], introduced in 2009, is a high-level programming abstraction that exploits the parallelism of many cores on a node and parallelism at the cluster level. The programming abstraction enables those low-level parallel programming tasks to be handled by the compiler and runtime systems. Thus the application developers are relieved from traditional coding difficulties. The programming model uses the virtualization of processors, virtualization of memory, implicit communications, implicit synchronization, automatic data distribution and locality management, and layered parallelisms. The authors implement four applications on a Cray XT4 machine

with a total of 9660 compute nodes, each node having 4 cores (AMD Opteron 2.3GHz Quad Core) and 8GB of physical shared memory. The performance of PPM was shown to be slightly better than MPI. In [92], a programming model is proposed that attempts to maximize data/process locality and balance computational load. The model is based on the observation that irregular neural networks mostly execute local operations, reduction, and broadcasts. The language used in this programming model is fully abstract; the number of processors, data distribution, process distribution, and the execution model are hidden from the user. The compiler can derive most of the information to generate parallel code from the non-annotated code. There are many other programming models that have been proposed by researchers such as [87], [94] but the community does not actively use these.

Attempts have also been made to compare the various parallel programming models. In [95], the authors compared four programming models for multiblock flow computation: data parallel, message passing, work sharing, and explicit shared memory. All four models were implemented on a 1024 processor Cray T3D system. Comparative advantages of each programming model were assessed in terms of ease of use, functionality, and performance. In [96], the authors discuss the benefits of using the two dominant programming models: shared memory and distributed memory, and then the merging of the two models into the Virtual Shared Memory (VSM) programming model. The VSM can be used with physically distributed memory and provide both programmer convenience and high scalability. They compared the shared and distributed memory models for several case studies and emphasize the use of VSM in the case studies. In

[97], performance of a hybrid programming model (MPI+OpenMP) for a cluster of multi-CPU nodes is compared with a pure MPI model. Two metrics, bandwidth and latency, were used for comparing the two models. The pure MPI implementation exhibited higher performance than the hybrid programming model based on the two selected metrics. In [98], MPI and Explicit Multi-threading C (XMTC) are compared in terms of programming effort. One group of subjects (students in one class) implemented sparse-matrix dense-vector multiplication using MPI while a second group of subjects implemented the same application on XMTC. The development time and correctness of the program were compared, and the authors reported that programming in XMTC was more likely to be correct and required 46% less effort (development time).

In [99], the author compared three programming models, Thread Building Block (TBB), OpenMP, and POSIX-threading in terms programming methodology. Performance results from ten different benchmarks were obtained using TBB, and the results were analyzed. In [100], two programming models representative of the distributed memory model (MPI) and the shared memory model (Unified Parallel C (UPC)) were compared in terms of speedup and runtime performance. The performance of MPI did not vary with the optimizations while the performance of UPC fluctuated significantly. Both models reportedly had similar performance but UPC required less programming effort, measured in terms of number of lines of code. In [101], six programming models for IBM cell processors were compared: Sequoia, StarSs, Cell-Gen, Tagged Procedure Calls (TPC), CellMP, and IBM's low-level Cell software developer's kit (SDK). The comparison involves performance and productivity. Three

applications were used in this study, CellStream, FixedGrid, and PBPI where the Cell SDK performed the best in terms of speed up among the six programming models.

There has been a limited amount of work done on systematic comparison of OpenCL and CUDA. In [113], the authors have accelerated an EMRI modeling application using Nvidia's C1060 as one of the accelerators and have achieved similar performance for both CUDA and OpenCL. In [114], the authors used the Adiabatic Quantum Algorithms (AQUA), which are Monte Carlo simulations, to compare CUDA and OpenCL on Nvidia's GTX-260 (with compute capability 1.3). The programming models were compared in terms of data transfer time, kernel execution time and end-to-end runtime. Their results indicated that the CUDA implementations perform consistently better than the OpenCL implementations. In [115], the authors studied the performance portability of OpenCL and concluded that the performance is not portable. They implemented TRSM and GEMM for their studies on both Fermi and Radeon architectures.

Thus, it is apparent that there is a need for further investigation of currently popular parallel programming models on x86 and GPGPU systems. In this research, we have selected four of the most popular x86 programming models (p-threading, OpenMP, Concurrency Runtime, and OpenCL) x86 and two of the GPGPU programming models (CUDA and OpenCL). Unlike many of the existing studies, our research varies the problem size and experiments with the algorithms that have of different "communication to computation" requirements providing well-rounded comparison between the programming models. Additionally, we have qualitatively and quantitatively ranked the

programming models in terms of programming effort required and provided insight into the relationship between programming models and application characteristics.

2.4 Performance Models

There are three general performance modeling methods used for predicting application performance on architectures: Visual Performance Modeling, Analytic Modeling and Simulation. Here we have discussed trends in these areas and their relation to our proposed *Fitness* model and its extension.

Various attempts have been made to model the performance of multicore and GPGPU architectures. In [22][34], Sam Williams introduces the Roofline model, a visual performance model, for floating-point algorithms and multicore architectures. We will discuss this model further in Section 2.6. There are some probabilistic models that analytically model the performance of multithreaded architectures. Xi E. Chen, et al. [38] presents a Markov chain model for analytically estimating the throughput of multicore architectures. They show that their models accurately predict cache contention and throughput trends across varying workloads on real hardware – a Sun Fire T1000 server. Noonberg, et al. [39] proposes a theoretical model of superscalar processor performance that is viewed as an interaction of program parallelism and machine parallelism.

In [35], Sunpyo Hong, et al., model GPGPU performance with a simple analytical model that estimates the execution time of parallel algorithms on GPGPUs. This model also provides information regarding performance bottlenecks of the algorithm on the GPGPU architecture. Sara S. Baghsorkhi, et al. [36] introduces a performance model for an application running on a GPGPU architecture. Their model is based on a symbolic

evaluation module that determines the effects of structural conditions and complex memory access on the GPGPU kernel performance. The model identifies the bottlenecks and can guide the compiler through the optimization process. Shane Ryoo, et al. [37] developed performance metrics for evaluating an optimization configuration on GPGPU architecture. By plotting optimization configurations on a Pareto-optimal curve, they were able to reduce the search space by up to 98% without missing the highest performance configuration.

Many researchers in the HPC community have used simulation-based performance models. This approach normally involves developing a model for the accelerator system and implementing the abstract workload of the application or trace data on the system model. For example, in [43], the Performance Evaluation Research Center (PERC) simulation-based frame-work is proposed which makes use of tools for machine profile and application signatures and provides automated convolution of the two. PERC can be applied both for single and multi-node processor systems.

Recently statistical methods have been applied to predict accelerator performance and identify bottlenecks [110][111]. These methods produce a machine signature by running a training set on them and derive the parameters from performance.

In our study, we have introduced a new model, the *Fitness* Performance model, which projects the *Fitness* match of the architecture with an application. The model can be applied equally to different architecture types. Our proposed *Fitness* model is a simple analytical model based on the peak performance values of the accelerators and

application characteristics taken from the algorithm. It can quickly predict the relative performance of accelerators for the parallel algorithms.

2.5 Case Studies

A single-precision matrix-matrix multiplication algorithm has been used by many researchers to analyze the performance of emerging architectures [102][103][104]. We use single-precision floating point since the four models of second case study make use of single precision. Thus to keep the precision consistent throughout the results and modeling, we use single precision floating point for all the case studies. Software routines included in libraries have been developed by vendors to accelerate matrix-matrix multiplications. BLAS with MKL [105], BLAS with ACML [106], CUBLAS [107], and CellBLAS [108] have been developed by Intel, AMD, Nvidia, and IBM respectively to accelerate matrix-matrix multiplication algorithms along with other linear algebra sub-routines.

SNNs are used as a case study for this research due to the range of compute and communication requirements, and the scalability. Further, there is increased interest in the neurology community to develop biological-scale implementations of spiking neural networks for studying the neuronal dynamics seen in the brain. The EPFL in Switzerland and IBM are developing a highly biologically accurate brain simulation at the sub-neuron level [55]. Their studies utilized the Hodgkin Huxley and the Wilfred Rall [56] models to simulate up to 100,000 neurons on an IBM BlueGene/L supercomputer consisting of 65,536 compute nodes (each computing node has 2 processors and can

have a maximum throughput of 11.2 Gflop/s). At the IBM Almaden Research Center, Ananthanarayanan and Modha [57] utilized the Izhikevich spiking neuron models to simulate 109 neurons and 1013 synapses (equivalent to a cat-scale cortical model) on a 147,456 processor IBM BlueGene/L supercomputer.

Many attempts have been made to build custom hardware for simulating SNNs. For example, the SpiNNaker project uses an ARM-based multiprocessor to evaluate leaky integrate-and-fire neurons [58]. The authors in [59] report the use of memristors for the design of neural circuits [60], and in [61], a simplified model of the neocortex based on spiking neurons was proposed using future CMOS and CMOL technologies. Most of the FPGA-based SNN implementations [62] [63] have focused on the integrate-and-fire (IF) model, which is computationally simplified (13 FLOPs) and less accurate biologically [64]. The Izhikevich neuron model has also been implemented on FPGAs [65] [66]. Among all the FPGA implementations of the Izhikevich model, the authors of [66] were able to simulate 9264 neurons in one node of a Cray XD1 and achieved a speedup of 8x. In [67], Weinstein, et. al. implemented 48 Hodgkin-Huxley neurons and achieved a theoretical performance of 8.7x real time utilizing 90% of the logic on a Xilinx Virtex-4 XC4VSX35-fg676-10 FPGA on a Xilinx XtremeDSP series development board.

2.6 Use of Roofline Model as a Visual Performance Model

As discussed in [22], the Roofline model is based on three performance components of single-program multiple-data (SPMD) type kernels: communication, computation, and locality. The communication component is from DRAM, the computation component is from the number of floating-point operations, and locality is associated with arithmetic

intensity (AI), defined as the ratio of floating-point operations to total bytes of communication. Here, communication is peak GB/s and with the computation is peak Gflop/s. The performance can naïvely be estimated as:

$$\text{Attainable GFlop/s} = \min \begin{cases} Peak\ GFlop/s \\ Peak\ GB/s \times AI \end{cases}$$

The attainable peak Gflop/s of a generic architecture is shown in Figure 2.1. Moving from left to right on the x-axis, we see a ramp up in performance (line labeled "w/ all mem opt") followed by a plateau at peak flop/s (line labeled "Peak SP FP") resembling a sloped roofline of a house. The outermost slant line represents the peak performance if all the memory optimizations are applied and the outermost horizontal line represents the performance if all the optimizations for floating-point operations are applied. On modern architectures, peak Gflop/s and peak GB/s typically cannot be achieved since it requires exploiting all architectural optimizations many of which may not apply to the given kernel. Moreover, this model assumes that the computation time and communication time is fully overlapped, which in practice is almost impossible. The most important compiler optimization techniques for floating-point performance are the Single Instruction Multiple Data (SIMD) computation, instruction level parallelism (ILP), and the balance of multiply with add instructions. Here, the compiler optimization ILP maximizes the instruction level parallelism through instruction scheduling. If any of these optimizations are not applied to a kernel, the attainable peak Gflop/s will be reduced as shown in the parallel horizontal lines below the peak SP FP in Figure 2.1. Similar parallel inner slant lines represent the peak Gflop/s when memory optimizations

cannot be achieved. Here, the memory optimization techniques are not specified, as the memory hierarchies are very different in multicore and GPGPU architectures.

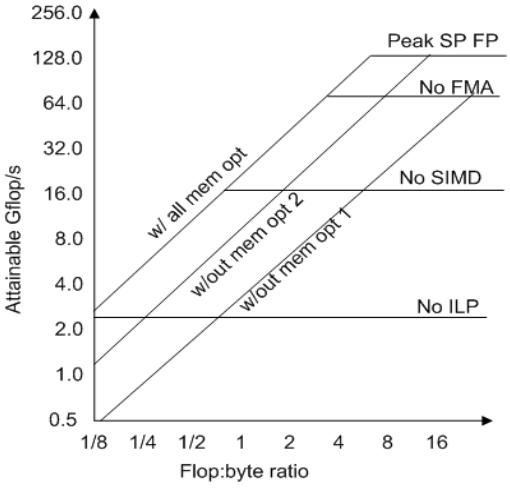

Figure 2.1. A general Roofline performance model

2.7 Conclusions

In this chapter we have discussed the research in the literature regarding performance analysis and modeling and the case studies. We have also suggested how this research is different and its relevance to the HPC field. In the next chapter we will discuss the experimental setup and methods for our experiments.

Chapter 3

Case Studies

Two types of applications have been used in this research, Spiking Neural Network models and a matrix-matrix multiplication algorithm. In this section we will first discuss the matrix-matrix multiplication algorithm followed by the four of the SNN models and the two level SNN network.

3.1 Matrix-matrix Multiplication

In this research, the first algorithm used to analyze the performance of accelerator architectures and programming models, and to later validate the proposed Fitness and regression models, is a matrix-matrix multiplication algorithm. In this case study two square matrixes are built and filled with random floating-point numbers. A typical matrix multiplication between two matrixes, A and B is shown in equation (3.1).

$$\begin{bmatrix} A_{00} & A_{01} & A_{02} \\ A_{10} & A_{11} & A_{12} \\ A_{20} & A_{21} & A_{22} \end{bmatrix} \times \begin{bmatrix} B_{00} & B_{01} & B_{02} \\ B_{10} & B_{11} & B_{12} \\ B_{20} & B_{21} & B_{22} \end{bmatrix} = \begin{bmatrix} C_{00} & C_{01} & C_{02} \\ C_{10} & C_{11} & C_{12} \\ C_{20} & C_{21} & C_{22} \end{bmatrix} \quad (3.1)$$

The floating-point matrix-matrix multiplication requires both floating-point addition and multiplication. We keep the matrix size of the matrices *A* and *B* equal and use a range of matrix sizes to scale the problem as shown in table 3.1.

Table 3.1. Floating-point operations and number of bytes required for matrix operations

Size of Each matrix	Total flops		Total bytes
	Multiplications	Additions	
100 x 100	1.00E+06	9.90E+05	2.00E+06
150 x 150	3.38E+06	3.35E+06	6.75E+06
200 x 200	8.00E+06	7.96E+06	1.60E+07
250 x 250	1.56E+07	1.56E+07	3.13E+07
300 x 300	2.70E+07	2.69E+07	5.40E+07
400 x 400	6.40E+07	6.38E+07	1.28E+08
500 x 500	1.25E+08	1.25E+08	2.50E+08
600 x 600	2.16E+08	2.16E+08	4.32E+08
700 x 700	3.43E+08	3.43E+08	6.86E+08
800 x 800	5.12E+08	5.11E+08	1.02E+09
900 x 900	7.29E+08	7.28E+08	1.46E+09
1000 x 1000	1.00E+09	9.99E+08	2.00E+09
1200 x 1200	1.73E+09	1.73E+09	3.46E+09
1300 x 1300	2.20E+09	2.20E+09	4.39E+09
1500 x1500	3.38E+09	3.37E+09	6.75E+09
1700 x 1700	4.91E+09	4.91E+09	9.83E+09
2000 x 2000	8.00E+09	8.00E+09	1.60E+10
2200 x 2200	1.06E+10	1.06E+10	2.13E+10
2400 x 2400	1.38E+10	1.38E+10	2.76E+10
2500 x 2500	1.56E+10	1.56E+10	3.13E+10

In Table 3.1, the multiplication and addition operations are shown in separate columns, but in one matrix-matrix multiplication, they are summed to calculate the total number of flops required. As seen from the table, the matrix is scaled from 100x100 to 2500x2500, which varies the total required flops from 2 Mega flops to 31 Giga flops, and the total memory accesses required from 2 MByte to 29 GByte.

3.2 Spiking Neural Networks

SNNs use a neuron-level model, where neuron states are updated using a mathematical algorithm in each time-step. Thus, all of the neurons in a network can be updated in parallel if the connections between neurons are preserved. The inherent

parallelism of the algorithm makes multicore processors and GPGPUs attractive platforms for accelerating SNN simulations.

A neuron consists of three functionally distinct parts called dendrites, axons, and a soma. Each neuron is typically connected to over 8,000 other neurons [24] [25] [26]. The dendrites of a neuron collect input signals from other neurons, while the axons send output signals to other neurons. Input signals coming in along dendrites can cause changes in the ionic levels within the soma, which in turn can cause the neuron's membrane potential to change. If this membrane potential crosses a certain threshold, the neuron is said to have "fired" or "spiked". In these events, the membrane potential rises rapidly for a short period of time (a spike) and causes electrical signals to be transmitted along the axons of the neuron to its corresponding connected neurons. Spiking is the primary mechanism by which neurons send signals to each other [26]. Over the last 50 years, several models [64][72] have been proposed that capture the spiking mechanism within a neuron. In this research, we use four of the most biologically accurate spiking neuron models for implementation on multicore architectures and GPGPUs. The model parameters are shown in Table 3.1 and the differential equations of the four SNN neurons models are given in the articles [26], [73], [74], and [75].

Table 3.2. SNN model parameters

Model	Differential Equations /neurons	Dynamic Variables /cycle/neuron	*flops /neuron
Izhikevich	2	3	13
Wilson	4	6	37
Morris-Lecar	2	4	132
Hodgkin-Huxley	4	6	265

*flops: floating point operations

3.2.1 Hodgkin-Huxley

The Hodgkin–Huxley (HH) model is considered to be one of the most biologically-accurate spiking neuron models. It consists of four differential equations and a large number of parameters describing the neuron membrane potential (twenty-five compared to only four in the Izhikevich model), activation of Na and K currents, and inactivation of Na currents. The model can describe almost all types of neuronal behavior if its parameters are properly tuned. This model is valuable for studying neuronal behavior and dynamics since its parameters are biophysically meaningful and measurable.

3.2.2 Morris-Lecar

Cathy Morris and Harold Lecar proposed a two dimensional conductance-based spiking model (ML) in 1981 [74]. The model consists of two differential equations. Three of the parameters in the differential equations are evaluated each cycle, thus adding a set of three more equations. These three equations involve hyperbolic functions, which make it computationally more expensive than the Izhikevich and Wilson models. The computational demand is lower than the Hodgkin-Huxley model however, making it popular in neurocomputation communities.

3.2.3 Wilson

The Wilson model [75], proposed in 1999, requires four differential equations. The model has more parameters than the Izhikevich model (discussed next) and tuning these parameters allows the model to exhibit almost all neuronal properties. Three of the parameters in the differential equations must be evaluated each cycle, which adds a set of three more equations.

3.2.4 Izhikevich

Izhikevich proposed a new spiking neuron model in 2003 [76] that is based on only two differential equations and four parameters. The model is attractive for slower computing systems since it requires fewer computations than the Hodgkin Huxley model (13 flops as opposed to 265 flops per neuron update) but can still reproduce almost all types of neuron responses that are seen in biological experiments.

3.3 Two-level Network

The SNN network used in this case study consists of two levels [27] [68] [77] where the first level acts as a collection of input neurons and the second level acts as a collection of output neurons. A binary input image is presented to the first level of neurons, each image pixel corresponding to a separate input neuron. The number of output neurons is equal to the number of training images that the network will recognize. Each input neuron is fully connected to all of the output neurons as shown in Figure 3.1. Each neuron has an input current that is used to evaluate its membrane potential.

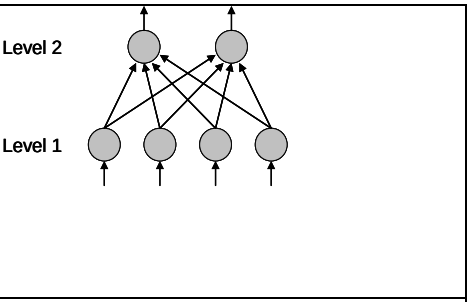

Figure 3.1. Two-level Character Recognition Network

If the membrane potential crosses a certain threshold during a cycle, the neuron is considered to have fired. For a level 1 neuron, the input current is zero if the neuron's corresponding pixel in the input image is "off". If the pixel is "on," a constant current is supplied to the input neuron. A level 2 neuron's overall input current is the sum of all the individual currents received from the level 1 neuron connections. This input current for a level 2 neuron is given by the following equation:

$$I_j = \sum_i w(i,j) f(i) \qquad (3.2)$$

Here, *w* is a weight matrix and *w(i,j)* is the input weight from level 1 neuron *i* to level 2 neuron *j*; and *f* is a firing vector where, *f(i)* is 0 if level 1 neuron *i* does not fire and a 1 if the neuron does fire. A single fire in level 2 indicates that an image has been detected.

The elements of the weight matrix, *w*, are determined through a training process as described in [27], [69]. The network was trained to recognize the 48 different input images given in [27] and later scaled for larger networks. In this study, we have accelerated the recognition phase of the network on the processor architectures and the network configurations used are reported in Table 3.3.

Table 3.3. SNN configurations for different image sizes

Input image size	Level 1 neurons	Level 2 neurons	Total neurons
96×96	9216	48	9264
192×192	36864	48	36912
240×240	57600	48	57648
384×384	147456	48	147504
480×480	230400	48	230448
720×720	518400	48	518448
960×960	921600	48	921648
1200×1200	1440000	48	1440048
1680×1680	2822400	48	2822448
2160×216	4665600	48	4665648
2400×2400	5760000	48	5760048
3120×3120	9734400	48	9734448

3.4 Summary

In this section, we have detailed the case studies used in this research. The matrix-matrix multiplication case study including problems sizes, flops, and required byte accesses are detailed in this chapter. Additionally, the four SNN models and the two-level SNN network have also been detailed forming the foundation for the following chapters. In the next chapter we will discuss the experimental systems and proposed methods for this research.

Chapter 4

Experimental Systems and the Implementations

In this chapter we present the experimental systems, implementations and optimization techniques used. The performance of the architecture set is studied by varying the problem size, available optimization techniques, number of cores, number of threads, and programming models. These studies will form the basis and analysis for the *Fitness* model and multiple statistical regression model proposed in Chapter 5.

4.1 Accelerators

Modern processors no longer rely on increasing clock frequencies to improve performance. Exploiting parallelism through multiple cores, heterogeneous systems with GPGPUs and FPGAs, memory hierarchy techniques, and in some cases vector parallelism is the current trend for achieving performance as discussed below.

4.1.1 Multicore

We have used multicore processors from four vendors (Intel, AMD, IBM, and Sun) that are currently used in state-of-the-art clusters and supercomputers. A short description of each of the multicore architectures used in this research is given below.

4.1.1.1 Intel Multicore

The first multicore processor system from Intel examined in this research is the Intel Xeon 5345 processor-based Dell PowerEdge 1950, which has two 2.33 GHz quad-core

Intel Xeon 5345 processors. These processors contain 32 KB of level-one cache per core and 8 MB of level-two cache per four cores. The processor can execute vector instructions using the SSE3 instruction set. The second Intel multicore system examined has two 2.8 GHz quad-core Intel i7 processors, for a total of 8 cores. This multicore processor has 8MB of level-three shared cache per four cores. The processor can also execute vector instructions using SSE3. The final Intel processor examined is the Intel 32 core processor available through the Intel Many Core Testing Lab. This experimental multiprocessor has four Intel Xeon X7560 2.27GHz processors, each of which has 8 cores. This processor contains 24.5 MB of shared cache per 8 cores and 256 GB of RAM.

4.1.1.2 AMD Multicore

The two X86 architecture processors from AMD studied in this research are the Opteron 2356 processor-based SunFire 2200 and an experimental 32-core processor. The SunFire 2200 has two 2.33 GHz quad-core AMD Opterons. These processors contain 64 KB of level-one cache per core, 512 KB of level-two data and instruction cache per core, and a 2MB level-three cache shared by four processing cores. Similar to the X86 processors from Intel, the Opterons can execute vector instructions using the SSE3 instruction set. The AMD 32-core multiprocessor is a 6134 system that has four, eight-core processors (total of 32 cores). The processor executes at 2.3 GHz with 256 GB of memory for the processor. These processors contain 512 KB of cache and can execute vector instructions using the SSE-like instruction set.

4.1.1.3 Sun T2+ UltraSparc

Available from Sun, the SPARC Enterprise T5440 Server is a long-term storage server. It has four,eight-core T2+ CoolThread processors (32 cores total). The processor operates at 1.4 GHz with 128 GB of memory. These processors contain a 4 MB level-two cache and unlike the X86 processors, cannot execute vector instructions with the SSE-like instruction set.

4.1.1.4 IBM Cell Architecture

The Cell Broadband Engine developed by IBM, Sony, and Toshiba [68] is a multicore processor that heavily exploits vector parallelism. The current generation of the IBM Cell BE processor operates at 3.2 GHz and consists of nine processing cores: a PowerPC based Power Processor Element (PPE) and eight independent Synergistic Processing Elements (SPE). All processors and internal RAM are connected through an Element Interconnect Bus (EIB). The PPE is primarily used for administrative functions while the SPEs provide high performance through vector operations. In the PS3 version of the Cell BE used in this research, only six of the eight SPUs are available for computation. The second Cell architecture examined in this research is a QS22 blade that has two IBM Cell Broadband Engine (BE) processors operating at 3.2 GHz and consisting of nine processing cores each: a PowerPC based Power Processor Element (PPE) and eight independent Synergistic Processing Elements (SPE). All processing cores and the internal 8GB of RAM are connected through an Element Interconnect Bus

(EIB). Like the PS3 version, the PPE is primarily used for administrative functions while the SPEs provide high performance through vector operations.

4.1.2 GPGPU

In the following subsections we will discuss the architecture of the most popular and powerful GPGPUs from NVIDIA and AMD.

4.1.2.1 NVIDIA GPGPU

Two NVIDIA GPGPUs were used in our research; the 1st generation Tesla C870 and the 20-series Tesla C2050, also known as Fermi. The GPGPU architecture appears as an array of streaming multiprocessors each containing scalar processors, special function units, double-precision units, and local memory (in CUDA terminology, it is called shared memory) to enable thread cooperation. NVIDIA's Tesla C870 card uses compute capability 1.0 and has 16 multiprocessors consisting of 128 cores, 1.5 GB global memory, 64 KB constant memory, 16 KB local memory and operates at a clock rate of 1.35 GHz. The system provides over 1500 MB/sec host-device transfers for pageable memory and over 60,000 MB/sec device-to-device bandwidth.

NVIDIA's Tesla 20-series architecture, codenamed Fermi, has brought a lot of innovation versus previous architectures: 512 CUDA cores organized as 16 SMPs with 32 cores each gathered around a L2 cache. A Gigathread scheduler dispatches thread blocks to the SMP thread schedulers. The GPGPU has the capability of supporting 6 GB of GDDR 5 DRAM memory. The SMPs also have a new look with an instruction cache, dual warp schedulers and dispatch units. SMPs now have two sets of 16 CUDA cores, 4

special function units for transcendental functions, 16 load/store units, a hefty register file, and most importantly, a configurable 64 KB of shared memory/L1 cache. The SMPs also share a second level L2 cache. More information about the architecture can be found in [69]. For our studies, we have used a single NVIDIA Tesla C2050 belonging to the Compute Capability 2.0 with 14 multiprocessors (448 cores), 2.6 GB global memory, 64 KB shared memory/L1 cache per multiprocessor, 768 KB L2 cache, 64 KB constant memory, and operating at a clock rate of 1.15 GHz. The system's GDDR interface offers data bandwidth up to 144 GB/sec.

4.2 Programming Models

In this section, we will discuss the four programming models used for the X86 multicore processors (X86POSIX-threading, OpenMP, Concurrency Runtime, and OpenCL) and the two programming models for GPGPU (OpenCL and CUDA).

4.2.1 POSIX-threading

POSIX threading [81] or p-threading is a C-based threaded programming model. For UNIX/Linux systems, this programming interface has been specified by the IEEE POSIX 1003.1c standard. In shared memory multiprocessor architectures, such as a symmetric multiprocessor (SMP), p-threading programming is used to exploit the parallelism offered by SMPs. Most modern multicore processor systems use the SMP architecture. In this paper, we have utilized the p-threading programming model to exploit the SMP-type Intel i7 multicore processor and study its performance variation.

4.2.2 OpenMP

Open Multi-Processing (OpenMP) [82] is an application programming interface that focuses on compiler directives to accelerate multi-threaded, shared memory programs using C, C++, and Fortran on Linux and Windows system. OpenMP uses a master thread approach to run serial code and fork new threads during sections of computation that benefit from multiple cores and/or processors. Once the parallel code has been completed, the threads are joined back into a master thread until parallel execution is required again. The programmer is given some degree of control over this process, but the details of the parallelization are left to the OpenMP compiler itself. Built-in options for controlling thread synchronization, data sharing, thread scheduling, and other basic functionality do exist in the API, though, and the number of threads to create can be set statically or dynamically by the user. The OpenMP standard does not require compilers to provide explicit checking for data dependencies, race conditions, deadlocks, or other hazards of parallelization, so any user-defined blocks of parallel code must be reviewed and analyzed carefully to determine suitability for OpenMP compiler directives.

4.2.3 Concurrency Runtime

There are three libraries provided by Visual Studio 2010 to facilitate parallelizing an application, including Concurrency Runtime, the Parallel Pattern Library, and the Asynchronous Agents Library [83]. These libraries provide powerful APIs for application parallelization [83]. In this research, we use Concurrency Runtime to

describe the programming model constituted by these three pieces. The Concurrency Runtime libraries provide several benefits such as cooperative scheduling, nested parallelism, and cooperative blocking for parallelizing an application.

First, in cooperative scheduling, Concurrency Runtime uses a cooperative task scheduler that implements a work-stealing algorithm. In the work-stealing algorithm, if a thread completes its assigned tasks, it offloads tasks from other threads that have remaining tasks. In this way all threads are kept busy. Second, in nested parallelism, if two nested loops are parallelized using concurrency runtime, the created threads coordinate with each other to share computation and communications resources. Third, in cooperative blocking, if a thread cooperatively blocks in a particular loop iteration for a specific resource, the Concurrency Runtime spawns another thread that will execute the remaining loop iterations if there are any remaining. The detailed architecture of the Concurrency Runtime can be found in [83].

4.2.4 OpenCL

The OpenCL programming model allows developers to target x86, GPGPU, Cell, and DSP architectures. A kernel written in OpenCL describes the functionality of each work item, or thread. All created work items run the same kernel but on different data sets. The developer can specify the number of work items in a local group, called a local work group size. If the target platform is an NVIDIA GPGPU, this work group can be further divided into a group of 32 threads, called warps. Several concurrent local work groups can run on a multiprocessor, depending on multiprocessor resources such as local

memory size and number of registers. The total number of work items in a kernel call is the global work group size and this size is defined before the kernel call. For an AMD GPGPU the kernel can support vector calculations. Thus if the algorithm allows, all or part of the computation in a kernel can be executed in a vectorized fashion.

OpenCL also supports any modern X86 processors whereas, CUDA does not support any platforms other than NVIDIA GPGPUs . In an X86 processor, the OpenCL kernel is translated into p-threading like functions. All of the APIs in OpenCL, that are used in GPGPU programming can also be used in X86 programming. The OpenCL runtime translate the APIs, which are originally intended for GPGPU hardware, into CPU hardware.

4.2.5 CUDA

In CUDA for C [85], the GPGPU functionality is defined by writing device functions called C Kernels. Typically, only one kernel can be executed on the GPGPU at a time. A *thread*, which is a sequence of instructions, is instantiated several thousands of times. When a kernel is called, *N* threads execute the kernel in parallel. Threads are accessed inside kernels using built-in variables: *threadIdx, blockIdx,* and *blockDim.* Collections of threads, called thread blocks are executed on the SMPs. 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 memory hierarchy in CUDA is comprised of a set of *registers* (on-chip) and *local memory* (residing in an off-chip DRAM) for each thread, a private shared memory for thread blocks, a *global memory* for all threads launched, and

read-only *texture cache* and *constant memory.* CUDA offers three primary optimization strategies, namely the *Memory Optimization, Execution Configuration Optimization*, and *Instruction Optimization*.

4.3 Matrix-matrix Multiplication: Implementation

The Matrix-matrix multiplication algorithm has been long studied in computing due to its pervasiveness in scientific computing. Many libraries have been developed to optimize the algorithm on various hardware platforms. On Intel the x86 platform, mkl_cblas library is provided by the Intel Math Kernel Library (MKL) for the faster matrix-matrix multiplication. On AMD platform, AMD Core Math Library (ACML) provides the best performance for the algorithm. Similarly, for matrix-matrix multiplication Cellblas and CUBLAS libraries are available for IBM Cell and Nvidia GPGPU platform. We have implemented the matrix-matrix multiplication algorithm on the Intel Xeon, AMD Opteron, IBM Cell, and Nvidia Fermi platform using the corresponding math libraries to obtain the best performance.

4.4 SNN: Implementation And optimization

In this section we first discuss the implementation and optimization techniques for the case studies on multicore architectures and finally the GPGPU implementation and optimizations of the case studies will be discussed.

4.4.1 Multicore

4.4.1.1 Multicore Implementation

Network parallelization: All neurons in both levels of the SNN evaluate in parallel and are independent of each other, allowing the neurons to be divided evenly across all available processing cores. Each processing core is assigned a set of level 1 neurons and generates a level 1 firing vector after evaluating all neurons in this set.

Evaluation of the level 2 neurons on the PS3 architecture is different from the X86 and SUN T2+ architectures. In the X86 architectures, the firing vectors store the index of each level 1 neuron that was evaluated by that processing core and which, if any, fired. The number of firing vectors is the same as the number of processing cores utilized. Each processing core reads all firing vectors form the other processing cores when evaluating the level 2 neurons. This approach has the advantage of simplifying the level 2 neuron weight computation in training, requiring only examination of the level 1 neurons that fired.

In the PS3 architecture, after generating its firing vector, each SPU calculates the current input for level 2 using its firing vector. Thus there will be six input current vectors for level 2, one per SPU, where each vector has 48 elements corresponding to the 48 output neurons. These current vectors are then sent to the PPU where they are accumulated to form the total current for each level 2 neuron. Finally, the PPU evaluates the level 2 neurons.

Vectorization: Neurons in a level are independent of each other. Thus, simply evaluating four neuron-level calculations in parallel on each processing core vectorizes the network.

4.4.1.2 Multicore Optimization

On the Intel and AMD architectures, three types of optimizations are applied: a) POSIX-threading (pth), b) SIMD computation (SC), and c) Software Prefetching (SP). POSIX threading (p-threading) was used to create threads and distribute the computations among threads. All of the X86 architectures support the SIMD computation called Streaming SIMD Extension 3 (SSE3). In this SSE3 technique, four similar floating-point operations are carried out in one cycle using 128-bit registers. Software prefetching techniques used in the GPGPU OpenCL optimizations (discussed later) can also be applied in the X86 implementations. Finally, software prefetching shows a significant speedup for most of the SNN models investigated, as will be seen in the results section.

The following optimization techniques were explored for the PS3 architecture: a) Multi-Threading (MT), b) SIMD Computation (SC), c) Double Buffering (DB), d) Reducing Conditional Statement (RCS), e) Loop Unrolling (LU), and f) Software Pipelining (SP). In the multi-threaded technique, 6 threads are created for execution on the 6 SPUs. In the SIMD optimization technique, four similar floating-point operations are carried out in one cycle using 128-bit registers. In the PS3, data transfers are accomplished through a DMA engine where two sets of variables are defined for computation. Double buffering is used to overlap communication with computation allowing one DMA request to transfer data from DRAM to one set of variables, while the SPU is computing on the other set of variables. Since the SPUs of the PS3 do not have branch prediction units, the technique of reducing conditional statements is useful to improve performance. Further performance gains can be achieved if conditional statements are reduced through loop unrolling. We have experimented with varied

amounts of unrolling in our case studies and found that by unrolling up to 8 loops, we achieve a measurable performance gain. Finally, software pipelining is implemented by prefetching the variables that will be used for computation. Software pipelining can achieve good performance if it is combined with loop unrolling since the variables are used for multiple computations instead of fetching each array element each time it is used.

4.4.2 GPGPU

The GPGPU implementation involves acceleration of the level 1 neurons, the most compute intensive level of the two levels, on the GPGPU. Level 2 is less than 5% of the total computation and requires accessing the large weight matrix (48 times the size of the level 1 voltage array). The GPGPU kernel is invoked at each time step and if the level 2 neurons are computed in the GPGPU, the weight matrix transfer time would dominate the computation time, slowing down the overall implementation. The size of the weight matrix increases with the network size and is 48 times larger than the other data structures used in these applications. For example, the network of 5.7 million neurons, the weight matrix size will be ($5.7\times10^6\times48\times4$ Byte =) 1.03 GB. To transfer this data from the host to device, takes about 0.7 sec (considering the data transfer rate of 1500 MB/s as given in Section 3.2.2.1). For the Izhikevich model, this will cause 244% increase in runtime and for the Hodgkin-Huxley model, the runtime will be increased by 35%. Evaluating the level 2 neurons on the GPGPU cannot compensate for this substantial increase in data transfer overhead since the level 2 neurons only take 5% of the total compute runtime. We have confirmed this performance analysis with an implementation

in which both network levels are evaluated on the GPGPU. However, if the kernel is only invoked once in the life of the simulation, it would provide a performance gain to evaluate both network levels on the GPGPU because, in that case, the overhead related to the kernel call and firing data transfer in each simulation cycle can be avoided. Unfortunately, the algorithm requires synchronization across all the threads in each simulation cycle and currently there is no way to synchronize all the threads inside the GPGPU kernel. For these reasons, the level 2 neuron evaluations and the associated weight matrix are kept on the host instead of porting them to the GPGPU.

The GPGPU computes the neuron dynamics of level 1 and supplies the resulting firing data to the host. The host uses this firing information to compute the neuron dynamics for level 2. A thread on the GPGPU computes the single neuron dynamics for level 1; the functionality of the thread is described by the kernel code. Different code optimization techniques are applied and evaluated for performance improvement as will be discussed in the next subsection. We have avoided data transfers between the host and GPGPU as much as possible, the only frequent communication involved was the transfer of the firing information from level 1 (from the GPGPU to the host) in each time step.

4.4.2.1 GPGPU Optimization

OpenCL was used to implement the SNN models on the GPGPU. A program written in OpenCL can target various architectures including CPUs, GPGPUs, CellBEs, DSPs and many other multiprocessors. We have also implemented the SNN with four neuron models in CUDA on Telsa C870 and presented results previous paper [78]. In this study, our target platform is a NVIDIA's GPGPU.

A kernel written in OpenCL describes the functionality of each *work item*, or thread. All created work items run the same kernel but on different data sets. The developer can specify the number of work items in a local group, called a *local work group size*. If the target platform is an NVIDIA GPGPU, this work group can be further divided into a group of 32 threads, called *warps*. Several concurrent local work groups can run on a multiprocessor; the number depends on the multiprocessor resources such as local memory size and number of registers.

OpenCL provides several optimization techniques that can be employed for optimal performance including Memory Level, Instruction Level, and Execution Configuration Optimizations. The optimization techniques used in this work are, a) Multithreading (MT), b) Software Prefetching (SP), c) Local Memory (LM), d) memory write function (MW), e) Native Math (NM), f) unsafe math (UM), and g) reducing conditional statement (RCS). The first naïve implementation uses a multithreading (MT) technique where the same OpenCL kernel is executed by all of the created work items. In each cycle, the kernel is launched from the host code and the firing information from the level 1 neurons is sent from the GPGPU to the host. This configuration is the base implementation and all other optimization techniques are applied to the base implementation.

Memory Level Optimizations: In the kernel code, the variables that determine the state of neurons are accessed many times from the global memory, which takes several clock cycles per access. Thus, to reduce the number of accesses to the global memory, these variables are first stored in local variables and once the processing is complete, they

are transferred back to the global memory. This memory optimization technique is called Software Prefetching (SP).

In every cycle of neuron updates, the firing vector containing the firing information from the level 1 neurons is sent from the kernel back to the host. The size of this firing vector is equal to the number of neurons in the level 1. From the OpenCL profiler, it is found that the time to send this firing vector is almost 40% of the total GPGPU device runtime. To reduce this data communication time, a separate array, called local_fire, is declared in the local memory. Each entry of this array contains a record of any firing in a local work group after executing a logical operation within a local group. This local_fire array, which has the size of 1/(local_group_size) of the firing vector, is sent to the host in every cycle and since it is significantly smaller, the data communication time is proportionally reduced. On the host, this vector is tested for any firing in the local_fire array and the full firing vector is only sent to the host if there is a flag set (which indicates the presence of firing) in the local_fire vector. This technique improves the performance because a neuron firing occurs in a small fraction of the cycles. The data communication cost between the host and GPGPU is reduced to about 3% of the total GPGPU time. This technique is referred to as local memory (LM) and is shown in Figures 4.1 and 4.2 In Figure 4.1, where the local memory technique is not used, the entire firing vector is transferred in each cycle. In Figure 4.2, where the local memory technique is used, the smaller local firing vector (Firing Vector/local_group_size) is transferred in each cycle, reducing the total data communication.

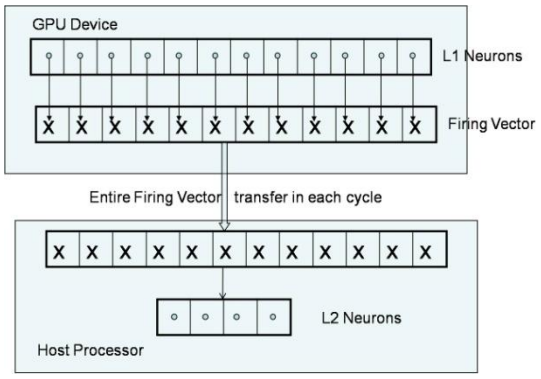

Figure 4.1. Data transfer without using local memory

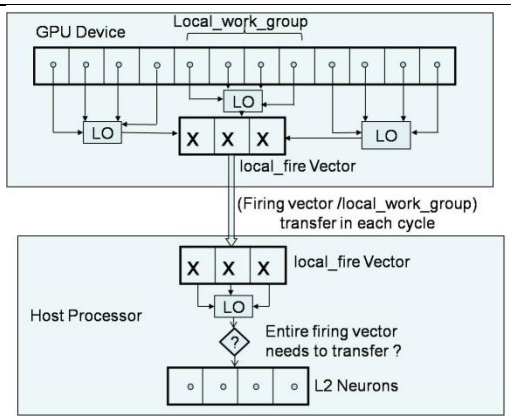

Figure 4.2. Data transfer using local memory (LO=Logical Operation)

In the naïve implementation, while sending the array from the host to the GPGPU, the clCreateBuffer() function was used to create a buffer and copy data from the host to the GPGPU. It is found that if the buffer is created in the initialization part of the application and then the function clMemWrite() is used inside the execution loop of the host function, better performance is achieved. This memory optimization is called memory write (MW).

Instruction Level Optimizations: OpenCL provides native math (NM) functions that are optimized versions of exponentials, trigonometric, and other complex functions. These native functions provide improved speed with some loss of precision and are therefore to be used cautiously so that the errors in precision do not negatively impact the overall results. Another math optimization technique that OpenCL offers is called unsafe math (UM). In this optimization, the compiler optimizes the floating-point arithmetic in the kernel but it may violate IEEE 754 standard and OpenCL numerical compliance requirements. Thus, this optimization technique is also to be used cautiously. The last

applied optimization is the reduction or elimination of conditional statements (RCS). If conditional statements are present, the threads in a local work group may be serialized, which will slow down the kernel execution. Thus, reducing any existing conditional statements will improve the performance.

4.5 Summary

In this chapter we have discussed the architectural features and peak performance of various computing architecture that are used in this research. We have also discussed the implementation basics of the matrix-matrix multiplication and SNN algorithms on these architectures and the optimization techniques used to improve performance.

Chapter 5

Architecture Performance Analysis

Experimental data was collected with the available multicore and GPGPU architectures to (a) validate the Fitness performance model and (b) provide data to study the effects of programming models on performance. The performance analysis of the Intel Xeon (8-core), AMD Opteron (8-core), IBM PS3, (6-core) and NVIDIA Fermi including problem size scaling for the matrix-matrix multiplication algorithm and SNN models is reported. Then the effect of accelerator configuration and optimization techniques for these architectures is also shown. We also report results for different programming models that will be studied in more detail in the next chapter. Performance results for the advanced processors, Intel-32 core, AMD 32-core, and IBM PowerXCell-based QS22 are also presented with a variation in the number of threads. -

The problem size for the case studies is varied to study the scalability of the architectures. For the SNN case study, we also investigate the impact of optimization techniques, and accelerator configuration (number of threads for the multicore and local work group size for the GPGPU) on performance.

5.1 Performance Results For Matrix-matrix Multiplications

The performance of the architectures (Nvidia Fermi, Intel Xeon, AMD Opteron, and IBM PS3) for a matrix multiplication algorithm is shown in Figure 5.1. As seen in the

figure, the performance of all the accelerators increases with the increase in problem size. The Fermi provides the maximum performance of 3370x for a matrix size of 2500x2500, over a serial implementation on Intel i7. For the same matrix size, the performance of the IBM PS3, Intel Xeon and AMD Opteron was 175x, 106x and 80.5x respectively. It is interesting that for smaller problem sizes, the PS3 performance is lower than the two x86 systems, while its performance improves and surpasses the x86 systems with the larger problem sizes. The thread creation and signaling overhead for the PS3 dominates for the smaller problem sizes but with the larger problem sizes the overhead becomes negligible.

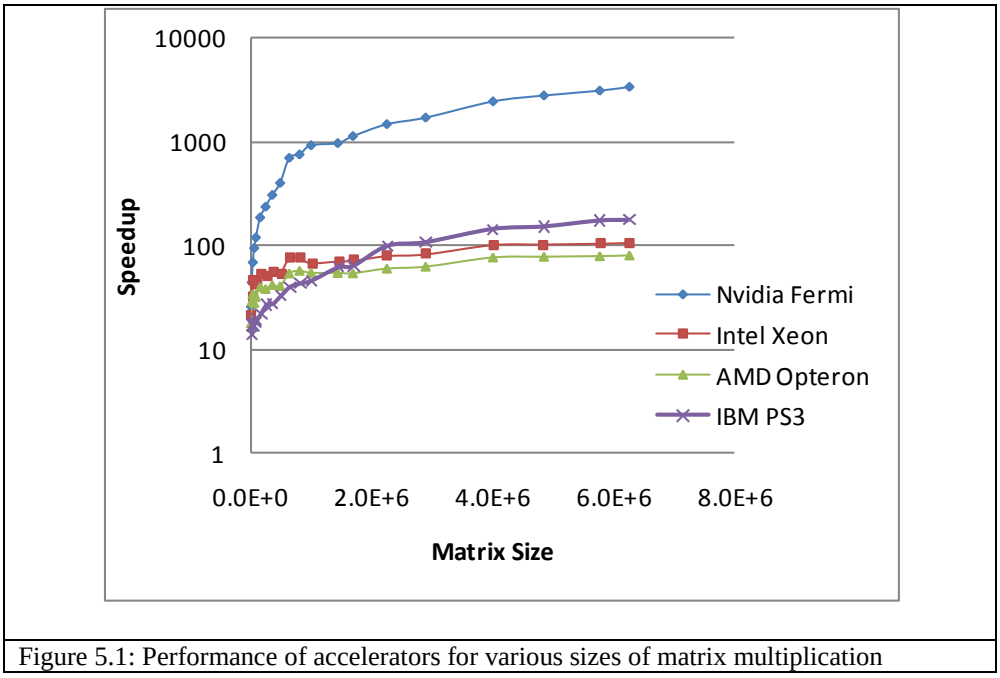

Figure 5.1: Performance of accelerators for various sizes of matrix multiplication

5.2 Performance Results For SNN models

This section introduces the performance results of the four base architectures for varying problem (network) sizes. The results are organized based on the SNN models.

5.2.1 Izhikevich Model

The performance of the four base architectures for the Izhikevich neuron model is shown in Figure 5.2. For the Fermi GPGPU implementation, the speedup initially increases with an increase in the number of neurons (problem size). This architecture provides only a nominal performance increase beyond 1 million neurons. A maximum speedup of 18.33x was observed for 5.8 million neurons. Initially, the speedup increases due to the increasing problem size. Referring to the Table 5.1, since the flop:byte ratio for this model is only 0.65, the performance improvement cannot be sustained because the nominal amount of floating-point operations on the GPGPU is unable to amortize the increased communication time required for larger network sizes. GPGPUs require a higher flop:byte ratio to overcome data transfer overheads and fully exploit the task and thread level parallelism.

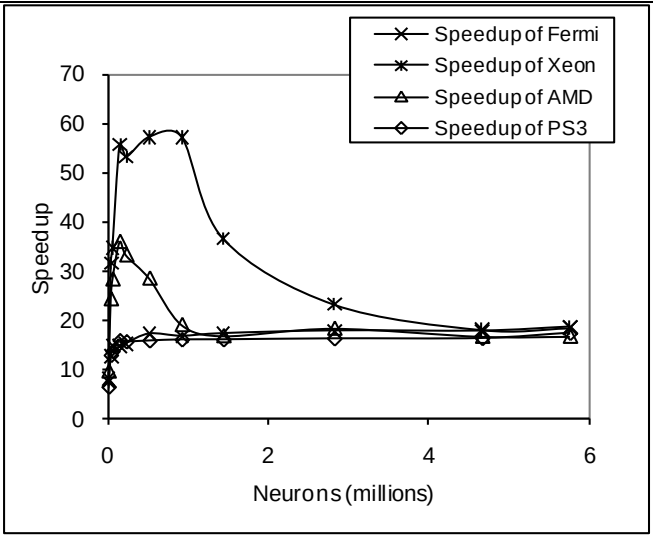

Figure 5.2. Izhikevich model: Speedup performance of the four architectures over Intel Core 2 Quad

Table 5.1. Flops and data transfer for neuron dynamics updates

Models	Flops	Memory Access per Neuron (Byte)	flops/byte ratio
Izhikevich	13	20	0.65
Wilson	38	44	0.86
Morris-Lecar	132	28	4.71
HH	265	44	6.02

As seen in Figure 5.2, the Xeon performance increases to about 58x for 1 million neurons and then begins to decrease until it stabilizes at 20x. The performance diminishes after 1 million neurons because of the increasing amount of data required to update the neurons cannot be accommodated by the cache, resulting in cache misses. Thus, the communication time grows faster and dominates the computation time, reducing the attainable speedup. The performance curve for the AMD Opteron has a similar shape with the exception that its peak performance was only 36x and begins to fall off sooner. Though both of the X86 systems are similar in most of the specifications, the Xeon performs better than the Opteron for almost all of the problem sizes. One of the main reasons is that the cache size of the Intel Xeon is larger than that of the AMD Opteron so for larger problem sizes, the Opteron may have more cache misses than the Xeon. We have further investigated this issue and found that the RAM used by the Opteron is 128-bits whereas the Xeon has 240-bits. Further, the Opteron uses ECC RAM modules while the Xeon uses fully-buffered DIMMS. The ECC RAM used by the Opteron imposes a restriction on the throughput that may also negatively impact the Opteron's performance.

The IBM PS3 demonstrates similar performance to the Fermi GPGPU for this model, showing a speedup of 17.3x for 5.8 million neurons. The PS3 has a regular performance

curve that increases initially and then saturates at around 17x. The initial increase in speedup can be attributed to the dominance of parallel computation over the fixed overhead cost (such as thread creation, signaling, and barrier synchronization). When the problem size is small, the communication and computation both can be parallelized, improving the performance. But after the initial increase, the performance saturates because the communication time becomes significant with the increase of problem size and the memory accesses cannot be fully parallelized as before.

5.2.2 Wilson Model

The performance results of the four architectures with the Wilson model are shown in Figure 5.3. Performance of the GPGPU increases to 29x at around 1 million neurons and then begins to saturate. The maximum speedup of the GPGPU is 31.7x at 5.8 million neurons. The GPGPU performance for this model is better than the Izhikeveich model because of the larger flop:byte ratio (0.86 vs. 0.65). The GPGPU performance does not have a maxima as seen in the X86 architectures, because there is no cache-size effect for the GPGPU. For the GPGPU, the entire required dataset fits in global memory (1.5 GB) up to the maximum size of the network so there will be no conflict or capacity misses.

The shape of the Xeon performance curve for the Wilson model is similar to that for the Izhikevich model, reaching a peak at 0.2 million neurons and then falling until it stabilizes at about 1.4 million neurons. The peak speedup is observed at 0.2 million neurons, where the data required begins to exceed the cache capacity of the Xeon architecture. The speedup for the Wilson model reaches this peak earlier than with the

Izhikevich model, because the data required is more than twice that of the Izhikevich model (see Table 5.1). Thus for the Wilson model, the capacity and conflict misses in the cache start earlier than were seen for the Izhikevich model. For the same reason, it is noticed that the peak speedup for the Wilson model cannot grow beyond 23x, whereas for the Izhikevich model the peak speedup was 58x. The performance curve for the Opteron with the Wilson model has a similar shape and explanation to that of the Xeon with the exception that the peak speedup is 15x and falls off earlier.

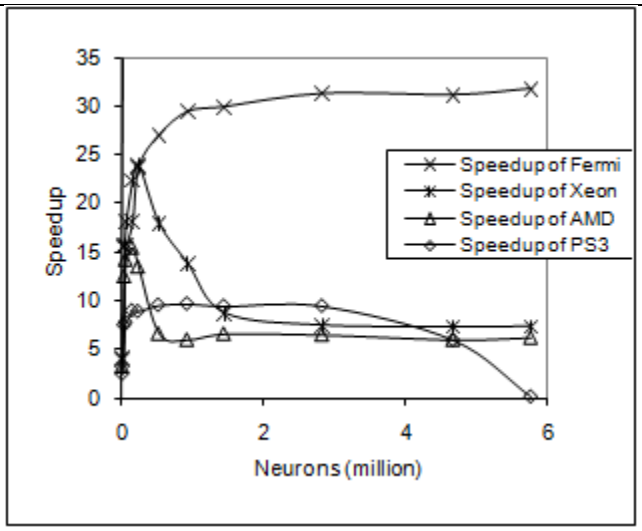

Figure 5.3. Wilson model: Speedup performance of the four architectures over Intel Core 2 Quad

The PS3 performance for the Wilson model is very different from the other architectures. The speedup increases to 9x at about 0.15 million neurons and the speedup stays at that value until 4.67 million neurons after which it starts to diminish until finally at 5.8 million neurons there is a slow down instead of a speedup. We have investigated the cause of the diminishing speedup beyond 4.67 million neurons. There is no capacity

or conflict cache misses involved in this case because the SPUs directly read data from DRAM through the DMA engine and put the data to its local store. Since the programmer manages the data transfer between the SPUs and DRAM, cache misses are ruled out. If the programmer puts more data in the local store than its capacity, the program crashes. Thus the only reason for the diminishing performance is that the working dataset exceeds the DRAM size. The network of 4.67 million neurons requires a total of 925 MB of data whereas the DRAM size is only 512 MB. Thus, this size network uses swap space on the hard disk to get the required data for updating the neurons. It is well known that the hard disk bandwidth is lower than the DRAM bandwidth; for this reason, the data communication time becomes much higher than the case where the working dataset fits in the DRAM (for example, 512 MB DRAM can accommodate a working dataset of 1.44 million neurons, which requires only 286 MB of data). For 5.8 million neurons, the speedup is 0.1, which means the parallel code in the PS3 is 10 times slower than the serial implementation. This speedup is much worse than the previous network sizes because it requires 1.14 GB of data, which will require more frequent accesses to the hard disk, degrading the performance.

5.2.3 Morris-Lecar Model

Figure 5.4 shows the speedup of the Morris-Lecar model for the four architectures. The GPGPU performs the best among all the architectures for this model. The GPGPU's performance initially grows with the network (problem) size until it begins to saturate at about 169x for a network size of 1.4 million neurons. The peak speedup of the GPGPU is 187.8x, which is higher than that observed for the Izhikevich and Wilson models. The

higher performance is possible because the flop:byte ratio for this model is higher, 4.71. Additionally, the Morris-Lecar model equations have hyperbolic functions that can be optimized by using the native math functions.

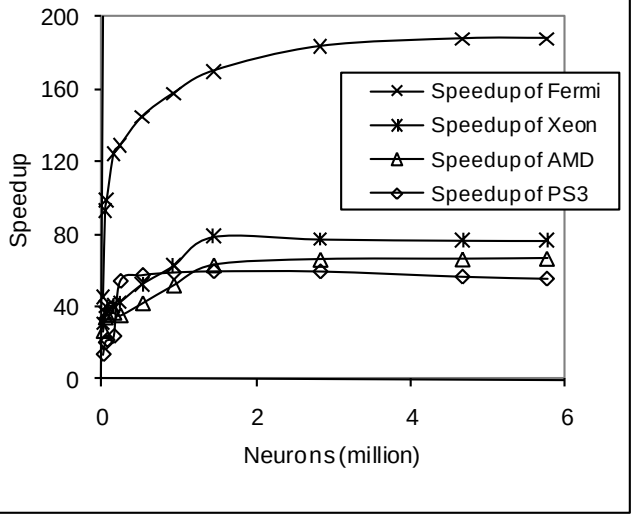

Figure 5.4. Morris-Lecar model: Speedup performance of the four architectures over Intel Core 2 Quad

The Xeon performance increases with the network size until it reaches 78x at 1.44 million neurons where it begins to saturate. The peak speedup of the Xeon, 78x, is significantly higher for the ML model than the Izhikevich or Wilson models because of the higher flop:byte ratio. There is not a noticeable maxima in the performance like the Izhikevich and Wilson cases because, though the working dataset for larger networks with this model exceeds the cache size of the Xeon and increases the number of capacity and conflict misses, the ML model has enough computation to amortize the increased data communication time. The Opteron performance initially increases until 65x at 2.8

million neurons where it saturates. The Opteron performance curve always stays below that of the Xeon due to the reasons explained earlier for the Izhikevich model.

The PS3 performance curve for the ML model initially grows to 59x at 0.9 million neurons and then saturates. The PS3 speedup for this model is larger than that of the Izhikevich and Wilson models again because of the model's higher flop:byte ratio.

5.2.4 Hodgkin-Huxley Model

Performance of the HH model on the four architectures is very different from the other three models as seen in Figure 5.5. With this model a significant speedup is achieved and the GPGPU clearly outperforms the other architectures. The peak speedup of 945x was observed for 5.8 million neurons. A substantial speedup was expected and seen for the GPGPU implementation since this model involves a significantly higher flop:byte ratio, 6.02 as shown in Table 5.1. There are several other reasons for achieving such a significant speedup for this model. First, the working dataset involved for this model (131 MB data for 5.8 million neurons) fits in the GPGPU global memory (1.5 GB), which means there are no cache or DRAM misses. In the other architectures, conflict and capacity misses cause the degradation of performance. Second, a significant and parallelizable number of flops are involved in this model, (265 flops/neuron, 416 Gflops for 5.8 million of neurons), which is the highest of all the models. We have verified with the CUDA profiler that the kernel execution time accounts for the 98% of the runtime and the host-to-device data communication time is only 2%. Thus, the highly-parallel execution model of the GPGPU, (each thread updates each neuron) can vastly improve the floating-point performance and in turn, the overall performance.

Third, there are math functions, such as exponential functions, in this model that can benefit from native math optimizations, which in this case improve the performance without losing noticeable precision. Fourth, all of the other available optimization techniques (such as fast math optimization, reducing conditional statements) are applied to this model (as explained elaborately in Section, 6.1) and further improve the performance.

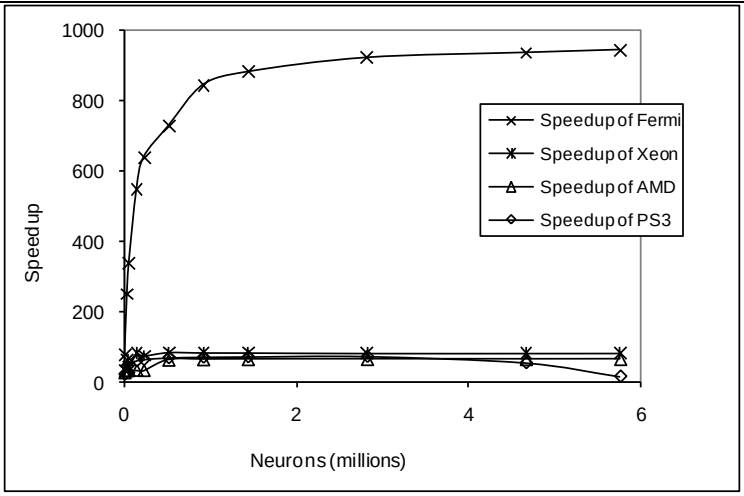

Figure 5.5. Hodgkin-Huxley model: Speedup performance of the four architectures over Intel Core 2 Quad

Performance of the Xeon and Opteron initially grows but saturates at 88x and 68x respectively at 0.5 million neurons. No maxima is observed in the Xeon or AMD speedup curves for this model; although the working dataset is larger than the cache size causing capacity and conflict misses, the model has enough parallelizable computation to amortize the increased communication cost.

As seen in Figure 5.5, the PS3 performance for the HH model has an initial increase up to 70x at 0.5 million neurons and then saturates until 2.8 million neurons where it

begins to fall. This behavior is attributed to the main memory size of the PS3, which is only 512 MB. The network of 0.5 million neurons requires a data size of only 106 MB, which fits into the main memory. Once the data size increases beyond the capacity of main memory, which occurs for a network size of 2.8 million neurons (requires 581 MB data), the performance begins to decline. Network sizes of 2.8 million neurons and larger will therefore cause page table misses and use the swap space located on the hard-disk. This abrupt increase in data communication overhead needed to move data in and out of main memory causes a decrease in performance as seen in Figure 5.5 for networks of 2.8 million neurons and larger. At the maximum size of the network, 5.8 million neurons, we still see a speedup of 16x, where for the Wilson model a speedup of 0.1x was observed for this size. Though the HH model and the Wilson model have the same working dataset size, the HH model has significantly higher flops/neuron than that of the Wilson model (264 flops as opposed to 37 flops), allowing the computation to amortize the communication cost.

5.3 Impact of Optimization Techniques

The impact of optimization techniques on performance is useful for parallel program development for GPGPU and multicore architectures. For each of the multicore architectures, two network sizes are selected to study the effects of optimizations on performance. All four base architectures use the maximum network size (5.8 million) and the second size is selected based on the location of the maxima in the performance graphs (Figures 5.2-5.5).

5.3.1 Fermi GPGPU

The mapping of the two-level neural network on the GPGPU architecture has been discussed in detail in Section 4.3 of Chapter 4. As mentioned in Section 4.3, the computationally dense level 1 neuron dynamics are performed on the GPGPU while the level 2 neuron dynamics are evaluated by the host processor. In each simulation cycle, the GPGPU device provides the host processor with the level 1 firing information so that the level 2 neuron dynamics can be evaluated.

Three primary optimization strategies have been used in this research with the CUDA programming model: Memory Optimization, Execution Configuration, and Instruction Optimization [112]. Prior to presenting each of the implementations, we first briefly describe these primary optimizations.

Several memory optimization strategies can be found in [85]; in this sub-section we introduce the optimization techniques used in this study. First, frequent host-device transfers must be reduced since the host-device bandwidth is several orders of magnitude smaller than the device-device bandwidth. It is highly beneficial to transfer all relevant data to the device memory for processing and later transfer the data back to host memory once all operations are finished. Host-device communication can be overlapped with kernel execution using Zero Copy (Z). Nvidia's Fermi architecture with compute capability 2.0 has a significantly different memory structure from previous GPGPU devices with the introduction of L1 and L2 caches. The Fermi architecture also allows the user to configure the amount of L1 cache and shared memory used. From the 64 KB of on-chip memory, 48 KB can be configured either as L1 cache or shared memory. The

user is also allowed to cache the global memory either in L2 cache alone or both in L1 and L2 caches [85]. Caching the global memory can promote performance improvement in applications that involve frequent global memory data accesses or those that suffer from register pressure. Software-Prefetching (SP) is another optimization technique that can be used to cache the computation variables in on-chip registers, providing faster access. Alternatively, Shared Memory (SM) can be used instead of registers to alleviate register pressure. As explained in [112], SM has been used in our research to minimize the communication between the device and the host.

Execution Configuration Optimization is an effective method to hide latency for memory bound kernels. Execution configuration is related to the number of threads per block. Varying the number of threads per block changes the multiprocessor occupancy (the ratio of the number of warps running on the multiprocessor to the maximum number of warps that can physically run on the multiprocessor). The CUDA profiler provides information about the multiprocessor occupancy. The number of threads per block should remain a multiple of 32 to facilitate coalescing and sufficiently large, typically greater than or equal to 192.

Instruction-level Optimization involves the use of fast math functions and Reduced Conditional Statements (RCS). The use of fast math results in fewer clock cycles for the instruction at the expense of reduced accuracy. The compiler optimization –use_fast_math forces compiling arithmetic functions as fast math functions. RCS reduces divergent paths taken within a warp since divergent paths are serialized resulting in reduced performance.

The three GPGPU implementations presented constitute a hierarchy where optimization techniques are successively added to the implementation [112]. All of the implementations in the hierarchy use the execution configuration optimization where a block-size is chosen that promotes maximum performance. Implementation 1 is the most basic implementation in the hierarchy that uses direct global memory accesses and software prefetching (SP).

Implementation 2 further enhances the global memory performance with the use of a block firing vector [116]. As mentioned in [116], the block firing vector acts as a collection of flags for a thread block and hence is blocksize magnitude smaller than the original firing vector (collection of flags for all the threads). 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 device to host (an improvement over Implementation 1 in data transfer overhead). This technique avoids unnecessary reads of the global firing vector by the host thereby reducing the overall application time. Section 4.3 detailed the concept of the block firing vector. CUDA Implementation 2 also uses Zero Copy (Z) to overlap host-device communication with the kernel computation and the cache-load scheme where global memory is cached using both L1 and L2 caches. Implementation 3 adds instruction level optimizations, including RCS and fast math functions.

5.3.1.1 CUDA Results

For each of the four SNN models, we present and contrast the performance of the three implementations developed by successively adding the common optimization techniques.

The performance for the Izhikevich model using CUDA is presented in Figure 5.6. Implementation 1 utilizes simple coherent global memory accesses to the device memory for computation. As explained earlier in Section 6.1, Implementation 1 requires transfer of the global firing vector from device to the host in each time-step. Thus this implementation is the least efficient of the three implementations with a speedup of 7.66x for the largest network size. Implementation 2 adds the concept of the block firing vector, which is blocksize order of magnitude smaller than the original firing vector. Consequently, the size of the data that must be transferred from the device to the host in each time-step is reduced by the order of blocksize. Additionally, the software time is reduced since the host is required to read the complete firing vector only if the block firing vector contains any firing information. Implementation 2 achieves a speedup of 13.2x for the largest network size. Implementation 3 of the Izhikevich model adds instruction level optimizations to reduce conditional statements and increase performance of mathematical functions. A maximum speedup of 17.1x was achieved for this implementation. A detailed study of these results using the profiler counter can be found in [112].

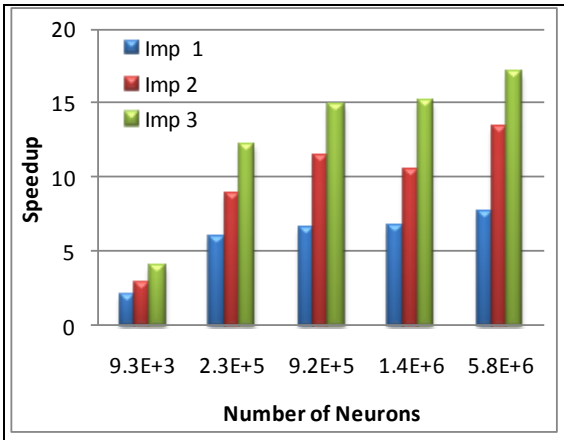

Figure 5.6. Speedup vs. network size for various optimizations: Izhikevich model on Nvidia Fermi using CUDA

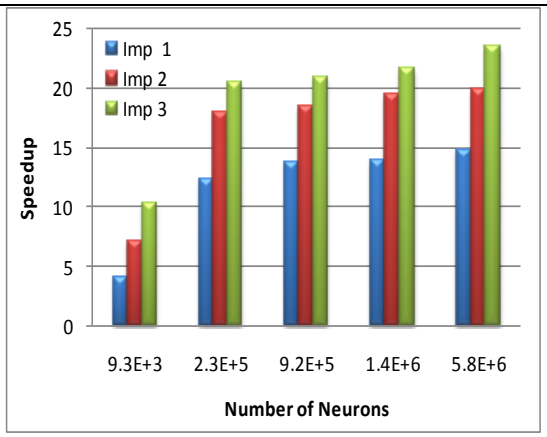

Figure 5.7. Speedup vs. network size for various optimizations: Wilson model on Nvidia Fermi using CUDA

Imp 1: Multithreading (MT) and Software Prefetching (SP)
Imp 2: Local Memory (LM) and Memory Write (MW)
Imp 3: Unsafe Math (UM), Native Math (NM), Reducing Conditional Statement (RCS)

Performance of the Wilson model using CUDA is presented in Figure 5.7. The Wilson model is communication bound and has a higher flops/byte ratio (1.52 vs 0.99 for the Izhikevich model). Implementation 1 of the Wilson model achieved a speedup of 14.7x for the largest network size, 5.8 million neurons, while Implementation 2 again improves performance with the efficient use of the block firing vector. The detailed study of this model using the CUDA profiler can be found in [112]. Further investigation with the CUDA profiler showed an improved instruction throughput, accounting for the improved performance. Implementation 3 has a performance gain over Implementation 2 due to the instruction level optimizations of the Wilson model.

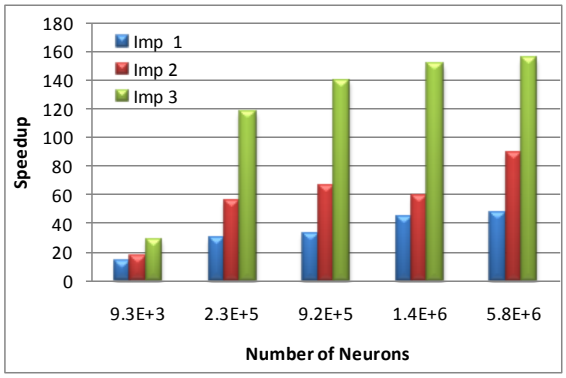

Figure 5.8. Speedup vs. network size for various optimizations: Morris-Lecar model on Nvidia Fermi using CUDA

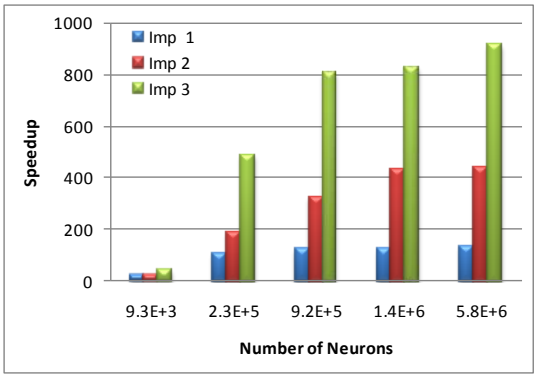

Figure 5.9. Speedup vs. network size for various optimizations: HH model on Nvidia Fermi using CUDA

Performance results for the three implementations of the Morris-Lecar model using CUDA are shown in Figure 5.8. Implementation 1 of the ML model achieves a speedup of 47.6x for the largest network size, while Implementation 2 improves performance and achieves a speedup of 89x using a block firing vector. Implementation 3 achieves a significantly higher speedup of 191x due to the increased computational requirements of the ML model.

The performance results for the three implementations of the Hodgkin-Huxley model using CUDA are shown in Figure 5.9. The HH model is the most computationally intensive of the four studied in this research, so it exhibited the largest performance gains from the various optimizations. Implementation 1 is the most basic of the three implementations with a speedup of 134x for the largest network size, while Implementation 2 achieves a speedup of 437x. Implementation 3 of the HH model adds instruction level optimizations. Since the HH model involves a considerable amount of computation (247 bytes per neuron update), the kernel extracts significant performance

benefits using RCS and fast math optimizations, giving Implementation 3 a maximum speedup of 919x.

The above explanations are summarized in Table 5.2. In this table we have qualitatively compared the performance characteristics of CUDA for the four neuron models. As seen in the table, the Izhikevich and Wilson models effectively utilize the memory level optimizations (communication parallelization) because of its low flop/byte ratio; thus, Implementation 2 provides the most performance gain. On the other hand, for the same reason, these two models cannot effectively utilize instruction level optimizations (computation parallelization), for which, Implementation 3 provides lower performance gain. The opposite is seen for the ML and HH models because of their high flop/byte ratio.

Table 5.2. Performance summary of CUDA on the Fermi GPGPU

Model	Flops/ Neuron	Byte /neuron	Flops /byte	Bound	Utilization of Communication Parallelization /Effectiveness	Utilization of Computation Parallelization /Effectiveness
Izhikevich	13	20	0.99	Memory	High/Low	Low/Low
Wilson	38	44	1.52	Memory	High/Low	Low/Low
Morris-Lecar	147	28	8.65	Computation	Low/Low	High/High
HH	246	44	9.84	Computation	High/Low	High/High

5.3.2 Intel Xeon

The Intel Xeon implementation of the Izhikevich model with the largest network, 5.8 million neurons, did not show significant performance improvement with the SSE and SP optimizations because of the low flop:byte ratio (see Figure 5.10). For the smaller network size, 0.9 million neurons, the speedup grows significantly as the optimizations

are added one by one. For the Wilson model with both the network sizes, SSE improves the performance significantly while SP provides only a slight improvement. SP has nominal effect in the Wilson model because the variables are only accessed a few times; SP can improve the performance significantly if the variables in a model are frequently accessed. For both network sizes of the ML model, the SSE techniques provide a significant speedup on the Intel Xeon architecture whereas SP provides only a nominal speedup (see Figure 5.11). Similar to the Wilson model, the ML model only accesses the variables a few times. On the other hand, for the HH model, all of the optimization techniques provide significant speedup.

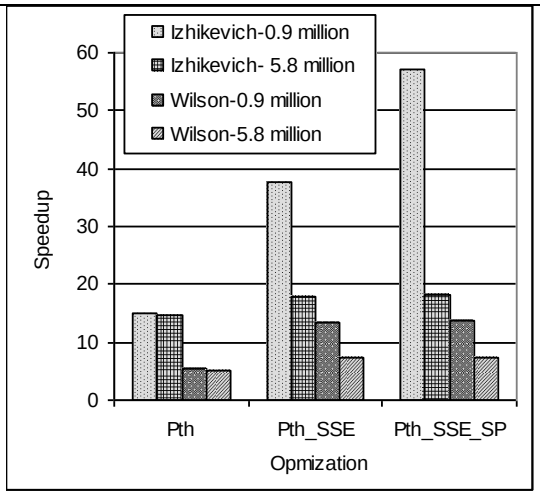

Figure 5.10. Intel Xeon speedup (8 cores) with different optimization techniques applied for the Izhikevich and Wilson models

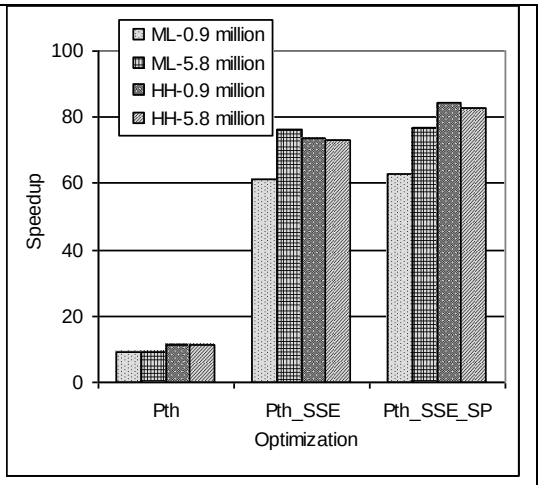

Figure 5.11. Intel Xeon speedup (8 cores) with different optimization techniques applied for the Morris Lecar and Hodgkin-Huxley models

(pth: POSIX thread, SSE: Streaming SIMD Extension 3, SP: Software Prefetching)

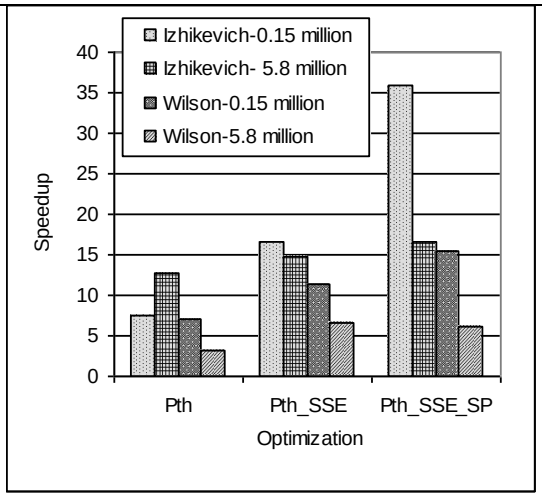

Figure 5.12. AMD Opteron speedup (8 cores) with different optimization techniques applied for the Izhikevich and Wilson models

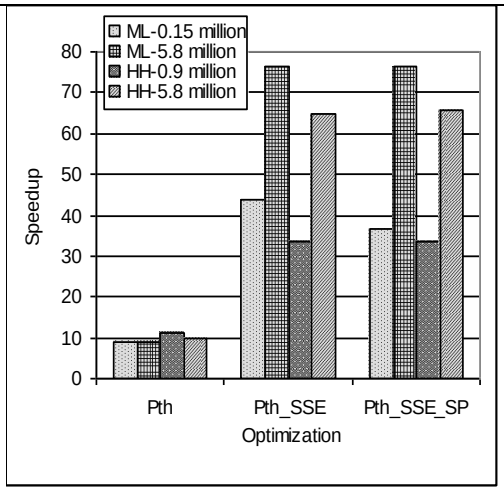

Figure 5.13. AMD Opteron speedup (8 cores) with different optimization techniques applied for the Morris-Lecar and Hodgkin-Huxley model

5.3.3 AMD Opteron

For the AMD Opteron, the same optimization techniques used for the Intel Xeon were applied as shown in Figures 5.12 and 5.13. For the smaller network using the Izhikevich and Wilson models, all of the optimizations provide significant speedup; while in the largest network, the improvement is nominal. For the largest network, the effect of cache misses halts further performance gain and thus, the effect of the optimization is negated.

For the ML and HH models, the pthreading technique and SSE optimization has the most effect on performance for both network sizes as shown in Figure 5.13. SP does not have a significant effect because the ML model does not access the variables many times.

5.3.4 Cell BE (PS3)

In Figure 5.14, the performance with available optimization techniques for the PS3 using the Izhikevich and Wilson neuron models is shown. From the figure, we observe

that for the Izhikevich model (for both network sizes of 5.8 million and 2.8 million neurons), multithreading and SIMD computation techniques contribute the most to the performance. Other optimization techniques do not have a significant effect on the performance because the communication time dominates the computation as detailed in the Section 5.2. For the Wilson model, the effect of optimization is only studied for the 2.8 million neurons. For this size, most of the performance gain comes from the MT and SIMD computation; other optimization techniques have a nominal effect on the performance because this implementation is also heavily dominated by the data communication time. For the largest network (5.8 million neurons), the PS3 did not provide any speedup and thus we do not report any effect of optimization for this network size.

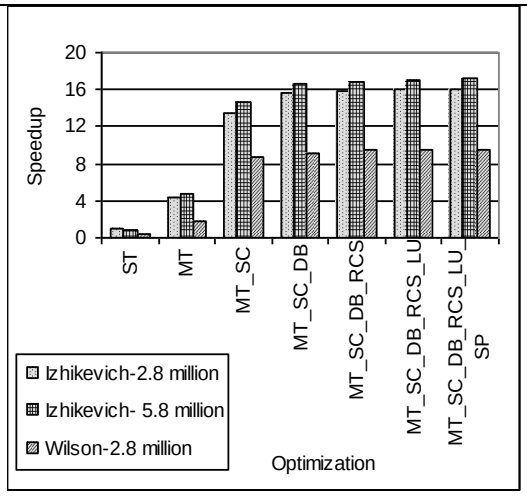

Figure 5.14. PS3 speedup (6 SPUs) with different optimization techniques applied for Izhikevich and Wilson model (Note: Only 2.8 million is shown for Wilson model as 5.8 million did not produce any speedup)

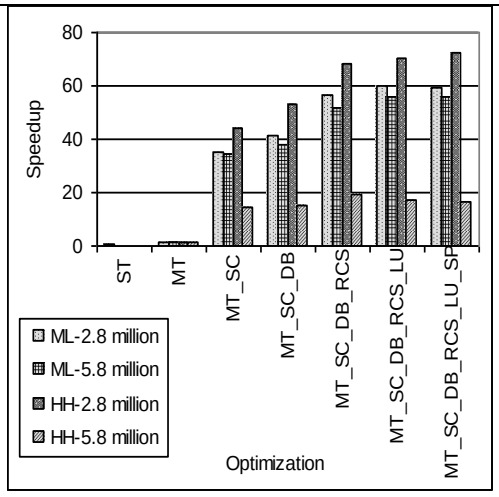

Figure 5.15. PS3 speedup (6 SPUs) with different optimization techniques applied for Morris-Lecar and Hodgkin-Huxley models

(ST: Single Thread, MT: Multithread; SC: SIMD computation, DB: Double Buffering, RCS: Reduced Conditional Statement, LU: Loop Unrolling, SP: Software Pipelining)

For the ML model with both network sizes (Figure 5.15), all of the optimization techniques except SP contribute to the performance. SP does not contribute much to the performance because the variables are not accessed many times as discussed in Section 5.2. For the HH model with a network size of 2.8 million neurons, the performance increases with each additional optimization technique. But for 5.8 million neurons, the loop unrolling (LU) and software pipelining (SP) techniques reduce the speedup, because for these optimizations, more instructions must be accommodated in the local store of the SPU resulting in more data transfer (i.e. more communication and overhead). Thus the speedup improvement achieved by LU and SP is negated by the increased communication cost. For both the HH and ML models, the speedup for the largest network is less than the smaller network due to DRAM misses, as discussed in the architecture performance sub-section.

5.4 Impact of Accelerator Configuration

The impact of accelerator configuration on performance is useful for parallel program development for GPGPU and multicore architectures. We have investigated the effect of accelerator configuration, such as the addition of cores or threads, on performance. Though it is generally expected that the addition of cores to an implementation will improve performance, cases discussed in the following section prove that this is not always true.

5.4.1 GPGPU

Changing the threads per block in can alter the execution configuration of the Fermi GPGPU and as a result, change the multiprocessor occupancy. But a higher occupancy does not always imply better performance, as the performance is influenced by many factors. However, higher occupancy can help hide memory latency of the multiprocessor execution in most cases. We have investigated the performance of the GPGPU implementation for all possible block sizes. We have only reported the top six performers with their associated block sizes (128 through 1024) in Table 5.3. The performance seen in this table is achieved with all of the memory and instruction level optimizations discussed in the previous section. From this table we see that the block size of 192 or 256 provide better performance. Actually, the effect of the block size is dependent upon the device architecture, features of the applications, and the optimization techniques applied. Thus for other GPGPU devices from NVIDIA or ATI and for different applications, the developer should investigate the best configuration for a particular application.

Table 5.3. Fermi GPGPU: Speedup with varying local work group size for four models with network size of 5.8 million neurons

Block size	Izhikevich	Wilson	ML	HH
128	14.31	19.99	185.79	919.34
192	14.28	20.11	188.31	942.11
256	14.41	20.05	187.84	945.07
512	14.40	20.06	186.12	892.30
960	14.21	19.77	178.86	802.73
1024	14.20	19.81	179.39	813.53

5.4.2 X86 (Intel Xeon And AMD Opteron)

Two network sizes are selected to study the performance while varying the number of processing cores for the Intel Xeon and AMD Opteron. These network sizes are the same as those used in the optimization techniques study in the previous Section, 5.2. Figure 5.16 shows the performance variation resulting from the addition of processing cores on the Xeon for both the Izhikevich and Wilson models. The figure shows that the performance of the Izhikevich model with a network size of 0.9 million neurons increases almost proportionally with the addition of processing cores. But for a network size of 5.8 million, the scaling is not proportional and after 2 processing cores, there is essentially no gain from additional cores. The primary reason for this behavior is the increased communication time compared to computation time; thus the performance gained by the addition of processing cores is negated by the increased communication time. For the Wilson model with the smaller network size, adding processing cores provides noticeable performance improvement except for the 6th core. This anomaly is due to the increased data communication for the Wilson model and the use of 6 cores further increases the communication overhead as information must be exchanged across two separate processors. On the other hand, the compute intensive ML and HH models (shown in Figure 5.17) scale well with the addition of processing cores for both network sizes because these two models have a higher flops/byte ratio than the Izhikevich and Wilson models. Thus, the flops of the ML and HH models are parallelized more effectively across all the cores.

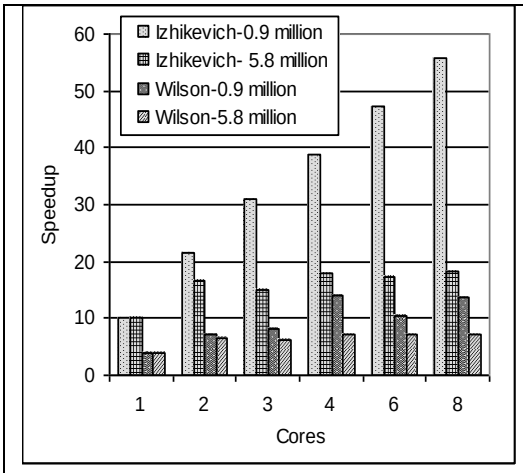

Figure 5.16. Intel Xeon speedup for increasing number of cores for the Izhikevich and Wilson models

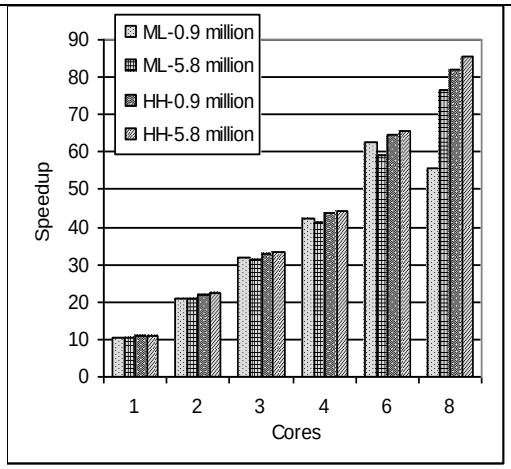

Figure 5.17. Intel Xeon speedup for increasing number of cores for the Morris-Lecar and Hodgkin-Huxley models

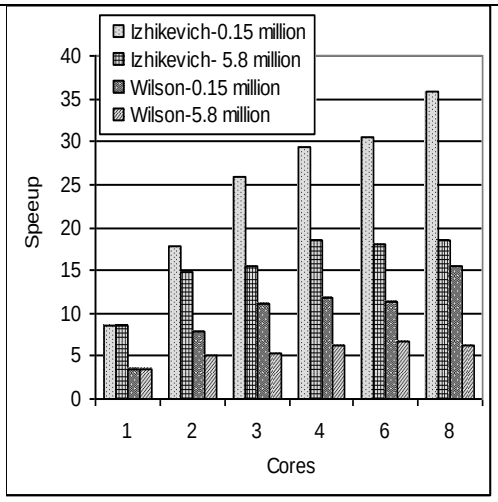

Figure 5.18. AMD Opteron speedup for increasing number of cores for the Izhikevich and Wilson models

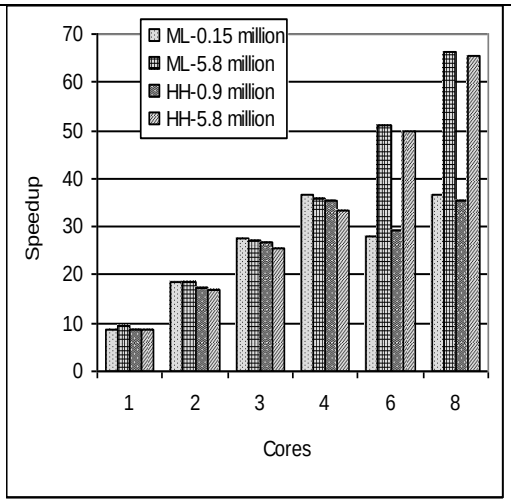

Figure 5.19. AMD Opteron speedup for increasing number of cores for the Izhikevich and Wilson models

For the AMD Opteron, we have reported results for the Izhikevich and Wilson models for a network size of 0.15 million, where the best performance occurs, and for the largest network size, 5.8 million in Figure 5.18. The performance as cores are added is

similar to that for the Xeon and the same explanation can be applied. For the ML and HH models (Figure 5.19), except for the smaller image size with the HH model, we find a similar trend to that of the Intel Xeon and the same explanation can also be applied. For the smaller network size with the HH model, the speedup decreases beyond 4 cores due to the additional communication overhead.

5.4.3 PS3

Figure 5.20 shows the performance variation for the PS3 architecture as the number of SPUs is varied for the Izhikevich and Wilson neuron models. In this study, the network sizes are the same as those used in the optimization techniques study for the PS3 (5.2). In the figure we see that the Izhikevich model with network sizes 5.8 million and 2.8 million, moving from 1 SPU to 2 SPUs, the speedup increases significantly; beyond 2 SPUs, the speedup grows marginally. This behavior is attributed to the increased communication and overhead time for the models, which cannot be parallelized by adding SPUs. For the Wilson model, the variation of speedup with the cores is only shown for the case of 2.8 million neurons because the speedup for the other size is very low and no variation is noticed when adding processing cores. It is found that after 3 cores, the speedup does not increase with of the addition of more cores. This model is communication bound and the addition of processing cores does not help to reduce the communication time.

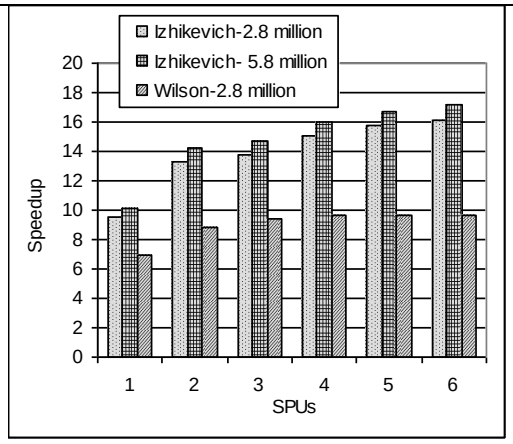

Figure 5.20. PS3 speedup for increasing number of SPUs for the Izhikevich and Wilson models

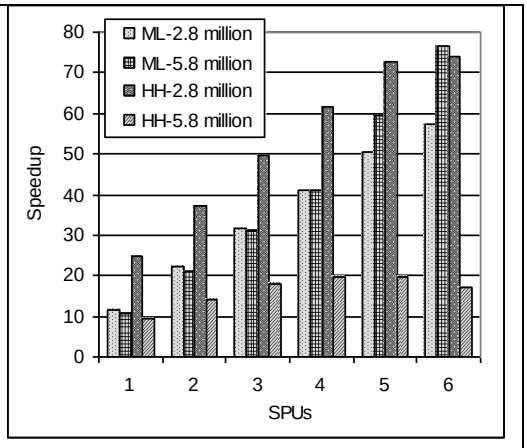

Figure 5.21. PS3 speedup for increasing number of SPUs for the Morris-Lecar and Hodgkin-Huxley models

For the ML model (Figure 5.21), adding PS3 processing cores provides almost linear speedup. The ML model requires more flops than the previous two models and has approximately the same amount of data transfer as the Izhikevich model. For the HH model with a network size of 2.8 million, it is observed that the addition of SPUs causes the speedup to grow almost proportionately. For the largest network size, 5.8 million, the communication time increases more than the computation time and the resulting speedup is slightly lower than that of the smaller network sizes. It is also notable that the speedup with 6 SPUs is slightly lower than that for 4 or 5 SPUs. This change most likely occurs due to the congestion in the EIB of the PS3. The EIB has only four lanes to transfer data between the SPU and DRAM. So when 6 SPUs issue a DMA request for data from the DRAM, the request processing time is increased due to the number and size of requests. Thus, repeated DMA requests will be issued, further increasing the communication time and eventually negatively impacting the performance.

5.5 Advanced Multicore and GPGPU Performance

Advanced multicores such as Intel 32-core, AMD 32-core, SUN T2 UltraSparc 32-core, and IBM PowerXCell based QS22 16-core were also studied in this research. In this section, we discuss the implementation of the four SNN models on these advanced architectures and report the performance results. For the advanced multicores, we focus on the performance variation of the architectures with the number of threads. These performance results will be used during development of the regression performance models for the advanced multicore architectures in Chapter 7.

5.5.1 Intel 32 Core

With early access to the Intel 32-core processors at the Intel Many Core Testing Lab, we have run the four SNN models for a network size of 9.7 million neurons. The performance of the four models as the number of threads increases is presented in Figure 5.22. The processor supports a multi-threads per core configuration. Thus a maximum of 64 threads were created to accelerate the SNN models. From Figure 5.22, we see that the performance of all four models increases until the number of threads reaches 32, where there are more threads than cores.

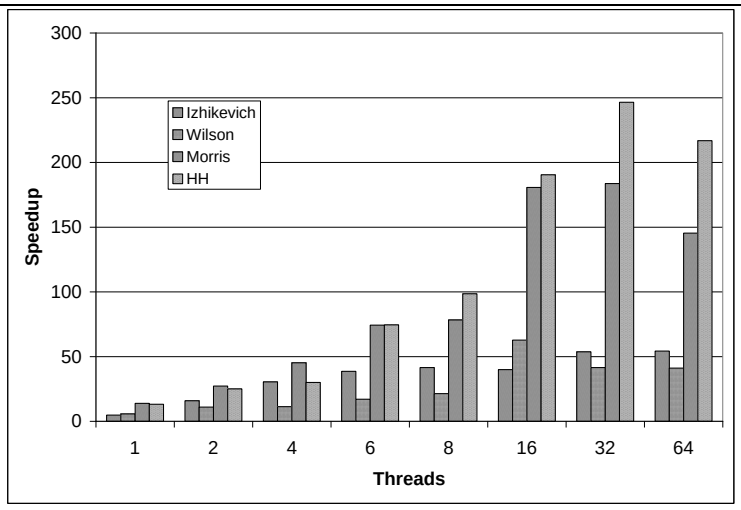

Figure 5.22. Speedup of Intel 32-core with varying number of threads for all four models

5.5.2 AMD 32 Core

The performance of the AMD 32-core processor with varying number of cores is shown in Figure 5.23. The performance of the Morris-Lecar and HH models is higher than that of the Izhikevich and Wilson models. Also, except for the Wilson model, the performance increases with an increase in the number of threads until 32, where the number of threads is equal to the number of cores.

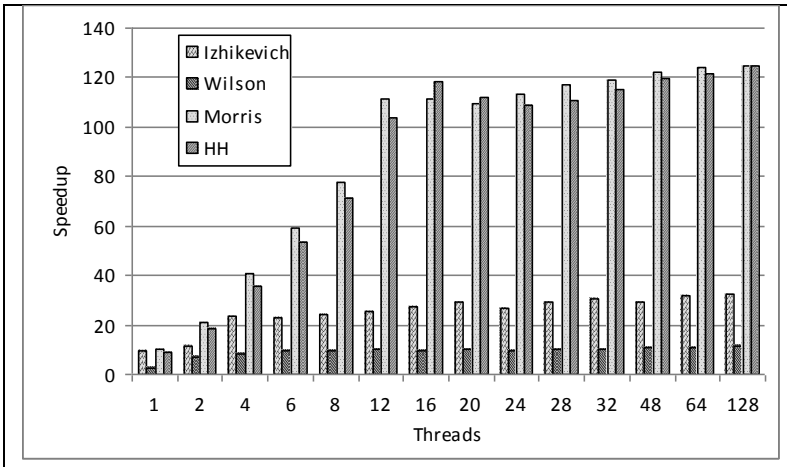

Figure 5.23. Speedup of AMD 32-core with varying number of threads for all four models

5.5.3 SUN T2+ UltraSPARC (16 Core)

The performance of the SUN T2+ UltraSPARC with varying number of threads is shown in Figure 5.24. The performance for the Morris-Lecar and HH models is higher than that of the Izhikevich and Wilson models. Also, the performance increases with

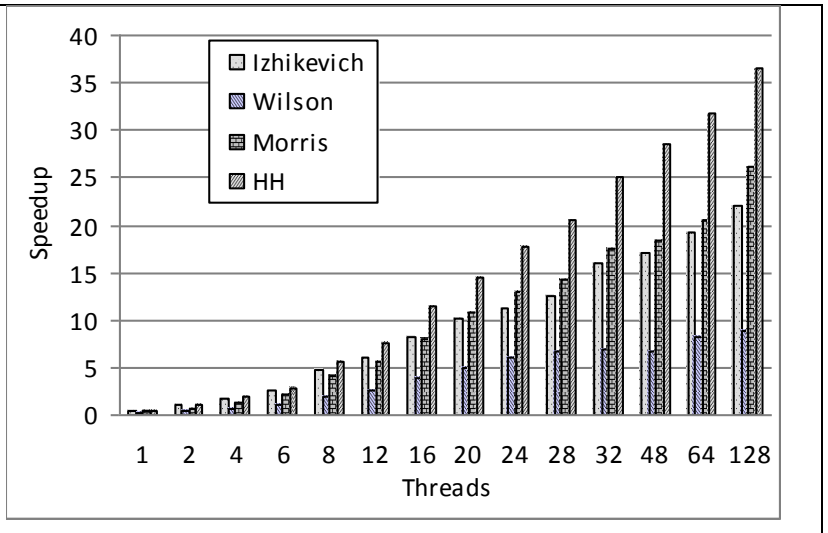

Figure 5.24. Speedup of SUN 16-core with varying number of threads for all four models

increasing number of threads until the thread count reaches 128, even though there are only 16 cores. The maximum speedup achieved from the SUN UltraSparc is 37x, which is far lower than that of the Intel or AMD many-cores but the performance increases rather than saturates until the maximum number of threads, 128, which did not happen for the Intel or AMD many-core architectures.

5.5.4 IBM PowerXCell 8i-based QS22 (16 Core)

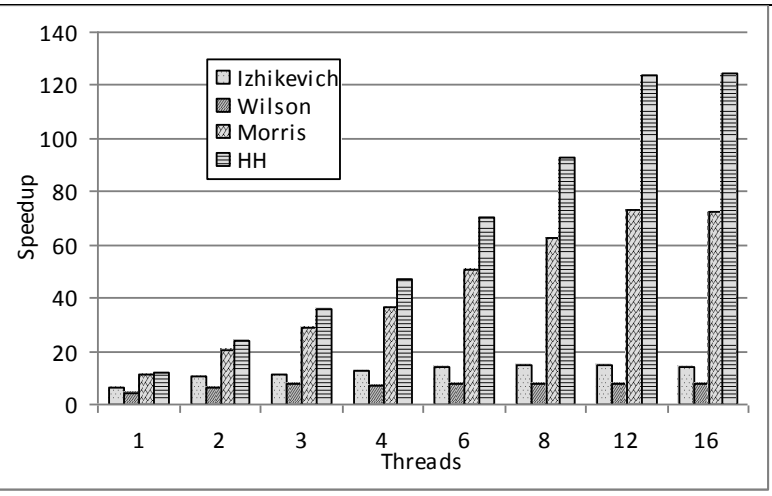

Figure 5.25. Speedup of QS22 16-core with varying number of threads for all four models

The performance of the QS22 with varying number of cores is shown in Figure 5.25. The performance for the Morris-Lecar and HH models is much higher than that of the Izhikevich and Wilson models. Also, the performance for the Izhikevich and Wilson models does not increase with the increase of threads after 2 cores. The reasons for this observation are similar to those discussed for the PS3. For the Morris-Lecar and HH models, the performance increases with the increase of threads. The maximum speedup

achieved from QS22 is 125x, similar to the Intel or AMD many-core architectures. For the QS22, threads more than the total number of cores cannot be created. Thus we could not verify if the performance changes with more threads than cores.

5.6 Summary

In summary, the experimental results in this chapter provide insight on various aspects of the performance of accelerator architectures. We have analyzed the performance variation of base set of four accelerators by varying the problem size of the applications, optimization techniques, and accelerator configurations. The effect of threading technique on advanced multicore processors is also investigated. The effect of software tools (such as POSIX-threading vs. OpenCL, OpenCL vs. CUDA) on the performance of accelerators is discussed next in Chapter 6. The results and analysis from this chapter and those of Chapter 6 will be used to develop the *Fitness* model and regression models for the impact of programming models on performance in Chapter 7. The performance models are then verified in Chapter 8, based on the data presented in Chapters 5 and 6.

Chapter 6

Programming Model Impact On Performance

The programming model used in algorithm development for accelerator architectures can have a positive or negative effect on performance. In this section, we investigate the effect of a) CUDA and OpenCL for GPGPU implementation, and b) POSIX-threading, OpenMP, Concurrency Runtime, and OpenCL for multicore implementation. First, we will describe the GPGPU programming models and then, we will discuss the x86 programming models.

6.1 GPU Programming Models

Two of the most populare and widely used GPGPU programing models are: CUDA and OpenCL. We have analyzed the effect of both the programming models on Nvidia Fermi GPGPU in this section. The resulting performance of the SNN case study was found to be better with the CUDA implementation than the OpenCL implementation on the NVIDIA Fermi. A maximum performance of 976.2x was observed for the largest network size (3120 x 3120) with the CUDA implementation, whereas the OpenCL implementation only achieved 878.4x for the same network size. Turning ECC off yielded a speedup of 1095x with the CUDA implementation and 1074x with the OpenCL implementation, for the largest network size (3120 x 3120).

6.1.1 CUDA

The implementations of SNN using CUDA programming models on Nvidia Fermi is detailed in Section 5.3.1. The performance analysis of Nvidia with CUDA for various optimizations is also reported in the same section. Thus, for the description of CUDA results we refer to that section.

6.1.2 OpenCL

There are minor differences between the CUDA and OpenCL implementations and optimizations for the SNN models. OpenCL provides several optimization techniques that can be employed for optimal performance. These techniques can be classified as Memory-Level and Instruction-Level optimizations. The optimization techniques used in this study are, a) *Multithreading* (MT), b) *Software Prefetching* (SP), c) *Local Memory* (LM), d) *Memory Write* (MW), e) *Native Math* (NM), f) *Unsafe Math* (UM), and g) *Reducing Conditional Statement* (RCS).

While some of the optimizations for OpenCL are similar to CUDA, some are specific to the particular programming model. In this study, the *Zero Copy* technique and cache preference settings were applied for CUDA implementations but were unavailable with the OpenCL 1.0 implementation from Nvidia. On the other hand, the use of the MW optimization is only used with OpenCL. All other optimization techniques for OpenCL are similar to those used with CUDA.

For OpenCL, the *Memory-Level Optimizations* include SP, the use of MW, and the use of LM. The SP technique is similar to that of CUDA. In the OpenCL

implementation, while sending the array from the host to the GPGPU, the clCreateBuffer() function was used to create a buffer and copy data from the host to the GPGPU. The MW optimization creates a buffer in the initialization part of the application and then the function clMemWrite() is used inside the execution loop of the host function, achieving better performance. The technique of using LM is similar to using *Shared Memory* with CUDA as described in the sub-section 4.4.2 and shown in Figures 4.1 and 4.2.

Instruction-Level Optimizations for OpenCL provide NM functions that are optimized versions of exponentials, trigonometric, and other complex functions. These native functions provide improved speed with some loss of precision. These functions are to be used cautiously so that any errors in precision do not negatively impact the overall results. Another performance improvement technique used with OpenCL is called UM optimizations where the compiler optimizes floating-point arithmetic in the kernel, but this process may violate the IEEE 754 standard and OpenCL numerical compliance requirements. Therefore, this technique should also be used cautiously. The last applied optimization is RCS, also used with CUDA.

6.1.2.1 Results

We now discuss the performance of the four SNN models with three OpenCL implementations developed by successively adding the optimization techniques. In *Implementation 1*, MT and SP were applied. *Implementation 2* includes two memory-

level optimizations: LM and MW. The three remaining instruction-level optimizations, UM, NM, and RCS, are used in *Implementation 3*.

The performance results of the Izhikevich model using OpenCL are given in Figure 6.1. As seen in the figure, all implementations have a significant impact on performance. *Implementation 1* achieves a speedup of 7.3x for the largest network size, while *Implementation 2* adds memory optimizations for a maximum speedup of 10.2x. *Implementation 3,* which uses all instruction-level optimizations, shows further improvement in kernel performance with a speedup of 15.1x. As seen in the graphs, the maximum speedup (15.1x) for the Izhikevich model with OpenCL is slightly lower than that for CUDA (17.1x). The maximum performance for the Izhikevich model is the lowest among all models because it is a computation bound model, as discussed earlier in this chapter.

The performance results for the Wilson model using OpenCL are presented in Figure 6.2. This model is also communication bound with a flop/byte ratio of 1.52, similar to the Izhikevich model. Thus the impact of the three implementations for this model is expected to be similar to the Izhikevich model results. All implementations provide significant performance improvement. Most notably, the speedup of *Implementation 3* for the largest network size is 21.3x, slightly lower than the CUDA speedup for the same model (23.5x).

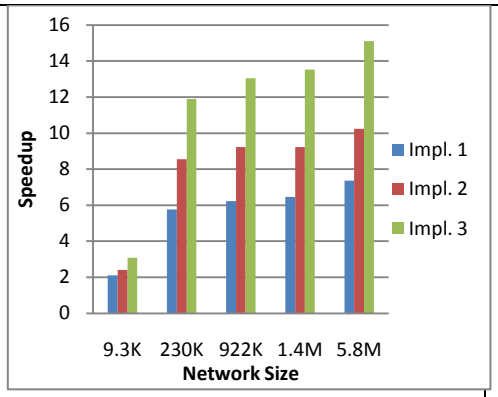

Figure 6.1. Speedup vs. network size for various optimizations: Izhikevich model on Nvidia Fermi using OpenCL

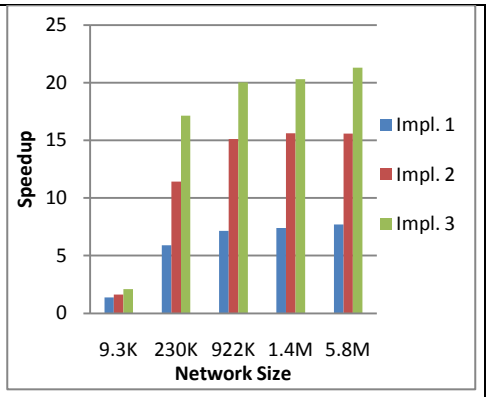

Figure 6.2. Speedup vs. network size for various optimizations: Wilson model on Nvidia Fermi using OpenCL

Impl. 1: Multithreading (MT) and Software Prefetching (SP)
Impl. 2: Local Memory (LM) and Memory Write (MW)
Impl. 3: Unsafe Math (UM), Native Math (NM), Reducing Conditional Statement (RCS)

Performance for the ML model with OpenCL for the ML model is shown Figure 6.3. As can be seen from the figure, performance for this model is much higher than the previous two models (145x vs. 21.3x for the Wilson model). Similar to CUDA, this observation can be attributed to the higher flop/byte ratio of the ML model (Table 5.1 of Chapter 5). Though all the implementations provide noticeable improvement in performance, *Implementation 3* provides the best performance. This result is expected, since the ML model uses several transcendental math functions that are optimized and accelerated using the instruction-level optimizations like UM and NM. Compared to CUDA, the maximum speedup of OpenCL for the ML model is slightly lower, 145x with OpenCL vs. 155x with CUDA.

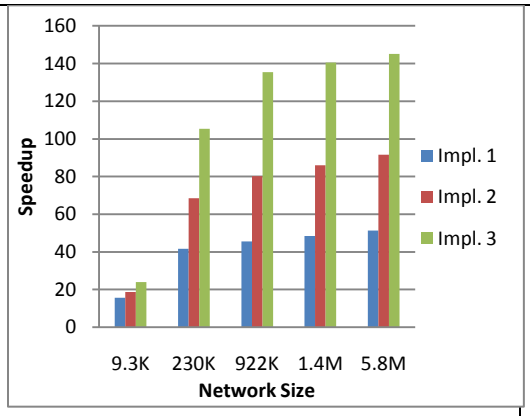

Figure 6.3. Speedup vs. network size for various optimizations: Morris-Lecar model on Nvidia Fermi using OpenCL

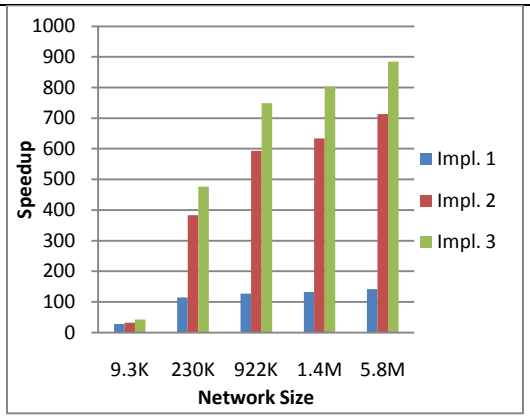

Figure 6.4. Speedup vs. network size for various optimizations: HH model on Nvidia Fermi using OpenCL

The performance results for the three implementations of the HH model are shown in Figure 6.5. Significant performance gains with the HH model are achieved with OpenCL: 884x for the largest network, which is the highest among the four SNN models. The highest speedup for the HH model with OpenCL, 884x, is however slightly lower than that with CUDA, 918.6x. *Implementation 2* provides the best performance gains among all implementations. *Implementation 2* utilizes mainly LM optimizations that reduce accesses to global memory and data transfers between host and device. Since the HH model requires an order of magnitude more simulation cycles than the other models (roughly 400 cycles compared to about 30 cycles for the Wilson model), reducing the data transfer in each cycle contributes significantly to performance. *Implementation 3* also provides noticeable performance gain because of the acceleration of transcendental functions in this model. Analysis of the profiler results can be found in [112].

The observations are similar to those summarized in Table 5.2 of Chapter 5 for the CUDA programming model. We see that the performance gain for *Implementation 3* of the HH model (i.e., instruction-level optimizations) with OpenCL is significantly lower than that with CUDA. The instruction-level optimizations, specifically NM and FM optimizations provide more performance gain in CUDA than in OpenCL because CUDA, a native language for Fermi, takes full advantage of the Fermi math hardware as opposed to OpenCL, which is a generalized language for multiple GPGPU and CPU architectures.

6.2 Multicore Programming Models

In this section we will discuss the effect of four multicore-based programming models on the performance of Intel x86 processor, Intel i7.

6.2.1 POSIX-threading

On Intel architectures, three types of optimizations are applied with POSIX-threading: 1) POSIX-threading with multiple thread calls (pthread_multiple); 2) POSIX-threading with a single thread call (pthread_single); and 3) Software Prefetching (SP). Pthread_multiple is the base optimization that creates threads and distributes the computations among them. In this optimization technique, the thread creation is invoked in every cycle of the simulation resulting in an overhead for creating and joining p-threads. In the second optimization, pthread_single, threads are created only once instead of every cycle to minimize this overhead, and each thread runs multiple simulation

cycles. Essentially, the loops (one for the cycle and the other for multiple thread creation) are swapped to reduce the thread overhead. Finally, in the third optimization (SP), data in arrays that are used multiple times are loaded into the registers, operated on, and stored back into the array at the end.

6.2.1.1 Results

The performance from POSIX-threading, also referred to as p-threading, for varying problem (network) sizes and optimization techniques on the Intel i7 processor are discussed in this sub-section. The results are organized based on the SNN models.

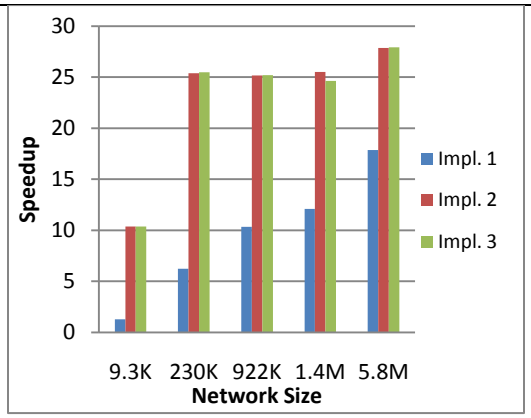

Figure 6.5. Speedup vs. network size for various optimizations: Izhikevich model on Intel i7 using p-threading

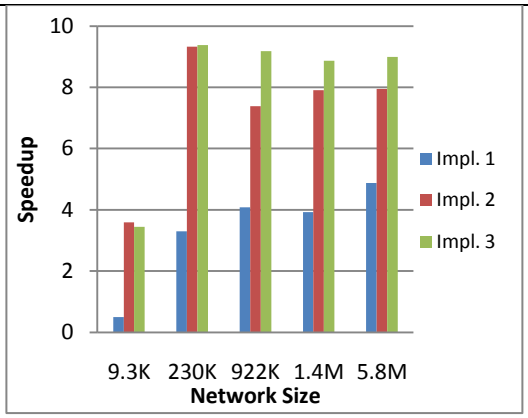

Figure 6.6. Speedup vs. network size for various optimizations: Wilson model on Intel i7 using p-threading

Impl. 1: p-threading with multiple time thread call (pthread_multiple)
Impl. 2: p-threading with single time thread call (pthread_multiple)
Impl. 3: Software Prefetching

Figure 6.5 shows the performance of the Izhikevich model with p-threading on an Intel i7 system based on problem network size and optimization technique.

Implementation 1 provides increasing speedup with network size but clearly does not take full advantage of parallelism. Implementation 2 avoids thread creation and joining in every simulation cycle, which saves significant overhead and provides a large gain over its predecessor (from a maximum speedup of 17.8x to 27.9x). Implementation 3 does not provide additional performance for the Izhikevich model because the algorithm requires only two arrays per neuron, which are reused only a few times, so there is little advantage to adding SP.

It should be noted that the speedup for the largest image size is 28x, which is more than 8x improvement with the 8-core Intel i7, i.e., super linear. With additional threads, the performance increases in two ways: a) data communication is parallelized with the presence of a level 1 cache in each core and b) computation is parallelized by the processor in each core. There is also overhead involved with the parallelization of the application that negatively affects the performance. Therefore speedup results only if the time saved by the parallelization of communication and computation is greater than the overhead time of parallelization. For the Izhikevich model, the overhead is negligible when the problem size is fairly large as seen in Figure 6.5. Additionally, this model is memory bound with a flop/byte ratio of 0.99, so parallelization of communication provides the best performance. This improvement mainly comes from the cache structure of each processing core and thus is difficult to quantify. The achieved performance depends on many factors such as the memory access pattern of the application, DRAM characteristics, cache hierarchy, cache size, cache line size, associatively, cache replacement policy, etc.

Figure 6.6 shows the performance of the Wilson model with p-threading on an Intel i7 system. The maximum performance for the Wilson model is slightly lower than that of the Izhikevich model (9.4x vs. 27.9x) due to the data intensive characteristics of the Wilson model. Though both models are memory bound, the number of memory accesses is much higher in the Wilson model than the Izhikevich model (44 bytes vs. 20 bytes), and thus it has more cache misses than the Izhikevich model. Even though the Wilson model has more computations per neuron (38 flops vs. 13 flops), this does not have a significant effect on performance. Peak performance is achieved around the network size of 230 thousand neurons, a result of the cache size effect. Implementation 2 provides the largest performance impact, just like with the Izhikevich model. Unlike the previous model, though, Implementation 3 provides slightly better performance for larger problem sizes. Since the Wilson model utilizes more arrays (five) that are accessed multiple times, the implementation gains noticeable benefit from SP.

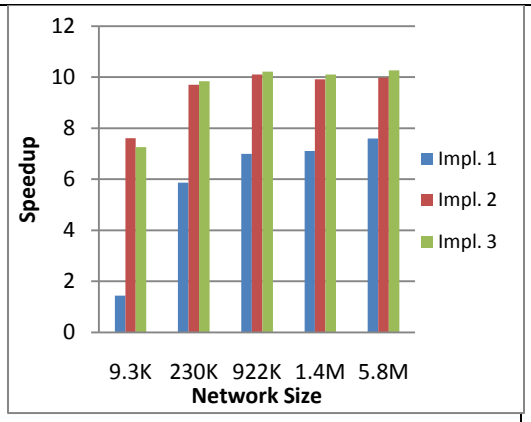

Figure 6.7. Speedup vs. network size for various optimizations: Morris-Lecar model on Intel i7 using p-threading

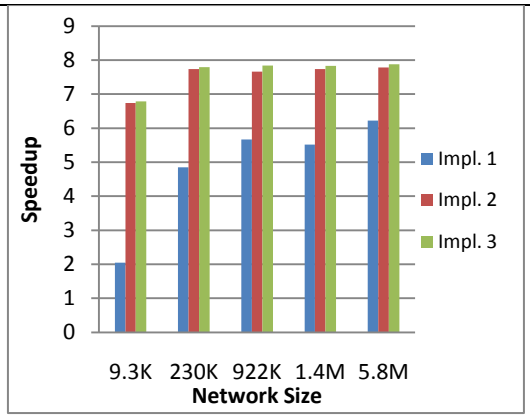

Figure 6.8. Speedup vs. network size for various optimizations: Hodgkin-Huxley model on Intel i7 using p-threading

Figure 6.7 shows the performance of the Morris-Lecar model with p-threading on an Intel i7 system. As seen in the figure, the maximum performance for the ML model is much lower than for the Izhikevich model (10.1x vs. 27.9x). Implementation 2 provides a noticeable improvement over the initial implementation, but Implementation 3 shows very little difference. The main reason for this lower speedup is that the ML model is computationally bound with a flop/byte ratio of 8.65, so most performance gain will come from the parallelization of computation, which is a maximum of 8x from the processor's 8 cores. The parallelization of communication through the cache does contribute to performance but does not have a significant impact as it did with previous models.

Figure 6.8 shows the performance of the HH model with p-threading on an Intel i7 system. The maximum performance is lower than the other three models (7.9x vs. 27.9x for the Izhikevich model) but similar to the ML model. The major difference with the HH model is the amount of data accessed per neuron is higher than the Izhikevich and ML models (44 bytes vs. 20 bytes for the Izhikevich model). Thus the HH implementation cannot exploit the parallelization of communication to the same degree as the Izhikevich model.

We have summarized the performance trends of the neuron models with p-threading on the Intel i7 in Table 6.1. As seen from the table, the performance with p-threading is the highest for the Izhikevich model because of the high effectiveness of communication parallelization since it is a memory bound model. On the other extreme, the performance

for the HH model is the lowest, though the utilization of computation is high because a maximum of 8x performance is plausible from the 8-core Intel i7.

Table 6.1. Performance summary of p-threading on the Intel i7 processor

Model	Flops/ Neuron	Byte /neuron	Flops /byte	Bound	Utilization of Communication Parallelization /Effectiveness	Utilization of Computation Parallelization /Effectiveness
Izhikevich	13	20	0.99	Memory	High/High	High/Low
Wilson	38	44	1.52	Memory	Medium/Medium	High/Low
Morris-Lecar	147	28	8.65	Computation	Medium/Low	High/High
HH	246	44	9.84	Computation	Low/Low	High/High

6.2.2 OpenMP Implementation

Both function-level and data-level parallelism can be achieved using the OpenMP model depending on the program's design. Independent sections of code can be separated and run concurrently or a single block, most commonly the iterations of a for-loop, can be divided among multiple threads. OpenMP has provided significant performance increases in multicore systems running computationally intensive code that can be partitioned into individual computation threads while sharing a single memory space, including matrix multiplication and graph matching algorithms.

The SNN models discussed in this paper typically involve a large number of level 1 neuron calculations that are all independent, so this portion of the code was split among 8 threads to parallelize the computation and send the final result to a shared firing vector with information for each neuron. After each time step of the algorithm, this level 1 neuron firing vector was used to update the set of level 2 neurons. Since the OpenMP API is based on a shared memory architecture, options for data sharing and synchronization were used throughout the implementations to control public and private

data. It was necessary for each thread to read in neuron information from shared memory, maintain its own values for intermediate variables, and then write back to the shared firing vector for level 2 neuron calculations.

Two optimizations were made to the code to improve the initial performance of the OpenMP implementation. First, a dynamic thread scheduler directive was used to improve efficiency and decrease processor idle time. Rather than dividing the entire section of parallel work initially and waiting for all threads to finish execution, *dynamic load balancing* (DLB) allows for the work to be divided into smaller pieces and assigned to threads as they complete their allotted work. This task division introduces additional overhead to the program but ensures minimal processor idle time. Second, *fast math* (FM) optimizations were introduced at the compiler level to accelerate the mathematical functions used in each algorithm (most importantly, the exponentiation function). Care must be taken with this optimization however, because some of the safeguards of the IEEE floating point standard are removed, resulting in performance improvement under the assumption that hazardous data is not used as input to the operation.

6.2.2.1 Results

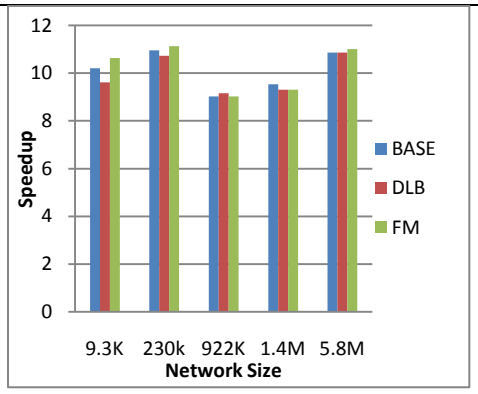

Figure 6.9. Speedup vs. network size for various optimizations: Izhikevich model on Intel i7 using OpenMP

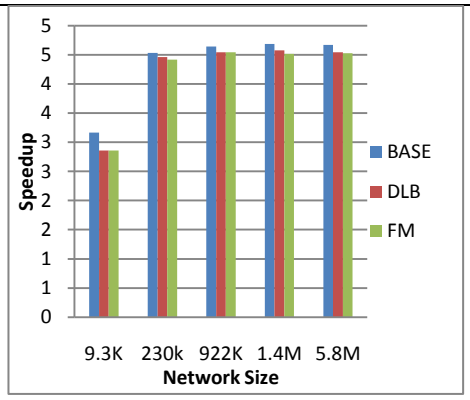

Figure 6.10. Speedup vs. network size for various optimizations: Wilson model on Intel i7 using OpenMP

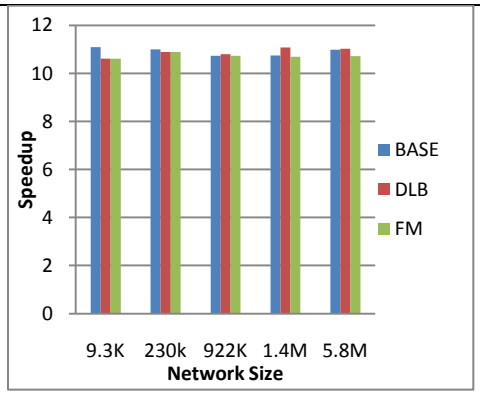

Figure 6.11. Speedup vs. network size for various optimizations: Morris-Lecar model on Intel i7 using OpenMP

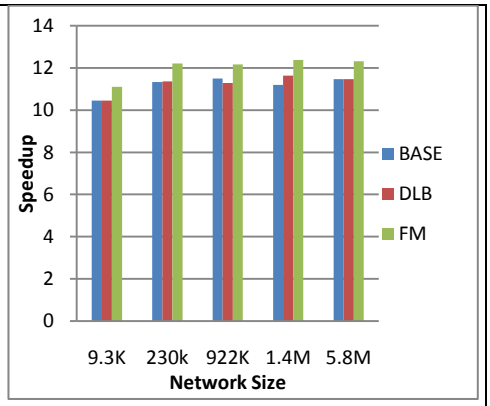

Figure 6.12. Speedup vs. network size for various optimizations: HH model on Intel i7 using OpenMP

The results show significant speedup when using OpenMP with 8 cores and threads, particularly with the computationally intensive Morris-Lecar and Hodgkin-Huxley models. The basic implementation of the Morris-Lecar algorithm exhibited a speedup of at least 10.8x with a maximum of 11.0x for the largest image size. The Hodgkin-Huxley

model followed a similar pattern with a minimum speedup of 10.2x for the smallest image size gradually rising to a maximum speedup of 11.6x for the largest image size.

The Izhikevich and Wilson models also exhibited a consistent speedup, but these models had a lower flop/memory access ratio so the cores were unable to achieve high efficiencies. The results of the unoptimized Wilson model ranged from 3.2x to 4.7x while the Izhikevich algorithm exhibited speedup of 9.2x to 11.1x. Synchronization time and memory sharing constraints were more significant with these models because of their shorter run-times, so their speedups were lower. Although the largest block of computationally intensive code was parallelized, any speedup was limited by Amdahl's Law since larger portions of the code are still serial because of data dependencies and critical sections.

The results from optimizations were mixed overall and very dependent on which model and image sizes were being used. For the two least computationally intensive models, Izhikevich and Wilson, DLB actually showed a slight decrease in performance from a speedup of 10.9x to 10.8x and 4.6x to 4.4x, respectively. These two algorithms do not appear to have a significant enough issue with load balancing to warrant the additional overhead of thread scheduling. In the intermediate case of the Morris-Lecar model, speedup was unchanged at 11.1x. For the Hodgkin-Huxley model, though, there was an increase of performance for a speedup of 11.7x from 11.6x. It should be noted that for all models, DLB performance generally improved as the image size grew. This result is consistent with the goal of thread scheduling since a larger image size allows for

a greater potential disparity between running times if the workload is not balanced dynamically.

A pattern also emerged in the results for the FM optimizations. For the first three models, there was virtually no trend or overall improvement of the performance since these models appear to be more constrained by memory sharing than any other factor. For the computationally intensive Hodgkin-Huxley model, though, there was a significant improvement in every image size giving an overall average speedup of 12.0x compared to 11.2x for dynamic thread scheduling alone. When running tests with an image size of over 1 million, speedup of 12.3x was achieved.

Table 6.2 provides a summary of the performance of OpenMP for the four SNN models. The utilization of parallel communication is low for these models while parallel computation is high. The memory bound models (Wilson and Izhikevich) exhibited some performance increase but not as significant as others. OpenMP provided its best performance for the computation bound models (Morris-Lecar and HH), which is consistent with these findings.

Table 6.2. Performance summary of OpenMP for the four neuron models on an Intel i7 processor

Model	Flops/ Neuron	Byte /neuron	Flops /byte	Bound	Utilization of Communication Parallelization /Effectiveness	Utilization of Computation Parallelization /Effectiveness
Izhikevich	13	20	0.99	Memory	Low/High	High/Low
Wilson	38	44	1.52	Memory	Low/Low	High/Low
Morris-Lecar	147	28	8.65	Computation	Low/Low	High/High
HH	246	44	9.84	Computation	Low/Low	High/High

6.2.3 Visual Studio Threading Technique (Concurrency Runtime)

The basic parallelization techniques used with the Concurrency Runtime programming model are similar to the techniques used with OpenMP. A template pattern provided with the Concurrency Runtime, called parallel_for, was used to parallelize the loops in both levels of the network. Parallelization of level 1 and level 2 calculations is a task-level parallelization that also exploits the work-stealing algorithm. Though level 2 computations are less than 5% of the total computation time, the tasks of both levels are parallelized and balanced with the work-stealing algorithm. With p-threading and OpenMP, the parallelization of the two levels would hurt performance since the work-stealing algorithm does not exist with these programming models. Thus Concurrency Runtime has more potential than the previous two programming models to exploit both the task-level and data-level parallelism without issues of load balancing.

Four implementations were used to study the performance of Concurrency Runtime for the four neuron models: a) *loop parallelization using a lock* b) *loop parallelization without a lock* c) *software prefetching* (SP) d) *parallelization of both the level 1 and level 2 neurons*. In *Implementation 1*, loop parallelization using a lock, only the level 1 neuron computations are parallelized. In this implementation, a lock is used before updating the count variable for level 1 neuron firing and only the index of a firing neuron is stored in the firing vector. The lock causes overhead and thus degrades performance. *Implementation 2* uses loop parallelization without a lock, the lock was removed by altering the algorithm. In this optimization, instead of counting the total number of firing neurons and storing the index of these neurons, each position of the firing vector is

upgraded with the firing information of each neuron. After this optimization, the amount of computation increases slightly since the whole firing vector must be checked for firing information. The need for a lock is removed at the price of an increase in computation. In *Implementation 3*, the SP technique is used. The effect of SP in Concurrency Runtime is similar to that for p-threading. In *Implementation 4*, the level 1 and level 2 neurons are computed in parallel as previously described.

6.2.3.1 Results

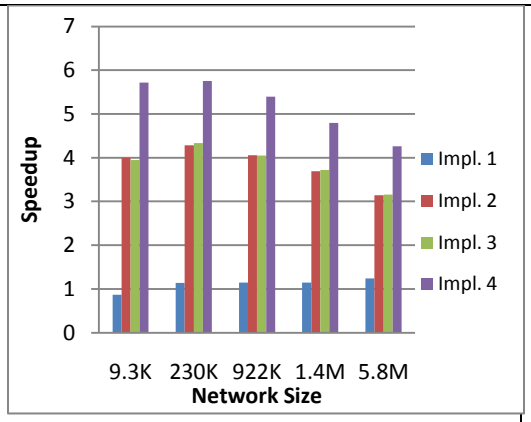

Figure 6.13. Speedup vs. network size for various optimizations: Izhikevich model on Intel i7 using Concurrency Runtime

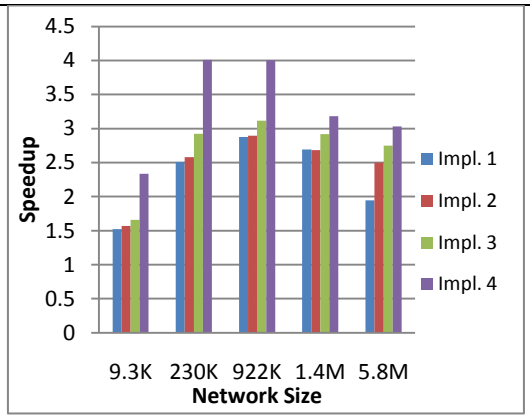

Figure 6.14. Speedup vs. network size for various optimizations: Wilson model on Intel i7 using Concurrency Runtime

Figure 6.13 shows the performance of the Izhikevich model with Concurrency Runtime on an Intel i7 system. The maximum speedup for this model is 5.8x. Since the Izhikevich model is memory bound, a programming model that can parallelize the data communication will provide better performance. Although the Intel i7 provides individual level 1 cache for each processing core, it is clear from the figure that Concurrency Runtime cannot take full advantage of this hardware parallelism. In

contrast, *p-threading* was able to take full advantage of cache-level parallelism and provided a maximum speedup of 27x. Two performance trends are visible in Figure 6.13. First, the performance declines with the increase in problem size due to the increased number of cache misses given that the Izhikevich model is a memory bound model. Second, *Implementation 2* and *Implementation 4* provide the best performance. *Implementation 2* removes the need for a lock at the cost of an increase in computation; the lock is significantly expensive in Concurrency Runtime. In *Implementation 4*, level 1 and level 2 are parallelized together using the Concurrency Runtime work-stealing algorithm.

Figure 6.14 shows the performance of the Wilson model with Concurrency Runtime. The maximum performance for this model is about 3x, which is lower than the Izhikevich model (27x). The reason for lower performance is attributed to the larger byte transfer requirement (44 bytes per neuron vs. 20 bytes per neuron for the Izhikevich model). The scheduling algorithm for Concurrency Runtime cannot take full advantage of the cache-level parallelism and thus the performance is worse than the Izhikevich model. Two trends can be seen from the figure. First, a maximum in the performance is seen due to the cache size of the Intel i7. Second, all of the optimizations could not provide significant performance improvement, which is likely because the Wilson model is memory bound with a flop/byte ratio of 1.52 and a larger data access requirement. As discussed earlier with *p-threading*, the data communication time dominates any improvement gained by the optimization techniques.

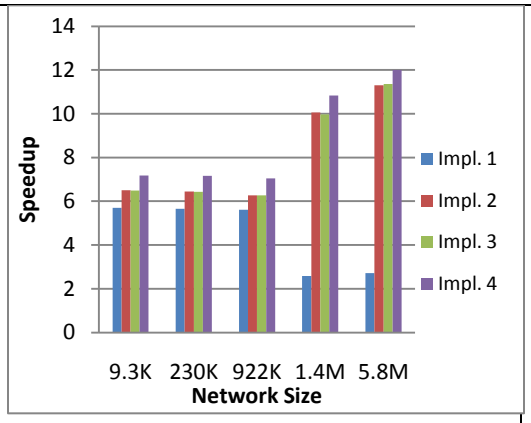

Figure 6.15. Speedup vs. network size for various optimizations: Morris-Lecar model on Intel i7 using Concurrency Runtime

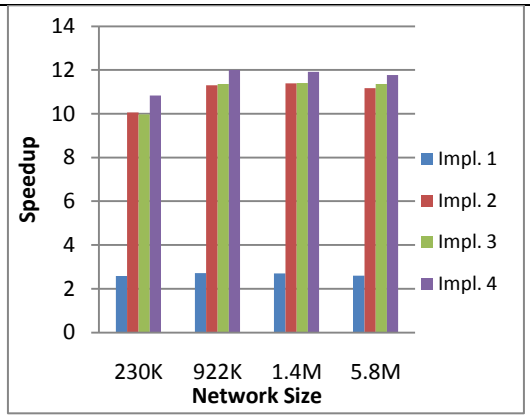

Figure 6.16. Speedup vs. network size for various optimizations: Hodgkin-Huxley model on Intel i7 using Concurrency Runtime

Figure 6.15 shows the performance of the Morris-Lecar model with Concurrency Runtime. In this case, a maximum speedup of 12x is notably higher than the previous two models. This model is a computationally bound model (flop/byte ratio of 8.62 vs. 0.99 for the Izhikevich model) and Concurrency Runtime is able to efficiently divide the computation among the processing cores. There are three noticeable trends in the results. First, the performance with *Implementation 1* decreases with an increase in problem size due to the overhead associated with the use of a lock. Second, the performance of the other implementations improve as the problem size grows. Larger problem sizes produce more computations, which are efficiently mapped onto the computing resources available in the Intel i7. Third, no maximum is observed in the performance because this model is computationally bound, and thus does not suffer from the cache size effects seen in the previous two models.

Figure 6.16 shows the performance of the HH model with Concurrency Runtime. The maximum performance for this model is 12x. The HH model is the most computationally

intensive model (flop/byte ratio of 9.84) and thus Concurrency Runtime can efficiently map the computation of the model onto the computing resources. The performance trends for the HH model are similar to those for the Morris-Lecar model.

The performance of Concurrency Runtime for the four neuron models is summarized in Table 6.3. As can be seen from the table, the utilization of parallel communication is low for the memory bound models (Izhikevich and Wilson), which is why the performance of these models is lower than the performance with p-threading. On the other hand, the utilization of parallel computation for the computation bound models (Morris-Lecar and HH) is high, and thus their performances are higher than the performances of the Izhikevich and Wilson models. However, they are still approximately the same as with p-threading and OpenMP because the computation performance is limited by the number of processing cores.

Table 6.3. Performance summary of Concurrency Runtime for the four neuron models on an Intel i7 processor

Model	Flops/ Neuron	Byte /neuron	Flops /byte	Bound	Utilization of Communication Parallelization /Effectiveness	Utilization of Computation Parallelization /Effectiveness
Izhikevich	13	20	0.99	Memory	Low/High	High/Low
Wilson	38	44	1.52	Memory	Low/Medium	High/Low
Morris-Lecar	147	28	8.65	Computation	Medium/Low	High/High
HH	246	44	9.84	Computation	Low/Low	High/High

6.2.4 OpenCL Implementation

The OpenCL implementation of the SNN models on an Intel i7 system is very similar to the implementation on an Nvidia Fermi GPGPU with some minor differences. One optimization, *vectorization*, was not supported on the GPGPU but was available and

applied on the Intel i7. Another difference is that the OpenCL driver and compiler for the Fermi GPGPU were provided by Nvidia while for the Intel i7 system, these tools are provided by AMD.

The optimization techniques used in this work are, 1) *Multithreading* (MT), 2) *Software Prefetching* (SP), 3) *Local Memory* (LM), 4) *Memory Write* (MW), 5) *Native Math* (NM), 6) *Unsafe Math* (UM), 7) *Reducing Conditional Statements* (RCS), and 8) *Vectorization* (Vec). Except for Vec, which operates on four variables simultaneously to improve performance, all other optimizations are similar to those applied for the OpenCL Nvidia Fermi implementation described in Section 6.1.

6.2.4.1 Results

Figure 6.17 shows the performance for the Izhikevich model with OpenCL using an Intel i7 system. The optimizations provide noticeable performance improvement as the problem size grows, with the best incremental performance gain from SM (4.6x to 8.3x for the largest network size). The SP and UM optimizations showed the least impact since there are not many computations (only 13 flops/neuron) and the model is memory bound. Vec yields significant performance gain because of the gains in efficiency discussed above, providing a maximum speedup of 9.8x.

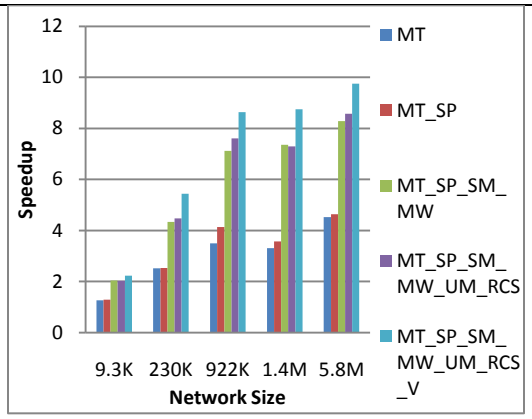

Figure 6.17. Izhikevich model on an Intel i7 using OpenCL

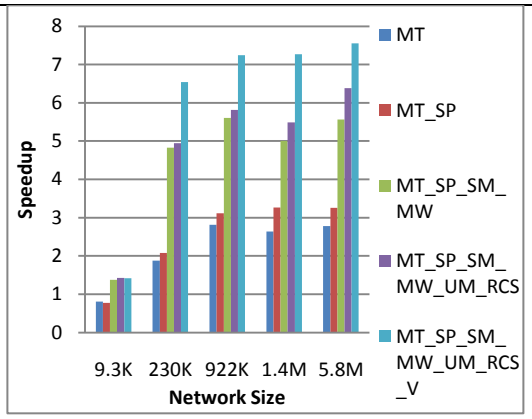

Figure 6.18. Wilson model on an Intel i7 using OpenCL

Figure 6.18 shows the performance for the Wilson model with OpenCL. The optimizations show a trend similar to that of the Izhikevich model and thus the same explanations can be applied. Using the largest network size, speedup values of 2.8x-7.6x for the Wilson model were consistently lower than the 4.5x-9.8x achieved with the Izhikevich model, which can be attributed to the larger data access requirement (44 bytes vs. 20 bytes) for the Wilson model.

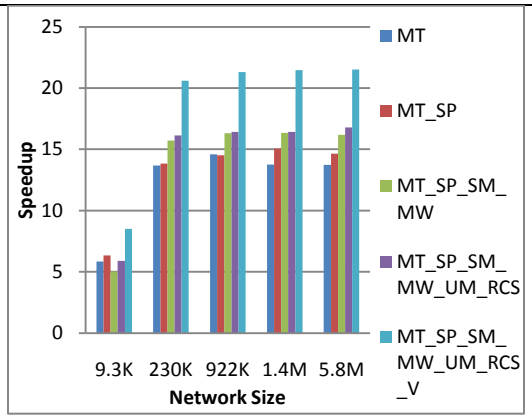

Figure 6.19. Morris-Lecar model on an Intel i7 using OpenCL

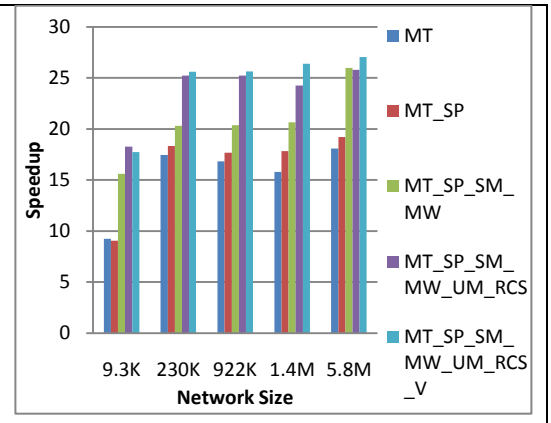

Figure 6.20. Hodgkin-Huxley model on an Intel i7 using OpenCL

Performance for the Morris-Lecar model with OpenCL, as shown in Figure 6.19, is better than the previous two models with a maximum of 21.5x. Morris-Lecar is a computationally intense model as previously discussed, and OpenCL is able to parallelize the calculations efficiently. Performance increases steadily with added optimizations and increased problem size. The Vec implementation provides a substantial performance improvement (from 16.8x to 21.5x speedup for the largest network size) due to the computational nature of the model, while the other optimizations had a significantly smaller impact.

Performance for the most computationally intensive HH model with OpenCL is the highest among the four neuron models, as shown in Figure 6.20. Due to their shared characteristics, the HH and ML models exhibit very similar performance trends with consistent improvement as the network size grows and each optimization is added. The NM and UM optimizations make a notably large impact on performance due to the heavy computational load of the HH model, providing a maximum speedup of 25.8x and 27.0x with the addition of Vec.

The results for OpenCL can be summarized similarly to those of Concurrency Runtime. Table 6.3 also can be applied for the summary of discussion of the OpenCL implementation.

6.3 Comparative Study of the Programming Models

In this section, a comprehensive comparative analysis of the results is presented. In Table 6.4, prominent features of the five programming models are summarized. Among

these programming models, OpenMP and Concurrency Runtime do not require any explicit thread creation. Thus from the developer's point of view, the level of abstraction of these two programming models is high, since the developer does not manage the low-level APIs. The other three models ,p-threading, CUDA and OpenCL, require explicitly creating threads along with the function routines executed by these threads. All programming models studied for the multicore processors except OpenCL support both SIMD and MIMD types of parallelism, allowing each of the cores in the multicore processor to run with an independent program counter when desired. Thus in our study, p-threading, OpenMP, and Concurrency Runtime support both SIMD and MIMD types of parallel execution and do not require lockstep style execution to be maintained. On the other hand, the GPGPU-based programming models, OpenCL and CUDA, maintain strictly lockstep style execution where all GPGPU cores execute instructions following a single program counter.

Another important feature, vectorization, is only supported by OpenCL. Most multicore processors, such as the Intel i7, and some GPGPU devices from AMD have support for vectorization. This feature makes OpenCL suitable for both GPGPU and multicore systems to take advantage of the vector units available in hardware. The other four programming models do not support API calls for vectorization. However, although SSE is not a part of the multicore-based programming models (p-threading, OpenMP, and Concurrency Runtime), SSE instructions can be used together with these programming models to exploit vectorization.

The last feature listed in the table is the programming effort required by the developer for a programming model. Three levels of effort, low, medium, and high, are reported for the programming models based on the subjective experiences of the authors. As seen in the table, OpenMP and Concurrency Runtime require low programming effort as explicit thread creation are not necessary and minimal changes to the serial code are required for parallelization; thus, they can be considered very developer-friendly programming models. The p-threading technique requires medium programming effort since it requires explicit thread creation but not other low-level hardware control such as specifying the device memory types to use or explicit data transfers between host and device. On the other hand, OpenCL and CUDA require high programming effort since low level API calls must be used (for example, explicit data transfer within the memory hierarchy) along with thread creation while using these programming models.

Table 6.4. Prominent features of programming models

	Explicit thread creation	Abstraction level	Parallelization form	Strictly lockstep style execution	Vector-ization	Programming effort (subjective)
p-threading (multicore)	Yes	Low	SIMD, MIMD	No	No	Medium
OpenMP (multicore)	No	High	SIMD, MIMD	No	No	Low
Concurrency Runtime (multicore)	No	High	SIMD, MIMD	No	No	Low
OpenCL (multicore & GPU)	Yes	Low	SIMD	Yes	Yes	High
CUDA (GPU)	Yes	Low	SIMD	Yes	No	High

The amount of programming effort is hard to measure quantitatively since it is a qualitative component. However, we have attempted to capture a rough estimate of the

programming effort to compare each of the programming models. In Tables 6.5 and 6.6, various parameters related to programming effort are listed. Table 6.5 shows the additional lines of code required to parallelize the application with each of the programming models. As seen in the table, the rank of the programming models is listed according to the average additional lines of codes; the smallest number of additional lines of code corresponds to the lowest rank. Assuming the programming effort is roughly proportional to the amount of additional lines of codes, OpenMP is the most developer friendly programming model requiring the least programming effort whereas OpenCL requires the most programming effort.

To further investigate the programming effort required by the programming models, additional implementation characteristics of the SNN models are listed in Table 6.6. Since every programming model requires using its own API in addition to a base programming language (in this case, C/C++), the number of different API calls used in the SNN implementation is listed in the table. The OpenCL SNN implementation uses 25 different API calls whereas Concurrency Runtime only requires 2 different API calls. Further, p-threading and OpenMP required fewer optimizations (only three), whereas CUDA and OpenCL required seven optimizations to achieve their highest performance. For OpenMP, it was not necessary to write additional functions as required for other programming models. The last component listed in the table is Explicit Thread Creation, which is required for all programming models except OpenMP and Concurrency Runtime.

Table 6.5. Comparing programming effort in terms of additional lines of code required for the five programming models

	Izhikevich	Wilson	Morris-Lecar	HH	Average	Rank
p-threading (multicore)	39	37	28	23	32	2
OpenMP (multicore)	26	27	24	17	24	1
Concurrency Runtime (multicore)	51	51	58	54	54	3
OpenCL (multicore and GPU)	136	187	187	175	171	5
CUDA (GPU)	117	158	167	110	138	4

Table 6.6. Comparing programming effort in terms of the increase of complexity of the programming models on Intel i7 and Nvidia GPU

	SNN implementations			
Programming model	Different types of API calls used	Optimizations	Additional Functions	Explicit Thread Creation
p-threading (Multicore)	6	3	1	Yes
OpenMP (Multicore)	5	3	0	No
Concurrency Runtime (Multicore)	2	4	2	No
OpenCL (Multicore & GPU)	25	7	1	Yes
CUDA (GPU)	12	7	1	Yes

Observing the values that are derived from the four SNN implementations and listed in Tables 6.5 and 6.6, the qualitative amount of programming effort required for the multicore architecture can be listed from low to high as: 1) OpenMP 2) p-threading 3) Concurrency Runtime and 4) OpenCL. Similarly, the ranked list for the GPGPU is 1) CUDA and 2) OpenCL.

To summarize the performance of the SNN case study with the five programming models, we have listed the highest speedup values in Table 6.7. As shown in the table for the Intel i7 multicore processor, p-threading provides higher performance for

applications with a low flop/byte ratio, such as the Izhikevich and Wilson models, whereas OpenCL provides the highest performance for applications with higher flop/byte ratios, such as the Morris-Lecar and HH models. The performance of OpenMP and Concurrency Runtime falls between p-threading and OpenCL. The performance of OpenMP is better than Concurrency Runtime and consistent across various applications with a range of low to high flop/byte ratios. Thus, we cannot rank the programming models in terms of their performance across all types of applications as we did in terms of programming effort. Therefore to select a programming model for an application on a given set of hardware, one must consider the application characteristics as well as the development time. For example, if the time to develop is open-ended and the application is computationally bound, then OpenCL would be a good choice. On the other hand, if there is a strict time constraint for parallel application development and the goal is to achieve moderate application performance, then OpenMP would be the best choice.

For Nvidia's Fermi GPGPU, CUDA consistently performs better than OpenCL for all SNN models in the application study. If code portability is not an issue, CUDA would always be the best choice for Nvidia GPGPUs since the performance and programming effort are both in favor of CUDA as shown in Tables 6.5-6.7.

Table 6.7. Comparing application performance for the four programming models on Intel i7 and Nvidia Fermi

Platform		Izhikevich	Wilson	Morris-Lecar	HH
Intel i7	p-threading	27.90	8.99	10.26	7.87
	OpenMP	11	4.5	10.8	12.5
	Concurrency Runtime	4.26	3.03	11.92	11.89
	OpenCL	9.75	7.55	21.52	27.04
Nvidia Fermi	CUDA	17.09	22.82	188.26	976.2
	OpenCL	12.10	18.04	142.05	878.4

6.4 Summary

In this chapter, we have critically studied four of the most popular programming models for x86 multicore processors (p-threading, OpenMP, Concurrency Runtime, and OpenCL) and two of the available GPGPU programming models (CUDA and OpenCL) for an Nvidia GPGPU. These results will be used to develop multiple regression models of the performance impact of programming models on accelerators. The programming models are compared in terms of performance and programming effort using four most biologically accurate SNN models (HH, ML, Wilson, and Izhikevich). Various implementations have been developed based on hand-tuned successive optimizations applied to all the programming models. The effects of optimizations as well as problem size scaling were critically studied; the advantages and limitations of each programming model were reported.

With the broad study conducted using a wide range of application complexity, multiple optimizations, and varying problem size, it was found that the programming models for the x86 processor cannot be ranked according to their achievable performance across all applications. For example, p-threading performs the best for one type of application whereas OpenCL is the best for another type. Thus the achievable performance with the programming models is found to be dependent on application characteristics. However, for the Nvidia GPGPU, CUDA implementations were found to perform consistently better than OpenCL across all SNN models. Conversely, programming models can be ranked in terms of perceived programming effort. Thus this study provides insights for a developer attempting to select a programming model for a

given pairing of application to architecture. A proper match among applications, accelerators, and programming models provides the best performance with minimal to moderate development effort. In the next chapter we will develop the Fitness model and multiple regression performance model based on the experimental results of Chapter 5 and 6.

Chapter 7

Performance Models

Many high-performance parallel architectures are available to accelerate scientific computing applications. These architectures have varying core counts, clock speeds, data bandwidth, cache hierarchies and sizes, memory sizes, etc. For example, a GPGPU device has a memory hierarchy with local (shared), texture, constant and global memories whereas an X86 processor has a cache hierarchy with level 1, 2 and 3 caches and DRAM. Conversely, the IBM CellBE has a completely different memory hierarchy with a DRAM and a local store in each SPU. From the application perspective, each has different computational and data bandwidth requirements. Thus it is very difficult for an application developer in high-performance computing to select the architecture that will offer the best performance for a particular application. Thus attempts have been taken by researchers to propose performance models of computing architectures for predicting application performance. We have discussed various performance models in Chapter 2 and we now propose a *Fitness* Performance model that can be used as an efficient way to predict the performance rank of various accelerators. In this chapter, the *Fitness* performance model is developed based on the performance analysis of various accelerators as discussed in Chapter 5. Then the *Fitness* model is extended using statistical regression. We have used multiple regression models to model the runtime of accelerators and impact of programming models. Along with the proposed *Fitness* model,

the Roofline model, another visual performance model from the literature, is also discussed in this chapter for comparison purposes. First, we will discuss the performance characteristics of architectures based on the projections of the Roofline model. Then we will develop the *Fitness* and regression models.

7.1 Performance Prediction With the Roofline Model

In Figure 7.1, the Roofline model of the Nvidia Fermi (C2050) GPGPU is shown. The position of the matrix multiplication and four SNN models is plotted in the framework as vertical lines drawn at the theoretical flop:byte ratio of each. As seen from the figure, the matrix-matrix multiplication, Izhikevich and Wilson models are communication bound since the corresponding lines intersect the slanted line in the plot. Conversely, the Morris-Lecar and Hodgkin-Huxley models are computation bound since their lines intersect the horizontal lines of the plot.

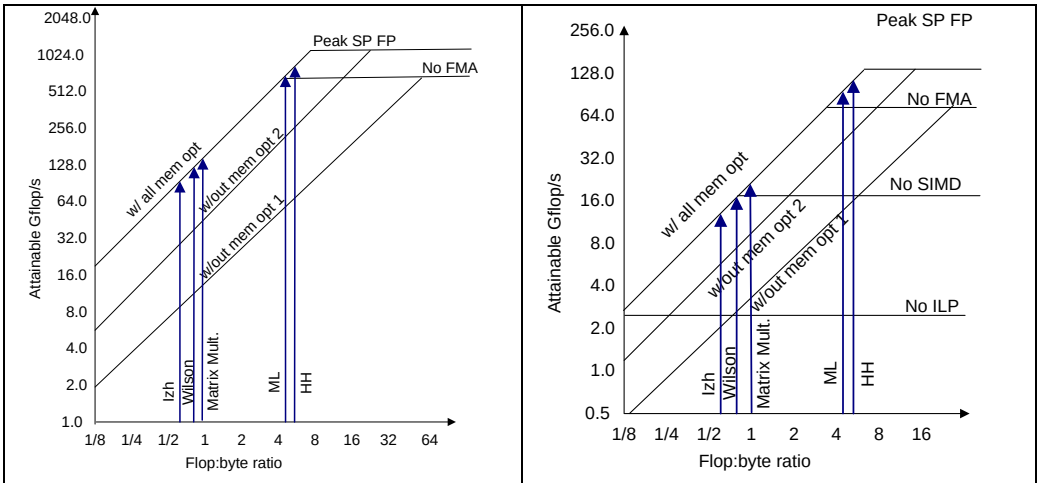

Figure 7.1. Roofline model for the matrix multiplication and four SNN models on the Fermi GPGPU

Figure 7.2. Roofline model for the matrix multiplication and four SNN models on the Xeon

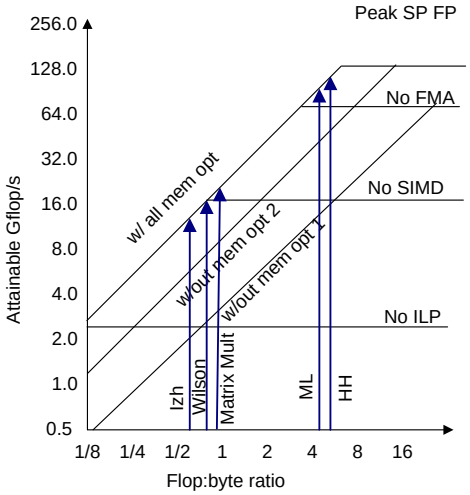

Figure 7.3. Roofline model for the matrix multiplication and four SNN models on the Opteron

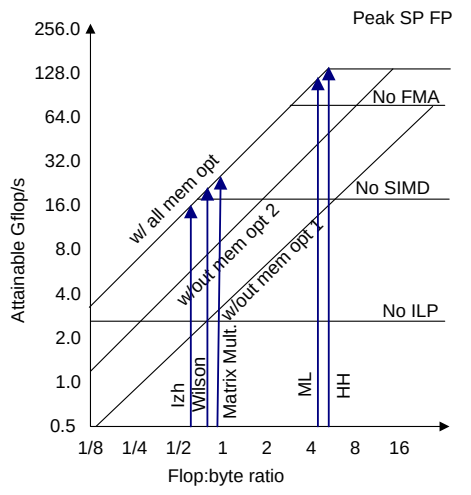

Figure 7.4. Roofline model for the matrix multiplication and four SNN models on the PS3

In Figure 7.2, the positions of matrix multiplication and all four SNN models are shown with the Roofline model for the Intel Xeon. Here it is found that matrix multiplication and all four models are communication bound as they intersect the slant ceiling. But the Morris-Lecar and Hodgkin-Huxley models have a higher level of attainable Gflops than the Izhikevich and Wilson models since they intersect at higher points on the sloped line. The vertical flop:byte lines that are shown for matrix multiplication and each SNN model represent the compulsory miss lines. The actual performance line may be shifted to the left since there will undoubtedly also be capacity and conflict misses of the cache as the problem size increases. Thus, the actual Gflop/s performance of all case studies will be lower than the theoretical values, but the relative achievable performance of the Intel Xeon for all case studies can be seen from the

Roofline model. A similar explanation can be applied for the Roofline model for the AMD Opteron and IBM PS3 as shown in Figure 7.3 and 7.4.

If we compare the performance prediction of Roofline model among GPGPU, Xeon, AMD, and PS3 we can easily find a trend. The GPGPU will perform the best for all the case studies while the other three architectures will have similar performance. The Roofline model is a coarse visual model that does not account for the problem size of applications and memory hierarchy of accelerators, on the other hand, the proposed *Fitness* model accounts for these characteristics. The *Fitness* model is detailed in the next section.

7.2 *Fitness* Performance Model

To rank the potential performance of a set of selected parallel architectures for an application or algorithm, we introduce a *Fitness* model that considers application and architecture parameters that are important for algorithm performance. In this model, the application is represented with the vector $\underline{APP}$. The vector $\underline{APP}$ has seven components:

$$\underline{APP} = \underline{i}SP + \underline{j}DP + \underline{k}DBT + \underline{l}HDT\_Ovfl + \underline{m}HDT\_Unfl + \underline{n}HDT\_UCntl + \sum_{n=1}^{N} \underline{o}_n OPT_n \qquad (7.1)$$

Here, *SP* is the total number of single-precision floating-point operations and *DP* is the total number of double-precision floating-point operations in the algorithm. *DBT* is the device byte transfer requirement for the algorithm, which represents the total bytes that must be transferred from device memory to the processing cores. For example, in the IBM CellBE architecture, it represents the amount of data transferred from the local stores to the processing cores of the SPU. On the other hand, in the GPGPU architecture,

DBT represents total data transferred from global memory to the individual scalar processor of the GPGPU device. *HDT_Ovfl* is the host-to-device data transfer requirement if the device memory is less than the total data required and *HDT_Unfl* is the host-to-device transfer if the device memory is sufficient to hold the total data required. For example, in the x86 architectures, if the maximum cache size is less than the total data required for computation, the total data will be *HDT_Ovfl*, otherwise, the total data will be *HDT_Unfl*. If the total data does not fit in the cache, then the data is transferred from the host to the device and after processing, the data is sent back to the host (*HDT_Ovfl*). In the PS3, the device memory (i.e., the local store) is very small (256 KB) compared to the cache size of x86. Thus instead of compulsory misses, there will always be capacity misses for any significant problem size. *HDT_UCntl* accounts for the host-to-device data transfer if the device memory is greater than total data and the programmer can manipulate the device memory. This kind of data transfer may happen with the GPGPU. Typically *HDT_UCntl* is the smallest among the three data transfers. *OPT* represents numerical values for the compute and data transfer optimization techniques that can be applied to the algorithm. For example, some algorithms can take advantage of software prefetching, the overlapping of data communication and computation. These optimization techniques can either reduce the amount of data communication or computation. Thus *OPT* represents either the reduced data to be transferred or reduced computation as a result of an applied optimization technique. Evaluating the numerical value of *OPT* requires in-depth analysis of the algorithm and the target accelerators.

Due to the flexibility of the representation, any accelerator can be represented by an accelerator vector, $\underline{ACC}$, which has a one-to-one component with the application vector, $\underline{APP}$. The accelerator vector can be written as:

$$\underline{ACC} = \underline{i}SSP + \underline{j}SDP + \underline{k}SDBT + \underline{l}SHDT - \sum_{n=1}^{N} \underline{o}_n SOPT_n \qquad (7.2)$$

Here, *SSP* represents the time in seconds, taken for an average single-precision floating-point operation in the accelerator. *SSP* can be calculated by taking the inverse of the theoretical single-precision flop/sec of the device. Similarly, *SDP*, for double-precision floating-point arithmetic, can be calculated by taking the inverse of the theoretical peak for double-precision floating-point operations. *SDBT* represents the time in seconds to transfer a single byte from the device memory to the processor. Typically, *SHDT* is the average byte transfer time for host-to-device and device-to-host data transfers. *SOPT* is the time in seconds for each floating-point operation or byte transfer depending on the optimization technique applied. This term is negatively signed since it saves time where the other terms increase the execution time.

In this model, the projected performance rank can be found by calculating the theoretical runtime of each algorithm on each platform. This runtime can be calculated by the scalar multiplication of an accelerator vector with an application vector. This scalar multiplication *ACC_APP*, as shown below in Equation 7.3 represents the projected performance and allows the user to rank the architecture and assess the match of an accelerator to application.

$$ACC\_APP = SP.SSP + DP.SDP + DBT.SDBT + (HDT\_Ovfl + HDT\_Unfl + HDT\_UCntl).SHDT - \sum_{n=1}^{N} OPT_n.SOPT_n \quad (7.3)$$

This *ACC_APP* represents the theoretical runtime if the algorithm is executed on the selected accelerator architecture. For an application/algorithm, the best match is the accelerator for which the value of *ACC_APP* is the lowest considering all possible accelerators as shown in Equation 7.4.

$$ACC\_APP_{best} = \min\left(\underline{APP}.\underline{ACC}_i\right) \quad (7.4)$$

The *Fitness* model makes some assumptions to provide a quick assessment of the match and therefore has some limitations. For example, it uses the peak value of hardware performance such as DRAM bandwidth and processor throughput. It also does not account for DRAM misses. These assumptions and limitations are detailed in Chapter 8.

7.3 Prediction Rank With the *Fitness* Model

Now we will use the *Fitness* model to predict the *Fitness* of architectures for accelerating all the case studies. First, we will discuss the performance prediction of matrix-matrix multiplication on the four architectures. Then, we project the performance of the four SNN models on the same four architectures.

7.3.1 Performance For Matrix-matrix Multiplication

In this section we will use the matrix multiplication algorithm case study to project the performance on accelerators using the *Fitness* performance model. Five representative square matrix sizes are chosen for the validation as shown in Table 7.1. Total number of flops, device-to-device transfer, and host-to-device transfer sizes for the five matrix-matrix multiplications are also given in the same table.

Table 7.1. Selected matrix sizes and corresponding characteristics

Matrix Size	Total flops	device-to-device transfer (byte)	host-to-device transfer (byte)
500×500	2.50E+08	1.00E+09	3.00E+06
1000×1000	2.00E+09	8.00E+09	1.20E+07
1500×1500	6.75E+09	2.70E+10	2.70E+07
2000×2000	1.60E+10	6.40E+10	4.80E+07
2500×2500	3.12E+10	1.25E+11	7.50E+07

To validate the *Fitness* model with the matrix-matrix multiplication, we utilize the four accelerators: Nvidia Fermi, Intel Xeon, AMD Opteron, and IBM PS3. The *Fitness* model requires the characteristics of accelerators, such as time required for the execution of floating point operations (SFLOP), device to device transfer (SDBT), and host to device transfer (SHDT), that are shown in Table A2a.

Table 7.2. Characteristics for the four accelerators

Accelerator	SFLOP (Sec/flop)	SDBT (Sec/Byte)	SHDT (Sec/Byte)
Nvidia Fermi	9.71E-13	6.47E-12	1.16E-10
Intel Xeon	6.71E-12	0	4.43487E-11
AMD Opteron	6.79E-12	0	4.63487E-11
PS3	6.51E-12	1.30208E-11	3.66662E-11

Table 7.1 and 7.2 provide the characteristics of applications and accelerators respectively that are used in the Fitness model to predict the performance rank of the architectures based on theoretical runtime. The theoretical runtime is shown in Table 7.3, and based on this table the predicted rank of the architecture is shown in Table 7.4. These results will be verified in Chapter 8.

Table 7.3. Predicted runtime (in second) of the accelerator based on Table 7.2

	500x500	1000x1000	1500x1500	2000x2000	2500x2500
Fermi	0.007059	0.055078	0.184317	0.435039	0.847505
Xeon	4.6E-02	3.68E-01	1.24E+00	2.95E+00	5.75E+00
Opteron	4.80E-02	3.84E-01	1.30E+00	3.07E+00	6.01E+00
PS3	1.48E-02	1.18E-01	3.96E-01	9.39E-01	1.83E+00

Table 7.4. *Fitness* Model Predicted rank of the accelerator based on Table 7.3

	500x500	1000x1000	1500x1500	2000x2000	2500x2500
Fermi	1	1	1	1	1
Xeon	3	3	3	3	3
Opteron	4	4	4	4	4
PS3	2	2	2	2	2

7.3.2 Performance For SNN

Now we will use the *Fitness* model to predict the *Fitness* of architectures for accelerating all four SNN models. To predict the performance, the theoretical values for floating-point performance and memory bandwidth are calculated from the architecture manuals. Table 7.3 in the previous subsection reports the theoretical performance values

for all four architectures as components of the *ACC* vector (all values are normalized to seconds).

Tables 7.5-7.7 provide the application vector components of the four SNN models. Here we use three problem sizes, the smallest network (9216 neurons), largest network (5.8 million neurons) and an interesting problem size (0.9 million neurons) to validate the *Fitness* performance model. The 0.9 million network size was selected because the four architectures rigorously studied behave uniquely for this network size. For example, the working dataset starts to exceed the cache size of the X86 for the Wilson and HH models. Table 4 contains the total single-precision floating-point (FP) operations for each SNN model, device byte transfer (DBT), and three kinds of host-to-device transfers (HDT) for 9216 neurons. For this problem size, the number of floating-point operations and amount of data transfer is much less than that for larger problems sizes.

After defining all of the values for the accelerator and application vector components, we can now predict the *Fitness* match with the scalar multiplication of the two vectors. After the scalar multiplication, we rank the matches between the accelerators and application as shown in Table 7.8. As shown in the table, the Xeon is predicted to perform the best for the Izhikevich and Wilson model whereas the PS3 is predicted to perform the worst. For the ML, and HH models the GPGPU is predicted to perform the best, whereas the PS3 is predicted to perform the worst.

Table 7.5. Application vector components of the four SNN models with a network size of 9216 neurons

Application	FP (Gflop)	DBT (MB)	HDT_Ovfl (MB)	HDT_Unfl (MB)	HDT_UCntl (MB)
Izhikevich	0.001	0.18	2.11	0.18	0.18
Wilson	0.010	0.40	11.21	0.39	0.40
Morris-Lecar	0.026	0.20	3.16	0.21	0.20
HH	0.911	3.42	144.25	0.39	3.42

Table 7.6. Application vector components of the four SNN models with a network size of 0.9 million neurons

Application	FP (Gflop)	DBT (MB)	HDT_Ovfl (MB)	HDT_Unfl (MB)	HDT_UCtl (MB)
Izhikevich	0.14	17.58	210.94	17.58	17.58
Wilson	0.99	39.55	1,121.48	560.74	39.55
Morris-Lecar	2.59	20.21	316.41	158.20	20.21
HH	91.10	341.89	14,424.61	7212.3	341.89

Table 7.7. Application vector components of the four SNN models with a network size of 5.8 million neurons

Application	FP (Gflop)	DBT (MB)	HDT_Ovfl (MB)	HDT_Unfl (MB)	HDT_UCntl (MB)
Izhikevich	0.001	0.18	2.11	0.18	0.18
Wilson	0.010	0.40	11.21	0.39	0.40
Morris-Lecar	0.026	0.20	3.16	0.21	0.20
HH	0.911	3.42	144.25	0.39	3.42

Table 7.8. Performance rank using the *Fitness* model for 9216 neurons

App_Acc match	Izhikevich	Wilson	ML	HH
GPGPU	3	3	1	1
Xeon	1	1	2	2
AMD	2	2	3	3
PS3	4	4	4	4

Table 7.9. Performance rank using the *Fitness* model for 0.9 million neurons

App_Acc match	Izhikevich	Wilson	ML	HH
GPGPU	3	1	1	1
Xeon	1	2	2	2
AMD	2	3	3	3
PS3	4	4	4	4

Table 7.10. Performance rank using the *Fitness* model for 5.8 million neurons

App_Acc match	Izhikevich	Wilson	ML	HH
GPGPU	3	1	1	1
Xeon	1	2	2	2
AMD	2	3	3	3
PS3	4	4	4	4

For the network of 0.9 million neurons, the total single-precision FP operations, device data transfer and the three kinds of host-to-device data transfers are reported in Table 7.6. After the multiplication, we rank the matches between the accelerators and application as shown in Table 7.9. As shown in the table, the Xeon is predicted to perform the best for the Izhikevich model whereas the PS3 is predicted to perform the worst. For the Wilson, ML, and HH models the GPGPU is predicted to perform the best.

Finally, for the largest network, 5.8 million neurons, the total single-precision FP operations, device data transfer and the three kinds of host-to-device data transfers are reported in Table 7.7. After the multiplication, we rank the matches between the

accelerators and application as shown in Table 7.10. In this case, the Xeon is predicted to perform the best for the Izhikevich model whereas the PS3 is predicted to perform the worst. For the other three models, the GPGPU is predicted to perform the best.

As seen from the rank tables (Tables 7.7-7.10), the ranks follow a trend. For the communication bound models (Izhikevich and Wilson) the X86 architecture is projected to be the top performer whereas for the compute intensive models (ML and HH), the GPGPU is projected to perform the best.

7.4 Extension of Fitness Model: Multiple Regression Model

Regression analysis is a statistical tool to model the relation between variables. When a range of data is available that represents the variation of a dependent variable with one or more independent variables, regression analysis can be utilized to determine if a relation exists between them. The regression analysis also determines the exact relation between the dependent and independent variables with a degree of confidence. Thus the process of finding a mathematical model (in the form of an equation) that best fits the data is a statistical technique known as regression analysis [118].

7.4.1 Background On Multiple Regression Model

Mathematically, regression analysis is concerned with relating a response y to a set of independent variables, x_1, x_2, x_3, ……. x_k. Here, the goal is to build a good mathematical model, i.e., a prediction equation. After the prediction equation is built, the response y can be predicted for a particular set of values of x_1, x_2, x_3, ……. x_k with a degree of

confidence. For example, if a response dependent variable, y only depends on two independent variables, x_1 and x_2, the predicted equation can take the following form:

$$y = \alpha_1 + \alpha_2 x_1 + \alpha_3 x_2 + \alpha_4 x_1^2 + \alpha_5 x_2^2 + \alpha_6 x_1 x_2 + \varepsilon \qquad (7.5)$$

Here, α_1, α_2, α_3, α_4, α_5, and α_6 are the estimates of the model parameters. Meaning of different terms will be described later in this section. These parameters can be evaluated by using several statistical methods. In our study we will use least square method for the regression analysis.

At this stage, any difference between the actual response and the modeled response for a particular set of independent variables produces errors. The least square method must satisfy two important properties regarding the errors. First, the sum of error (SE) should be equal to zero. Second, the sum of squares of the errors (SSE) is the minimum. Mathematically,

$$SE = \sum (y_i - \hat{y}_i) = 0 \qquad (7.6)$$

$$SSE = \min \left\{ \sum (y_i - \hat{y}_i)^2 \right\} \qquad (7.7)$$

Where, y_i is the actual response and $\hat{y}_i$ is the estimated response with the regression model. Thus, to establish a mathematical model using regression analysis on a set of data, all "α"s of equation (7.5) are to be evaluated satisfying equations (7.6) and (7.7).

In addition to satisfying the properties (7.6) and (7.7) with the regression model, we will make four basic assumptions about the probability distribution of ε. In brief the assumptions are as follows [118]:

1) The mean of the probability distribution of ε is zero
2) The variance of the probability distribution of ε is constant for all settings of the independent variables *x*'s
3) The probability distribution of ε is normal
4) The error associated with any two different observations are independent

These assumptions help to develop a measure of reliability for the least squares estimators and a hypothesis tests for the model.

There are two popular types of regression model that are used in many practical purposes: 1) First-Order Model and 2) Multiple Regression model. The Interaction Model and Second Order Model are also derivatives of the Multiple Regression model. In our research, we use both the Interaction model and Second Order model to model the performance of architectures and programming models.

7.4.1.1 First Order Model

The First Order model is the simplest model and is also called the Straight Line model since the relationship between the response and dependent variable can be shown as a straight line. In this model, the response *y* depends on only one independent variable, *x*. Thus the first-order model is:

$$y = \alpha_1 + \alpha_2 x_1 + \varepsilon \qquad (7.8)$$

Here, α_1 is the *y*-intercept of the line. Whether this α_1 has a physical meaning or not, depends on the range of data used in the model. Only when $x = 0$ is within the range of the x values in the sample, will α_1 have a meaningful interpretation. α_2 is the slope of the line, i.e., amount of increase or decrease in the mean of *y* for every 1-unit increase of *x*.

7.4.1.2 Multiple Regression Model

Regression models that include more than one independent variable are called multiple regression models. The general form of the multiple regression models is

$$y = \alpha_1 + \alpha_2 x_1 + \alpha_3 x_2 + \alpha_4 x_3 + ... + \alpha_k x_k + \varepsilon \qquad (7.9)$$

Where, *y* is the dependent variable and all *x*'s are the independent variables. Here any dependent variable can be higher ordered. For example x_2 may represent x_1^2.

If the model includes only terms for quantitative independent variables, it is called a multiple first-order regression model. This is the most basic multiple regression model. If the model includes quadratic terms, x^2, it will enable us to hypothesize the curvature in the graph.

Moreover, if the effect of change of an independent variable (x_1) on the value of dependent variable (y) is also depends on another variable (x_2), then it can be said that x_1 and x_2 interact. Thus, to model this situation, an interaction term (x_1x_2) is included in the model. For example if there are only two independent variables, x_1 and x_2, and the two variables interact, then the regression model will be:

$$y = \alpha_1 + \alpha_2 x_1 + \alpha_3 x_2 + \alpha_4 x_1 x_3 + \varepsilon \qquad (7.10)$$

Extreme care must be taken when dealing with interaction models. First, as the independent variables interact, the coefficient of the independent variables may appear to be negative; though that may not mean that the response decreases with an increase of that independent variable as the same variable is also present in the interaction term. Second, the most important parameter in this model is the interaction term. Thus when we want to perform a hypothesis test (such as H_0: $\alpha = 0$) to evaluate the utility of the model, we will first evaluate the interaction term (such as H_0: $\alpha_4 = 0$). Once the interaction term is detected to be valid by the hypothesis test, the hypothesis test of other terms (α_1 ,α_2 ,α_3) are not needed. The presence of an interaction term implies that both x's are important. The application of the regression model with the performance model will use both the models shown in equation (7.9) and (7.10).

7.4.2 Regression Model For Architectures

As we mentioned in the preceding discussion, we have gathered vast amount of data on the performance of four baseline accelerators (Intel Xeon, AMD Opteron, PS3, and

Nvidia GPGPU) and four advanced architectures (Intel 32 core, AMD 32 core, Sun 32 core, IBM 16 core). In this section we will apply a regression model using the JMP software by SAS [119] to the performance results of the architectures. We will first develop a regression model for the four baseline architectures and then, we will develop a regression model for the four advanced architectures.

7.4.2.1 Intel Xeon

The runtime results of the Intel Xeon architecture were collected for the four SNN models. Runtimes for various problem sizes of the neural networks, and also for various accelerator configurations were collected. A regression model was developed based on this dataset; in this model, runtime is the dependent variable and there are three independent variables: total number flops, total number of memory access required, and the number of cores. To evaluate the reliability of the model we rely on the R-squared value of the overall model and the p-values of the individual coefficient. The model is considered reliable if the *R-squared* value is greater than 0.95 and the individual p-value is smaller than 0.05. The general model for the Intel Xeon is:

$$R\_time = \alpha_1 + \alpha_2\, flops + \alpha_3 bytes + \alpha_4 cores + \alpha_5\, flops \times cores + \varepsilon \qquad (7.11)$$

Where, *R_time* is the runtime variable (the dependent variable), *flops* is the total number of flops, *bytes* is the total number bytes access in the applications, *cores* is the number of cores in the processor, and ε is the estimation error. After running the JMP statistical software using equation (7.11) on the dataset for the Intel Xeon, we obtain the following equation:

$$R\_time = 3033 + 2.6\times10^{-8}\ flops + 1.23\times10^{-7}\ bytes - 479.06 cores - 1.41\times10^{-8}\times(cores - 6.37)(\ flops - 32\times10^{10}) + \varepsilon \quad (7.12)$$

In equation (7.12), in addition to the three independent variables, *flops*, *bytes*, and *cores*, one additional term, *cores*bytes* were included. While experimenting with the model development, it was found that, without the *cores*×*flops* term, the *R-squared* term is 0.51 which is far below the desired value of 0.95 and the individual p-values are also far below the desired value of 0.05, indicating the model is not considered reliable. After the inclusion of the term, *cores*×*flops*, the *R-squared* becomes 0.915 (slightly smaller than 0.95), and the individual p-value of the coefficient is less than 0.05. This interaction term also has a physical meaning: with the addition of processing cores, flops are divided into smaller parts among the processing cores.

7.4.2.2 AMD Opteron

Four SNN model runtimes on the AMD Opteron architecture for various problem sizes and with the variation of processing cores are used to build a regression model. As before runtime is the dependent variable and the total number of flops, total number of memory access required, and the number of cores are the independent variables. The regression model for the AMD Opteron is derived similar to that of equation (7.11).

After running the JMP software on the Opteron runtime dataset, the following regression model is developed:

$$R\_time = 3181.24 + 4.06\times10^{-8}\ flops + 1.30\times10^{-7} bytes - 495.53 cores \\ -1.96\times(cores - 6.37)(\ flops - 2.6\times10^{10}) + \varepsilon \quad (7.13)$$

As for the Intel Xeon architecture, the regression model includes the interaction *cores*flop* term in addition to the three independent variables. The *R-squared* value for the model is 0.99, and the *p*-values for each of the coefficients are smaller than 0.05, thus, satisfying the utility criteria of regression model.

7.4.2.3 IBM PS3

Using similar independent and dependent variables, the regression model for IBM PS3 can be written as shown in equation (7.13). After running the JMP statistical software using equation (7.13), the following equation is developed:

$$R\_time = 45.71 + 2.3041\times10^{-8}\ flops + 5.1863\times10^{-8} bytes + \varepsilon \quad (7.14)$$

For this model, the *R-squared* value is 0.88 which is slightly smaller than 0.95. The model can tolerate such value of *R-squared* if all the *p*-values satisfy the utility criteria (i.e., lower than 0.05). Observing the p-values, it is found that other than the p-value for the coefficient of *cores* (0.15), all of the p-values of the other coefficients are smaller than 0.05. Since the p-value for the coefficient of the *cores*flop* term is smaller than 0.05, the p-value for the coefficient of the *cores* term does not have to satisfy the utility criteria [118]. Thus this model is a valid model with sufficient support from both the *R-squared* and *p-values* satisfying the utility criteria.

7.4.2.4 Nvidia GPGPU (Fermi)

The regression model for the Nvidia Fermi GPGPU is different from the previous three models. In this regression model, *runtime* is the dependent variable and *total number flops* and *total number of memory access required* are the two independent variables. Since the number of cores is not varied for the GPGPU, the *cores* term is not included in the model. Thus, with two independent variables, the simple regression model for GPGPU can be presented as:

$$R\_time = \alpha_1 + \alpha_2\, flops + \alpha_3 bytes + \varepsilon \qquad (7.15)$$

After running JMP software using equation (7.15), it is found that this simple model does not satisfy the utility criteria. Thus we have added the *flop*byte* term to the equation (7.15):

$$R\_time = \alpha_1 + \alpha_2\, flops + \alpha_3 bytes + \alpha_4\, flop \times bytes + \varepsilon \qquad (7.16)$$

After including this interaction term, the *R-squared* and the *p-value* of each of the coefficients satisfy the utility criteria. The physical meaning of the interaction term (*flop*byte*) for the GPGPU model is that the number of flops and number of bytes required interact, i.e., the time required for GPGU multiprocessor to execute each floating point operation partially depends on the number of bytes it requires as operands. After running the JMP software, based on equation (7.16), the following equation, which satisfies the utility criteria, is developed:

$$\begin{aligned} R\_time = {} & 36.22 + 1.21\times10^{-8}\, flops + 8.77\times10^{-8} bytes \\ & - 3.41\times10^{-21} \times (flop - 4.3\times10^{10})(bytes - 7.5\times10^{9}) + \varepsilon \end{aligned} \qquad (7.17)$$

7.4.2.5 Advanced Multicores

In this section, we will develop regression models for the Intel 32 core, AMD 32 core, SUN 32 core, and IBM 16 core systems. The runtime experimental runtime data for these advanced architectures was used to develop a regression model for the architectures; the general equation for these architectures is shown in equation (7.18). The independent variables in this equation are same as the regression model for the Intel Xeon discussed in Section 7.4.2.1.

$$R\_time = \alpha_1 + \alpha_2\, flops + \alpha_3 bytes + \alpha_4 cores + \alpha_5\, flop \times cores + \varepsilon \qquad (7.18)$$

After running the JMP software on the relevant datasets, the complete models are given in equations (7.19) - (7.22)

a) Intel 32 core:

$$\begin{aligned} R\_time = {} & 31654.39 + 3.69\times10^{-8}\, flops + 5.06\times10^{-8} bytes \\ & - 33681.7 cores - 3.88\times10^{-7} \times (flop - 8.5\times10^{10})(cores - .9446) + \varepsilon \end{aligned} \qquad (7.19)$$

b) AMD 32 core:

$$\begin{aligned} R\_time = {} & 43595.43 + 4.55\times10^{-8}\, flops + 1.10\times10^{-7} bytes \\ & - 46664 cores - 5.4\times10^{-7} \times (flop - 8.5\times10^{10})(cores - .93446) + \varepsilon \end{aligned} \qquad (7.20)$$

c) SUN 32 core

$$\begin{aligned} R\_time = {} & 786328 + 1.64\times10^{-6}\, flops + 9.43\times10^{-7} bytes - 841157 cores \\ & - 1.73\times10^{-5} \times (flop - 4.8\times10^{10})(cores - .9345) + \varepsilon \end{aligned} \qquad (7.21)$$

d) IBM 16 core

$$R\_time = 32394 + 3.67\times10^{-8}\ flops + 1.11\times10^{-7} bytes - 34784 cores - 4.04\times10^{-7}\times(flop - 8.5\times10^{10})(cores - .9283) + \varepsilon \quad (7.22)$$

All of these models satisfy the utility criteria for the *R-squared* value and *p-values*.

7.4.3 Regression Model for the Programming Model

The parallel programming models each have their own effect on the performance capabilities of accelerators as discussed in the previous chapters. In this section we will develop a regression model for each of programming models. First, we will develop models for the GPGPU-based programming models: CUDA and OpenCL on the Nvidia Fermi architectures. Then we will present regression models for the X86-based programming models: POSIX-threading, OpenMP, Concurrency Runtime, and OpenCL on Intel i7 system.

7.4.3.1 GPU-based Software Model

Since the Nvidia Fermi GPGPU supports both the CUDA and OpenCL programming models, it is used as the GPGPU platform for studying and developing while developing the regression models.

The runtime results with CUDA on the Fermi architecture are used to develop the regression model shown below, which is similar to equation (7.17). Thus same explanation for equation (7.17) can be applied here.

$$R\_time = 36.22 + 1.21\times10^{-8}\ flops + 8.77\times10^{-8} bytes - 3.41\times10^{-21}\times(flop - 4.3\times10^{10})(bytes - 7.5\times10^{9}) + \varepsilon \quad (7.23)$$

Similarly, for OpenCL on the Fermi architecture, the regression model can be represented by equation (7.24):

$$R\_time = 28.802 - 1.17\times10^{-8}\ flops + 8.49\times10^{-8} bytes - 2.17\times10^{-21}\times(flop - 4.3\times10^{10})(bytes - 7.5\times10^{9}) + \varepsilon \quad (7.24)$$

Both of the regression models shown in equations (7.23) and (7.24), satisfy the utility criteria for the *R-squared* and *p-values* of the coefficients.

7.4.4 X86-based Programming Models

In this section, the performance impact for the four programming models on Intel i7 system is modeled with statistical regression models.

7.4.4.1 POSIX-threading

The runtime dataset for p-threading on the Intel i7 architecture is used to develop a regression model as shown in equation (7.25):

$$R\_time = \alpha_1 + \alpha_2\, flops + \alpha_3 bytes + \alpha_4 (bytes)^2 + \alpha_5\, flops \times bytes + \varepsilon \quad (7.25)$$

In equation (7.25), *flops* and *bytes* are used as independent variables; however, a quadratic term $(bytes)^2$ and an interaction term, $flops \times bytes$, are present in this model. Without these terms, the *R-squared* value and *p-values* of each coefficient would not pass the utility criteria. The reason for the $(bytes)^2$ term is that the relation between

runtime and *bytes* has a quadrature component because the p-threading technique creates explicit threads for each of the cores. Thus each of the threads makes use of the Level-1 cache for each of the cores. This explicit use of the Level-1 cache requires the $(bytes)^2$ term. The interaction term ($flops \times bytes$) is also included due to the explicit thread creation by the p-threading technique. For executing each floating-point operation, p-threading makes use of the cache structure of each of the cores to make the *bytes* available. Thus the runtime for executing each flop also depends on the number of *bytes* it requires to execute. After running the statistical software, JMP, based on equation (7.25), the *R-squared* value of the model is 0.99 and all the p-values are lesser than 0.05. The resulting model is shown below:

$$\begin{aligned} R\_time = {} & 111.01 + 3.13\times10^{-7}\ flops - 3.27\times10^{-7} bytes \\ & - 3.27\times10^{-17}\times(bytes - 7.5\times10^{9})\times(bytes - 7.5\times10^{9}) \\ & + 5.43\times10^{-18}(flops - 4.3\times10^{10})\times(bytes - 7.5\times10^{9}) + \varepsilon \end{aligned} \qquad (7.26)$$

7.4.4.2 OpenMP, Concurrency Runtime, And OpenCL

The general regression model for the remaining three x86 programming models, OpenMP, Concurrency Runtime, and OpenCL is:

$$R\_time = \alpha_1 + \alpha_2\, flops + \alpha_3 bytes + \varepsilon \qquad (7.27)$$

Where, *R_time* is the runtime variable which is the dependent variable, *flops* is the total number of flops and *bytes* is the total number bytes access in the applications, and ε is the estimation error. This regression equation does not have the quadratic term or interaction term that was found in p-threading model because these three models do not create threads explicitly for each of the processing cores as explained in Chapter 6.

After running the JMP software based on equation (7.27), OpenMP, Concurrency Runtime, and OpenCL, the respective regression models are given in (7.28), (7.29), and (7.30):

$$R\_time = 70.34 + 1.39 \times 10^{-7}\ flops + 1.68 \times 10^{-7} bytes + \varepsilon \qquad (7.28)$$

$$R\_time = 214 + 2.12 \times 10^{-7}\ flops + 1.37 \times 10^{-7} bytes + \varepsilon \qquad (7.29)$$

$$R\_time = 70.34 + 1.39 \times 10^{-7}\ flops + 1.68 \times 10^{-7} bytes + \varepsilon \qquad (7.30)$$

7.5 Summary

In this chapter we have introduced and described the Roofline model, *Fitness* model, and regression models for architectures and performance impact of programming models. The *Fitness* and regression models evolved through the lessons learned from the performance analysis of various multicore and GPGPU architectures and performance impact of programming modes discussed in Chapter 5 and 6. The *Fitness* model ranks the accelerators for an application with a specified problem size. We have ranked four accelerators for various problem sizes of matrix multiplication and three different problem sizes for the four SNN models. In this version of *Fitness* model, we have not

included the performance impact of programming models and threading effects. An extended version of the *Fitness* model using multiple regression techniques has been developed. The multiple regression models include the effect of threading (i.e., number of processing cores) and programming models on accelerator performance. We have developed eight regression models for eight accelerators and six regression models for six programming models. In addition to the *Fitness* and regression models, we have utilized the published Roofline model to predict the performance of architectures qualitatively. The Roofline also provides insights regarding the potential bottlenecks of application performance on architectures. As seen from the discussion of the Roofline model, it also does not include the effect of memory hierarchy, problem sizes, software effect, and the threading effect. The validation of these models using the datasets in Chapters 5 and 6 are presented in the next chapter.

Chapter 8

Performance Model Validation

There are three accelerator performance models considered in this research. First, is the published Roofline model [22]; second, our proposed *Fitness* model; and, the third is the extended Fitness model, i.e., multiple regression model. We validate the *Fitness* model using our performance analysis of the multicore and GPGPU architectures using the matrix-matrix multiplication and SNN case studies. Next, the multiple regression models are also verified using the SNN models case study. Finally, we use the Roofline model to further verify the *Fitness* model findings by reporting the throughput performance of the four accelerators implementing the SNN case studies.

8.1 *Fitness* Model Validation

In this section we will discuss the validation of the *Fitness model* using the matrix-matrix multiplication and SNN case studies.

8.1.1 Matrix-matrix Multiplications

In this section we will use the matrix multiplication algorithm case study to validate the *Fitness* performance model. Five representative square matrix sizes are chosen for the validation as shown in Table 8.1. Total number of flops, device-to-device transfer, and host-to-device transfer sizes for the five matrix-matrix multiplications are also given in the same table.

Table 8.1. Selected matrix sizes and corresponding characteristics

Matrix Size	Total flops	device-to-device transfer (byte)	host-to-device transfer (byte)
500×500	2.50E+08	1.00E+09	3.00E+06
1000×1000	2.00E+09	8.00E+09	1.20E+07
1500×1500	6.75E+09	2.70E+10	2.70E+07
2000×2000	1.60E+10	6.40E+10	4.80E+07
2500×2500	3.12E+10	1.25E+11	7.50E+07

Table 8.1 provides the characteristics of applications are used in the Fitness model to predict the performance rank of the architectures based on the theoretical runtime. The predicted rank of the architectures is shown in Table 8.2.

Table 8.2. Fitness Model predicted performance rank of the accelerator

	500x500	1000x1000	1500x1500	2000x2000	2500x2500
Fermi	1	1	1	1	1
Xeon	3	3	3	3	3
Opteron	4	4	4	4	4
PS3	2	2	2	2	2

The Tables 8.1 and 8.2 were originally developed in Section 7.3 and are repeated here for convenience.

The actual runtime of the matrix-matrix multiplication for various problem sizes is reported in Table 8.3 which is extracted from Figure 5.21 of Chapter 5. Based on this runtime table, the actual performance rank is reported in Table 8.4. Comparing Tables 8.2 and 8.4, we see that all of the predictions match with the actual rank except two cases: the rank of PS3 for the 500x500 and 1000x1000 matrix sizes. For these two

matrix sizes, the rank of PS3 is predicted to be 2, but, the actual rank is 4. The actual performance of the PS3 architecture is less than the predicted values for the larger problem sizes due to the congestion in the EIB of the PS3. The EIB has only four lanes to transfer data between the SPU and DRAM. So when the problem size is very large, 6 SPUs issue multiple DMA request simultaneously for data from the DRAM, the request processing time is increased due to the number and size of requests. The DMA requests that take longer than usual fail because of the time out. Thus, repeated DMA requests will be issued, further increasing the communication time and eventually negatively impacting the performance. As this issue is not considered in the *Fitness* model, the PS3 performance is predicted higher than the actual performance.

Table 8.3. Actual runtime (in second) of the accelerators

	500x500	1000x1000	1500x1500	2000x2000	2500x2500
Fermi	0.003435	0.010979	0.0222	0.03884	0.05881
Xeon	0.015864	0.148708	0.410725	0.9353	1.869817
Opteron	0.021199	0.185512	0.544591	1.23107	2.460275
PS3	0.03	0.22	0.33	.6601	1.13

Table 8.4. Actual performance rank of the accelerators

	500x500	1000x1000	1500x1500	2000x2000	2500x2500
Fermi	1	1	1	1	1
Xeon	2	2	3	3	3
Opteron	3	3	4	4	4
PS3	4	4	2	2	2

8.1.2 SNN Models

In Chapter 6, we ranked the architectures for the SNN model case studies for three different network sizes. Now we will use the performance data collected in Chapter 5 to determine the actual rank and validate the *Fitness* performance model with this case study. Starting with the smallest network size, 9216 neurons, the performance ranks reported in Table 8.5 are based on the measured performance shown in Figures 5.1 - 5.4 of Chapter 5. The predicted ranking shown earlier in Table 7.8 of Chapter 7 is approximately same as the experimental results shown in Table 8.5, the only difference is that the rank of the GPGPU and PS3 for the Izhikevich model is interchanged.

Table 8.5. Actual performance rank for 9216 neurons

	Izhikevich	Wilson	ML	HH
GPGPU	4	3	1	1
Xeon	1	1	2	2
AMD	2	2	3	3
PS3	3	4	4	4

In the performance prediction, the effect of optimization techniques was not considered. For the PS3, the double buffering optimization, which is used to overlap the computation with communication, improves the performance significantly. The effect of double buffering on the PS3 can also be seen in Figure 5.1 of Chapter 5. These optimizations are not available for the GPGPU implementation. Thus we believe, this is one of the main reasons for the error in the prediction from the actual implementation for the Izhikevich model.

Table 8.6. Actual performance rank for 0.9 million neurons

	Izhikevich	Wilson	ML	HH
GPGPU	4	1	1	1
AMD	1	2	2	2
Xeon	2	4	3	4
PS3	3	3	4	3

The *Fitness* rank based on the actual implementation for 0.9 million neurons is reported in Table 8.6. Comparing this table with the corresponding *Fitness* projection table, Table 7.9 of Chapter 7, a couple of differences are noted. For the Izhikevich, Wilson, and HH models, ranks 3 and 4 are interchanged. For all of the cases, the PS3 performance was predicted to be the worst whereas, in the actual implementation the rank was slightly better, rank 3. As we discussed for the smallest network, the double buffering technique was not considered in the *Fitness* model causing the rank for the PS3 to be predicted lower than its actual implementation. Another reason is that the AMD has an architectural limitation (as discussed in subsection 5.2) that was not captured in the *Fitness* model.

Table 8.7. Actual performance rank for 5.8 million neurons

	Izhikevich	Wilson	ML	HH
GPGPU	4	1	1	1
Xeon	1	2	2	2
AMD	2	3	3	3
PS3	3	4	4	4

The actual performance rank of the four architectures for the largest network, 5.8 million neurons, is given in Table 8.7. Comparing these with the projected ranks in Table 7.10, we again find one difference. For the Izhikevich model, the predicted PS3 rank is 4, whereas the actual performance rank for PS3 is 3. The same explanation, given in the previous two cases can be applied.

8.1.3 Fitness Model Validation Summary

From the above discussion, we note that except for some of the cases with the PS3 architecture, the *Fitness* model projections match the actual implementation. The current *Fitness* model only ranks the relative *Fitness* of an architecture; it does not at this time predict the actual runtime performance of an architecture. One limitation is that the model does not account for the cost involved in DRAM misses. Thus, the *Fitness* model does not account for the performance variation on the PS3 while implementing the largest network of the Wilson and HH models. In these cases, the DRAM misses cause a sharp decline in performance. The Fitness model also does not account for complicated bus architectures, such as EIB of PS3. For larger problem sizes, the EIB becomes congested negatively impacting the performance as seen with the matrix-matrix multiplication case study (Section 8.1.1).

The *Fitness* model allows for the use of the numeric performance values for the optimization techniques applied. We have not used these values in this research, since it requires rigorous study to estimate them, which defeats the purpose of the *Fitness* model: to provide a quick estimation of the potential top performing architectures for a given algorithm. As will be discussed in future work, we plan to reduce these limitations and apply the *Fitness* model for other applications.

8.2 Regression Model Validation

In this section we will verify the statistical regression models for accelerators and programming models developed in Section 5.2.

8.2.1 Regression Model For Accelerators

To verify the regression models for accelerators, we have chosen two representative accelerators: one x86 processor, Intel Xeon and one GPGPU, Nvidia Fermi. We have chosen one data point, 1680x1680 for the four SNN models. The total number of flops and required bytes are shown in Table 8.8. Utilizing these values and the regression models developed in section 7.4, the predicted runtime for the Intel Xeon and Nvidia Fermi is calculated and reported in Tables 8.9 and 8.10 respectively.

Table 8.8. SNN model characteristics for verification of Accelerator Regression models

Observation	SNN Model	Network Size	Total Flops	Total bytes
1	Izhikevich	1680x1680	4.4E+08	8.13E+08
2	Wilson	1680x1680	3.11E+09	3.60E+09
3	Morris-Lecar	1680x1680	5.59E+09	1.19E+09
4	HH	1680x1680	2.79E+11	4.63E+10

Table 8.9. Predicted runtime from the Intel Xeon regression model

Observation	Intercept Term	flop term	byte term	core term	flop*core term	Predicted runtime (ms)
1	3033.8719	1.15E+01	1.00E+02	-3832.5272	727.3944066	40.50462
2	3033.8719	8.14E+01	4.44E+02	-3832.5272	665.8559339	392.7348
3	3033.8719	1.46E+02	1.46E+02	-3832.5272	605.6113447	99.41567
4	3033.8719	7.30E+03	5.71E+03	-3832.5272	-5663.18192	6552.802

Table 8.10. Predicted runtime from the Nvidia Fermi regression model

Observation	Intercept Term	flop term	byte term	flop*byte term	Predicted runtime (ms)
1	36.223629	-5.33E+00	7.13E+01	-0.970496517	101.1952119
2	36.223629	-3.77E+01	3.16E+02	-0.530305283	313.8113892
3	36.223629	-6.77E+01	1.04E+02	-0.80557599	71.68442536
4	36.223629	-3.38E+03	4.06E+03	-31.23914143	688.1657636

The predicted runtime from the regression model is compared to the measured results in Table 8.11. The error rate is reported in the same table. From the table we see that the error is lower for longer runtimes. The highest error rate is 16.7% for the shortest runtime (132.21 ms) and the lowest error is 0.27 % for the longest runtime (8517 ms). This error variation results because a small deviation from the model for shorter runtimes creates a greater percentage of error, whereas the same deviation for longer runtimes creates a smaller percentage of error.

Table 8.11. Verification of results for the regression model of two accelerators

Observation	Actual runtime for Intel Xeon (ms)	% of Error for Intel Xeon	Actual runtime for Nvidia Fermi (ms)	% of Error for Nvidia Fermi
1	45.19	10.37	105.5568	4.310121
2	406.16	3.30	300.1	-4.36931
3	113.31	12.26	78.6187	9.673279
4	6632.00	1.19	623.98	-9.32703

8.2.2 Regression Model for Programming Models

In this section, the regression model for the programming models will be verified against the actual runtime results. A data point 1200x1200 for the four SNN models is used in the verification process. We have chosen two representative programming models for verification: Concurrency Runtime and CUDA. Table 8.12 provides the total flops and total bytes required for the four SNN models. Plugging the values from Table 8.12 into the regression model equations developed in Section 7.4 (equation (7.23) and (7.29)), the individual terms and the calculated predicted runtime is reported in Table 8.13 and 8.14 for CUDA and Concurrency Runtime respectively.

Table 8.12. SNN model characteristics for verification of accelerators regression model

Observation	SNN Model	Network Size	Total Flops	Total bytes
1	Izhikevich	1200x1200	2.25E+08	4.15E+08
2	Wilson	1200x1200	1.59E+09	1.17E+09
3	Morris-Lecar	1200x1200	2.85E+09	6.05E+08
4	HH	1200x1200	1.42E+11	2.36E+10

Table 8.13. Predicted runtime from the CUDA regression model

Observation	Intercept	flop term	byte term	flop*byte term	Predicted runtime
1	36.22	-2.72	36.36	-1.03	68.83
2	36.22	-19.22	102.53	-0.89	118.64
3	36.22	-34.53	53.03	-0.94	53.78
4	36.22	-1723.70	2072.26	-5.46	379.32

Table 8.14. Predicted runtime from the Concurrency Runtime regression model

Observation	Intercept	flop term	byte term	Predicted runtime
1	214.69	30.78	88.32	333.79
2	214.69	217.40	249.02	681.11
3	214.69	390.61	128.80	734.11
4	214.69	19500.14	5033.18	24748.01

These predictions, are compared with the actual runtime to calculate error and are reported in Table 8.15. A similar error pattern occurs, high error rate for short runtimes and lower error rate for the longer runtimes, as was found in the case of accelerator regression model.

Table 8.15. Verification of results for two programming models

Observation	Actual time for CUDA	% of Error for CUDA	Actual time for Concurrency Runtime	% of Error for Concurrency Runtime
1	62.24	-10.59	296.1	12.76497
2	99.86	-18.80	610.43	11.67447
3	60.68	11.36	792.82	-7.4208
4	365.19	-3.86	24724	-0.11818

8.2.3 Verification With Roofline Model

We have evaluated the throughput of the four base architectures for the SNN model. The throughput and the percentage of peak throughput for the four architectures with the four SNN models is reported in Tables 8.16 and 8.17. It is clear from Table 8.16 that the throughput for the Izhikevich and Wilson models with the four architectures is about 6%. It can be partially explained by examining the Roofline model. These two SNN models are communication bound and thus can achieve less throughput than a kernel that is computation bound. For the ML and HH models shown in Table 8.16, the throughput of the four architectures is higher as shown in the Roofline graphs, Figures 7.1 - 7.4 in Chapter 7. The maximum throughput, 267 Gflops (77% of the peak Gflop) was achieved by the GPGPU for the HH model, which was discussed in the architecture performance sub-section.

Table 8.16. Achieved and % of Peak Gflop/s performance for the Izhikevich and Wilson models

	Izhikevich		Wilson	
	2.8 million	5.8 million	2.8 million	5.8 million
	Gflop/s (% of peak)	Gflop/s (% of peak)	Gflop/s (% of peak)	Gflop/s (% of peak)
GPGPU	3 (1%)	3 (1%)	16 (5%)	17 (5%)
Xeon	9 (6%)	4 (3%)	7 (5%)	6 (4%)
AMD	4 (3%)	4 (2%)	3 (2%)	5 (3%)
PS3	4 (3%)	4 (3%)	8 (5%)	0.14 (0.1%)

Table 8.17. Achieved and % of Peak Gflop/s performance for the ML and HH models

	Morris-Lecar		Hodgkin-Huxley	
	2.8 million	5.8 million	2.8 million	5.8 million
	Gflop/s (% of peak)	Gflop/s (% of peak)	Gflop/s (% of peak)	Gflop/s (% of peak)
GPGPU	43(12%)	45 (13%)	244 (70%)	267 (77%)
Xeon	33 (21%)	32 (21%)	39 (25%)	39 (25%)
AMD	26 (17%)	28 (18%)	30 (20%)	30 (20%)
PS3	25 (17%)	23 (16%)	33 (23%)	8 (5%)

8.3 Performance Explanation With Fitness And Roofline Model

In this section, we will use the *Fitness* and Roofline models to analyze and explain the performance behavior of the four accelerators. In Section 5.2, the performance of the accelerators was studied by varying the problem sizes for the SNN case studies. In that section, the performance behavior of the accelerators was explained in terms of the characteristics of accelerators and algorithms. In this section we will revisit the same performance results and explain them in terms of the *Fitness* and Roofline models. The successful explanations of the performance behavior of accelerators using *Fitness* and Roofline models further validate the models.

8.3.1 Izhikevich And Wilson Model

The performance behavior for the Izhikevich and Wilson models on the four architectures is shown in Figures 8.1 and 8.2 respectively. For the GPGPU implementation of the Izhikevich model, all of the working datasets fit in global memory and no maxima is seen in the speedup graph. However, the amount of computation in the Izhikevich model is very low compared to the required data transfer and the GPGPU cannot take advantage of its high computation throughput as the problem size increases.

The GPGPU host-to-device bandwidth (*1/SHDT*=2 GB/s) is the lowest of all the architectures studied and there is another data transfer required inside the GPGPU (*1/SHDT*=1500 GB/s), which together result in the GPGPU having the lowest rank performance among the architectures. For the Xeon, the working dataset does not fit in the cache beyond 1 million neurons, therefore in the *ACC_APP* equation, the required data transfer value is shifted from *HDT2* to *HDT1*. Since *HDT1* is greater than *HDT2*, it requires additional time to transfer data resulting in degraded performance as the problem size increases. For the AMD Opteron, a similar explanation can be applied. For the PS3, the working dataset is larger than the device memory size (SPU local store size) for even the smallest problem size. Thus, the equation *ACC_APP* uses *HDT1* for all of the network sizes and there is no maxima seen in that graph.

A similar explanation applies for the Wilson neuron model on the four architectures using the *Fitness*.

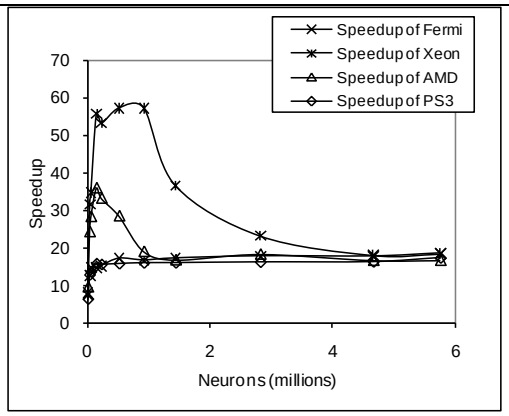

Figure 8.1. Izhikevich model: Speedup performance of the four architectures over Intel Core 2 Quad

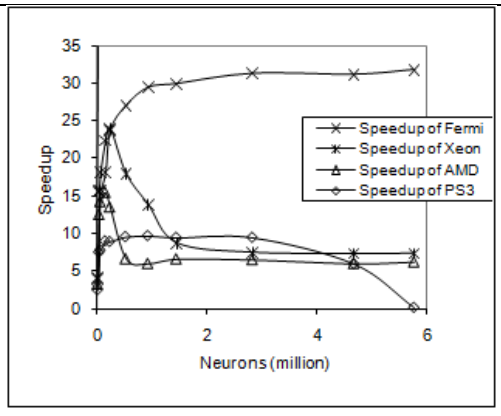

Figure 8.2. Wilson model: Speedup performance of the four architectures over Intel Core 2 Quad

With the help of the Roofline model, we see that the Izhikevich neuron model is communication bound for the x86 architectures. Thus the performance of the Xeon and Opteron architectures for the Izhikevich neuron model do not grow beyond a peak value. However, the maxima seen in the x86 performance curve for the Izhikevich model cannot be completely explained by the Roofline model alone, as the Roofline model does not consider the variation of performance with the problem size. For the same reason, the Roofline model does not explain the maxima seen for the Wilson model.

8.3.2 Morris-Lecar And HH Model

The *Fitness* and Roofline performance models provide insights into the performance of the Morris-Lecar and HH neuron models on the four architectures as shown in Figures 8.3 and 8.4 respectively. As mentioned, the Morris-Lecar model has a higher flop requirement and approximately same amount of data transfer compared to the Izhikevich model as seen in Table 5.1 in Chapter 5. For the GPGPU, the computation term (FP.SFP) of the ACC_APP equation is smaller, compared to the other architectures, because the SFP of the GPGPU is much smaller than that of the other architectures. Though, the data transfer rate for the GPGPU is lower (i.e., SHDT is higher) than that of the other architectures, the GPGPU has much less data to transfer (HDT3). Thus the communication term (HDT3.SHDT3) is essentially lower than that of the other architectures. Moreover, the computation term (FP.SFP) is much lower than the other architectures, since the SFP is the smallest among all the architectures allowing the GPGPU to perform better than the other architectures for the Morris-Lecar model. From

the perspective of the Roofline model, we also see that the Morris-Lecar model is computation bound for all of the architectures. Using the *Fitness* and Roofline model, a similar explanation applies for the performance of the HH neuron model.

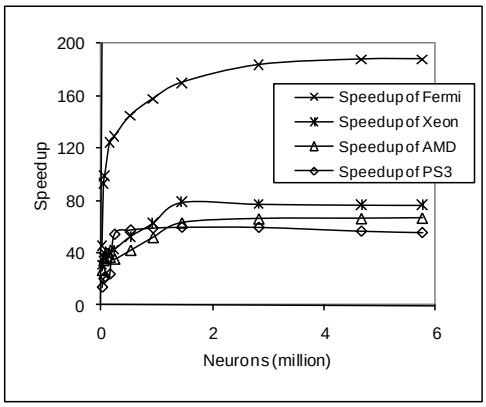

Figure 8.3. Morris-Lecar model: Speedup performance of the four architectures over Intel Core 2 Quad

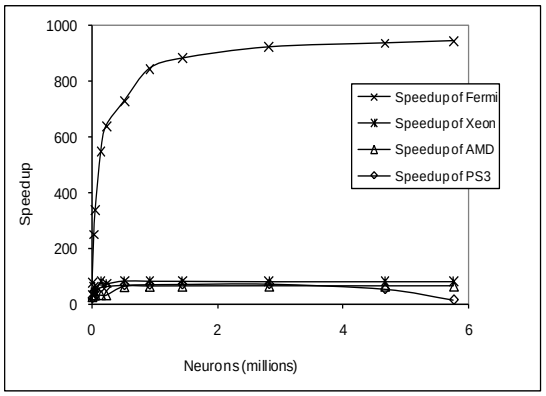

Figure 8.4. Hodgkin-Huxley model: Speedup performance of the four architectures over Intel Core 2 Quad

8.4 Summary

In this Chapter we have validated the Fitness model, multiple regression performance model, and Roofline model using the two case studies: matrix-matrix multiplication and SNN neuron models.

The numerical error of the *Fitness* model as applied to both the case studies can be calculated. For the SNN case study, there were 48 rank predictions for 3 network sizes (4 architectures × 4 models × 3 network sizes). Among them, five of the predicted ranks for the PS3 did not match with the experimental result. Thus predicting the ranks in this case study produces the error rate of (5×100/48), i.e. 10%. On the other hand, for the matrix-

matrix multiplication case study, there are five matrix sizes used, thus, there were 20 rank predictions (4 architectures × 5 matrix sizes). Two of the predicted ranks for the PS3 did not match with the experimental results. Thus the error rate is (2×100/20), i.e., 10%. Thus it can be summarized that the accuracy of the *Fitness* model is 90% accurate. It is also noted that all the errors of rank prediction by the Fitness model occurred with PS3. After further analysis, we find that the limitations of the *Fitness* model are (1) the model does not include the effect of optimization techniques and (2) the complicated bus structure of the PS3, such as EIB congestion, is not included. These limitations of the *Fitness* model are responsible for the error in rank predictions of the PS3.

The error rate for the four regression models is calculated in Section 8.2. We have reported the other six architectures and four programming models in the Appendices B and C respectively. Among the sixteen cases, the maximum error found was 28% for the shortest runtime and the minimum error was 0.27% for the longest runtime. The error varies because a small deviation from the model for shorter runtimes creates a greater percentage of error, while the same deviation for longer runtimes creates a smaller percentage of error. This error pattern suggests that if multiple regression techniques are used on a dataset that has a large range of values for the dependent variable, the prediction for a low-valued dependent variable should be used cautiously, whereas the prediction for a high-valued dependent variable can be used with high confidence. This error pattern is a known limitation of the multiple regression models.

The average error for the sixteen regression models is 11%. The utility of the model does not depend on the error rate alone, the R-squared value and p-values are also an

important indication of the usefulness of the model. The JMP output of the sixteen models is reported in the Appendix A. From this information, we find that the R-squared values are in the range 90% to 99% for all of the models. Thus the R-squared value satisfies the utility criteria for all sixteen models. The p-values also satisfy the utility criteria in 95% of the cases. In the remaining 5% of the cases, though the p-values do not satisfy the utility criteria, the R-squared value does satisfy the utility criteria, and thus we retain the model. Therefore, the combined support from the R-squared and p-values together with the average error (11%) of the sixteen models proves that regression modeling is a very useful technique for projecting the performance of architectures and programming models. In the future work section of Chapter 9 we will discuss potential techniques to improve the regression models and reduce the error.

We have also verified the prediction of maximum performance by the Roofline model after calculating the actual throughput and DRAM bandwidth from the performance graphs shown in Figures 5.1 - 5.4 of Chapter 5. The prediction of the Roofline model was found to be correct after comparing the actual throughput and DRAM bandwidth. In addition to the prediction, the Roofline model also provides information about potential bottlenecks (memory bound or computation bound) of the implementation.

Therefore, three performance models were validated in this Chapter, each having their own advantages and limitations. All three models are useful for a developer while implementing an application using a parallel programming model on an architecture.

Chapter 9

Conclusions And Future Work

The high performance computing community is now experiencing a rapid change in computing technology both in hardware and software. A range of computing architectures and programming models are available for use by the community posing new challenges for the researchers and software developers to select the best hardware and programming models for their applications. To address this issue, this dissertation focuses on performance modeling and analysis of two dominating high performance computing architectures, multicore CPUs and GPGPUs, and the available programming models for these architectures. Two performance models were developed in this research: a *Fitness* model to rank the potential performance of architectures for an application and multiple regression performance models to predict the runtime of an architecture and the performance impact of programming models. To facilitate the development and validation of the performance models, the performance of the multicore and GPGPU architectures was studied for varying problem sizes, optimization techniques, and accelerator configurations and experimental data was collected.

9.1 Summary of Research And Findings

The multicore CPU and GPGPU architectures currently used in high performance computing vary in memory and bus architecture, processing core count, organization, and

clock speed. Thus, performance modeling of these architectures is very useful for the high performance community to enable estimation of impact of architecture and application characteristics on performance. In this research we have developed two performance models for popular multicores and GPGPUs: *Fitness* performance model and multiple regression performance models.

This research proposes the *Fitness* performance model to quickly estimate the performance of accelerators for an application. The *Fitness* model accounts for the architecture characteristics (such as throughput and memory bandwidth) and application characteristics (such as floating point operations, bytes access required) and estimates the performance rank of the architectures based on theoretical performance. This performance rank information can save a substantial amount of development time. For example, Computed Tomography (CT) scanner is equipment used in healthcare requires parallelizing and accelerating the CT algorithms to reconstruct images [109]. To select the best architecture for this algorithm without the aid of a performance model, the developer must implement and test the algorithm on the available architectures, which costs a substantial amount of development time and potentially the cost of hardware and software. Alternately, with the use of the *Fitness* model, the developer can quickly analyze and select the best architecture for CT algorithm, saving time and money. Further, administrators of computing clusters, supercomputers, and computing clouds can use the *Fitness* model to select the best computing nodes for their target applications saving a significant amount of development time and hardware/software costs.

While the *Fitness* model ranks the performance of accelerators, often estimating the actual performance (runtime) of an application on a given architecture is required. By predicting the runtime for an application beforehand, a system administrator can manage the job queue more efficiently; the developer and scientific researchers can estimate when the results from the run are expected. For these scenarios, multiple regression models (a statistical tool) were developed that can predict the actual runtime of an architecture for an application. We also use the regression models to estimate the performance impact of programming models on application performance. Combined, these models help the developer to select the best programming model for a particular combination of hardware and application characterizes so that substantial time is not invested experimenting with all available programming models.

The performance data collected in Chapters 5 and 6 were used to validate the performance models. The dataset includes the impact of problem sizes, optimizations, accelerator configuration, and programming models on performance. Four popular architectures, Intel Xeon, AMD Opteron, IBM PS3, NVIDA Fermi, and four advanced architectures, Intel 32 core, AMD 32 core, IBM 32 core, and SUN 16 core were used. The programming models used include p-threading, OpenMP, Concurrency Runtime, and OpenCL for the x86 architectures, and CUDA and OpenCL for Nvidia GPGPU.

Some interesting results were observed while collecting the performance data for the architecture and programming model combinations. For example, while varying the problem size, a peak in performance (maxima) for the x86 accelerators and a dramatic performance decrease in the PS3 performance for larger problem sizes was observed.

These deviations in performance were explained in Chapter 5. Other cases where the performance of an architecture saturates or decreases as the number of processing cores increases were documented. Moreover, the performance impact with the increase of processing cores was found to be dependent on the correlation of the application characteristics, such as flop/byte ratio, and the architecture characteristics as explained in Chapter 5.

In Chapter 6, the performance impact of four multicore programming models and two GPGPU programming models was investigated. We qualitatively rank the programming models in terms of the required programming effort by the developer. Attempts are also taken to quantify the programming effort required for the programming models. Thus together with the regression models for programming models, the above analysis is helpful for selecting the best programming model in terms of both performance and programming effort.

9.2 Contributions

This research provides four significant contributions to the paradigm of high performance computing:

i) *Fitness* performance model is proposed that predicts the performance rank of architectures for an application. The *Fitness* model accounts for the characteristics of both the architectures and applications and predicts. The model is validated using experimental performance data collected for two kinds of applications: matrix-matrix multiplications and four SNN neuron models.

ii) Multiple regression models for the architectures are proposed that predict the actual runtime. Multiple regression models for each architecture have been developed using the experimental performance dataset collected. The architectures considered include three popular multicore CPUs one GPGPU, and four advanced multicore processors.

iii) Multiple regression models to predict the performance impact of the programming models are also developed. Four multicore programming models and two GPGPU programming models are modeled with multiple regression techniques. These regression models are validated using the experimental performance dataset. Additionally, attempts are taken to qualitatively and quantitatively compare the programming efforts required by the developer. The programming models are ranked in terms of the programming effort.

iv) Performance analysis of leading multicore and GPGPU processors. We have presented and analyzed the performance of Intel Xeon, AMD Opteron, and IBM PS3, Nvidia Fermi GPGPU for varying problem size, optimization techniques, processing core count, and flop/byte ratio. The performance of advanced many-core processors, Intel 32 core, AMD 32 core, IBM 16 core, and Sun T2+ UltraSparc, is also analyzed. The performance analysis was conducted to collect experimental data for the development and validation of the performance models. However, these analyses also provide valuable

insights on the interaction between architecture and application characteristics.

9.3 Future Work

In future, the following issues can be addressed:

9.3.1 Use of micro-benchmark values with the *Fitness* model

We have introduced the *Fitness* model *for si*ngle-node configuration. While validating the model we have used theoretical peak values for various performance metrics, such as throughput and DRAM bandwidth. But to achieve a theoretical peak, all optimization techniques must be applied, which is not practical for most cases. Thus in future, we propose to investigate the validation of the *Fitness* model using micro-benchmark values such as the STREAM benchmark values.

9.3.2 Validation with others applications and accelerators

In the study of architecture performance and in the validation of *Fitness* and multiple regression models we have used a matrix multiplication algorithm and four types of SNN models as case studies. The computation and communication demand of the case studies covers a wide range making them good candidates to study the performance of the architectures. However, we propose to use other applications so that that *Fitness* model can be further validated over a wider range of applications. Potential application studies include but are not limited to: molecular dynamics kernels, bioinformatics kernels, and optimization problems.

9.3.3 Effect of operating systems

The experimental results presented include the performance impact of programming models on application performance. Future work will include an investigation of the performance impact of the operating system, such as Windows and Linux, on application performance.

9.3.4 Comparison between the two performance models

The basic linear regression model (for which no squared and interaction terms are present) can be compared with the Fitness model to further validate and rationalize both models. By setting the intercept term of the regression model (α_1) as zero, the two models can be directly compared if the linear regression model for architecture is redeveloped. Initial investigations have found that in some cases, one of the coefficients for the regression model is negative which is physically not possible. There are three potential reasons for this issue that need further investigation. First, the number of data points is inadequate compared to the range of values for the dependent variables (0.2 msec. to 15000 ms). Second, additional terms (interaction, squared, cubed) may be needed to meet the utility critera. Third, some of the data points with extremely low values for the dependent variable are also negatively affecting the balance of the regression model. By addressing these three issues, the regression model can be modified then compared with the respective coefficients of the Fitness model.

The rank prediction capability (i.e. error) of the two models can also be compared. Initial investigations found that except for the smallest problem size, the regression

model predicts the rank accurately. For the smallest problem size, the regression model predicts negative values for the Intel Xeon runtime, which is not practical and invalidates the rank prediction. Similar reasons as described in the previous paragraph are potentially responsible for this behavior. Thus, by addressing these three issues, the rank prediction for the smallest problem sizes can potentially be corrected.

An additional reason for the above limitations of the regression model is that it does not have the required interaction, squared, cubed, or other powered terms. When adding terms to the model to correct its coefficients and prediction, we also must know when to stop adding terms. The Akaike Information Criterion (AIC) is a well-known criteria for discontinuing the addition of terms to the model. By applying this criteria in the future work, one will know when there is a sufficient number of terms in the regression model.

9.3.5 Modeling of heterogeneous computing nodes

To model the performance of heterogeneous computing nodes (multicore CPU and GPGPU), extension to the *Fitness* model are needed. The two architectures in a heterogeneous system must be modeled separately along with possible other terms for application load balance, background load, communication between the two architectures, etc.

Bibliography

[1] http://www.eetimes.com/electronics-products/processors/4091586/Intel-s-teraflops-chip-uses-mesh-architecture-to-emulate-mainframe

[2] http://www.tilera.com/products/processors

[3] http://www.NVIDA.com/content/global/global.php

[4] http://www.amd.com/us/Pages/AMDHomePage.aspx

[5] http://sc10.supercomputing.org/

[6] http://isscc.org/doc/2011/isscc2011.advanceprogramabstracts.pdf

[7] http://www.amd.com/us/press-releases/Pages/16-core-interlagos-2010nov16.aspx

[8] "NVIDA CUDA Programming Guide", http://developer.download.NVIDA.com/ compute/cuda/3_0/toolkit/docs/NVIDA_CUDA_ProgrammingGuide.pdf

[9] L. Ligowski and W. Rudnicki, "An Efficient Implementation of Smith Waterman Algorithm on GPGPU using CUDA, for Massively Parallel Scanning of Sequence Databases." *Proceedings of IPDPS 2009*, Rome Italy, May 2009

[10] http://www.ks.uiuc.edu/Research/namd/

[11] http://lammps.sandia.gov/

[12] https://simtk.org/home/openmm

[13] http://www.khronos.org/opencl/

[14] D. Pellerin, and S. Thibault. 2005. Practical FPGA Programming in C. *Prentice Hall Press*.

[15] SRC Carte C Programming Environment. SRC Computers.

[16] http://www.gidel.com/

[17] http://www.xtremedata.com/

[18] http://www.cray.com/products/Legacy.aspx

[19] http://www.nallatech.com/

[20] Low power hybrid computing for efficient software acceleration. http://www.mitrion.com/?document=Hybrid-Computing-Whitepaper.pdf

[21] Agility Design Solutions 2007. Handel-C Language Reference Manual. Agility Design Solutions, http://www.agilityds.com/literature/HandelC Language Reference Manual.pdf.

[22] S. Williams, A. Waterman, D. Patterson. “Roofline: An Insightful Visual Performance Model for Floating-Point Programs and Multicore Architectures,” *Communications of the ACM (CACM),* 65-76, April 2009.

[23] B. Holland, A. George, H. Lam, and M. Smith, "An Analytical Model for Multi-Level Performance Prediction of Multi-FPGA Systems," *ACM Transactions on Reconfigurable Technology and Systems (TRETS)*, to appear.

[24] C. Johansson and A. Lansner, “Towards Cortex Sized Artificial Neural Systems,” *Neural Networks*, 20(1), 48–61, Jan. 2007.

[25] E. Izhikevich and G. Edelman, "Large-Scale Model of Mammalian Thalamocortical Systems," *Proceedings of the National Academy of Sciences,* 105(9), 3593–3598, Mar. 2008.

[26] E. M. Izhikevich. Dynamical Systems in Neuroscience, *MIT press*, Cambridge, Massachusetts, 2007.

[27] M. A. Bhuiyan, T. M. Taha, R. Jalasutram, “Character recognition with two spiking neural network models on multicore architectures,” *IEEE Proceedings of CIMSVP*, Tennessee, 29-34, March 2009.

[28] A. Chandramowlishwaran, S. Williams, L. Oliker, I. Lashuk, G. Biros, R. Vuduc,”Optimizing and Tuning the Fast Multipole Method for State-of-the-Art Multicore Architectures,” *Proceedings of International Parallel & Distributed Processing Symposium (IPDPS)*, GA, 1-12, 2010.

[29] S. Williams, J. Carter, L. Oliker, J. Shalf, K. Yelick,”Optimization of a Lattice Boltzmann Computation on State-of-the-Art Multicore Platforms,” *Journal of Parallel and Distributed Computing (JPDC)*, 69(9), 762-777, Sep. 2009.

[30] K. Datta, S. Williams, V. Volkov, J. Carter, L. Oliker, J. Shalf, K. Yelick, “ Auto-tuning the 27-point Stencil for Multicore,” *4th International Workshop on Automatic Performance Tuning (iWAPT)*, 2009.

[31] M. Araya-Polo, J. Cabezas, M. Hanzich, M. Pericas, F. Rubio, I. Gelado, M. Shafiq, E. Morancho, N. Navarro, E. Ayguade, J. M. Cela, M. Valero, “Assessing Accelerator-based HPC Reverse Time Migration,” *IEEE Transactions on Parallel and Distributed Systems*, 99, PrePrints, 2010.

[32] A. Khajeh-Saeed, S. Poole, J. B. Perot, “Acceleration of the Smith–Waterman algorithm using single and multiple graphics processors,” *Journal of Computational Physics*, 229(11), 4247-4258, Jun. 2010.

[33] J. M. Nageswaran, N. Dutt, J. L. Krichmar, A. Nicolau, A. Veidenbaum, “A configurable simulation environment for the efficient simulation of large-scale spiking neural networks on graphics processor,” *Special issue of Neural Network*, Elsevier, 22(5-6), 791-800, July 2009.

[34] S. Williams, N. Bell, J.W. Choi, M. Garland, L. Oliker. R. Vuduc, "Sparse Matrix Vector Multiplication on Multicore and Accelerators", *in Scientific Computing on Multicore and Accelerators*, CRC Press, ISBN: 978-1-4398253-6-5, 2010.

[35] S. Hong, H. Kim, "An Analytical Model for a GPGPU Architecture with Memory-level and Thread-level Parallelism Awareness," *Proceedings of the 36th International Symposium on Computer Architecture (ISCA)*, Austin, TX, June 2009.

[36] S. S. Baghsorkhi, M. Delahaye, S. J. Patel, W. D. Gropp, W. W. Hwu, "An Adaptive Performance Modeling Tool for GPGPU Architectures," *Proceedings of the 15th ACM SIGPLAN, symposium on Principles and practice of parallel programming*, India, 105-114, 2010.

[37] S. S. Baghsorkhi, M. Delahaye, S. J. Patel, W. D. Gropp, W. W. Hwu, "An Adaptive Performance Modeling Tool for GPGPU Architectures," *Proceedings of the 15th ACM SIGPLAN, symposium on Principles and practice of parallel programming*, India, 2010.

[38] Xi E. Chen, T. M. Aamodt, "A First-Order Fine-Grained Multithreaded Throughput Model," *Proceedings of HPCA*, 329-340, 2009.

[39] D. B. Noonburg and J. P. Shen, "A Framework for Statistical Modeling of Superscalar Processor Per-formance," *International Symposium on High Performance Computer Architecture*, pp. 298-309, 1997.

[40] M. C. Smith, "Analytical Modeling of High Performance Reconfigurable Computers: Prediction and Analysis of System Performance," *Ph. D. Dissertation*, The University of Tennessee, Knoxville, December 2003.

[41] B. Holland, K. Nagarajan, and A. D. George, "RAT: RC amenability test for rapid performance prediction," *ACM Transactions on Reconfigurable Technology and Systems (TRETS)*, 1(4), 1–31, 2009

[42] A. Alexandrov, M. F. Ionescu, K. E. Schauser, and C. Scheiman, "LogGP: In-corporating long messages into the LogP model for parallel computation," *J. Paral. Distrib.Comput.*, 44(1), 71-79, 1997.

[43] A. Snavely , L. Carrington , N. Wolter , J. Labarta , R. Badia , A. Purkayastha, "A framework for performance modeling and prediction," *Proceedings of the ACM/IEEE conference on Supercomputing*, Baltimore, Maryland , 1-17, Nov. 2002.

[44] J. Archuleta, Y. Cao, T. Scogland and Wu-chun Feng, "Multi-dimensional characterization of temporal data mining on graphics processors," *Presented at 2009 IEEE International Symposium on Parallel & Distributed Processing (IPDPS).* Available: http://dx.doi.org/10.1109/IPDPS.2009.5161049

[45] O. D. Lampe, I. Viola, N. Reuter and H. Hauser, " Two-level approach to efficient visualization of protein dynamics," *IEEE Trans. Visual. Comput. Graphics*,13(6), pp. 1616-1623. Available: http://dx.doi.org/10.1109/TVCG.2007.70517

[46] S. Sur, M. J. Koop, Lei Chai and D. K. Panda, "Performance analysis and evaluation of mellanox ConnectX InfiniBand architecture with multicore platforms," *15th Annual IEEE Symposium on High-Performance Interconnects*, 125-134, 2007.

[47] S. R. Alam, P. K. Agarwal, S. S. Hampton, Hong Ong and J. S. Vetter, "Impact of multicores on large-scale molecular dynamics simulations," *IEEE International Symposium Parallel and Distributed Processing(IPDPS 2008)*, 1-7, 2008

[48] P. M. Martin, M. C. Smith, S. Alam and P. Agarwal, “Implementation methodology for emerging reconfigurable systems,” *51st Midwest Symposium Circuits and System (MWSCAS 2008)* 169-172, 2008

[49] L. Vogt, R. Olivares-Amaya, S. Kermes, Y. Shao, Carlos Amador-Bedolla, and, Alán Aspuru-Guzik, “Accelerating Resolution-of-the-Identity Second-Order Moller−Plesset Quantum Chemistry Calculations with Graphical Processing Units,” *The Journal of Physical Chemistry A,* 112 (10), 2049-2057, 2008.

[50] I. S. Ufimtsev and T. J. Martinez, “Quantum chemistry on graphical processing units. 1. strategies for two-electron integral evaluation,” *J. Chem. Theory Comput*, 4(2), 222–231, 2008

[51] I. S. Ufimtsev and T. J. Martinez, “Quantum Chemistry on Graphical Processing Units. 2. Direct Self-Consistent-Field Implementation,” *Journal of Chemical Theory and Computation,* vol. 5, no. 4, pp. 1004–1015, April 2009. [Online]. Available: http://dx.doi.org/10.1021/ct800526s

[52] Y. Allusse, P. Horain, A. Agarwal and C. Saipriyadarshan, “GPGPUCV: A GPGPU-accelerated framework for image processing and computer vision,” *Presented at 4th International Symposium,(ISVC 2008),* Available: http://dx.doi.org/10.1007/978-3-540-89646-3_42

[53] D. Castano-Diez, D. Moser, A. Schoenegger, S. Pruggnaller and A. S. Frangakis, “Performance evaluation of image processing algorithms on the GPGPU,” *Journal of Stuctural Biology,* 164(1), pp. 153-60. Available: http://dx.doi.org/10.1016/j.jsb.2008.07.006

[54] K. Fok, T. Wong and M. Wong, “Evolutionary computing on consumer graphics hardware” *IEEE Intelligent Systems,* 22(2), pp. 69-78. Available: http://dx.doi.org/10.1109/MIS.2007.28

[55] H. Markram. The Blue Brain Project, *Nature Reviews*, Neuroscience, 7, 153-160, 2006.

[56] W. Rall, “Branching dendritic trees and motoneuron membrane resistivity,” *Experimental Neurology*, 1, 1959; 503-532.

[57] R. Ananthanarayanan, S. K. Esser, H. D. Simon, and D. S. Modha, “The Cat is Out of the Bag: Cortical Simulations with 109 Neurons, 1013 Synapses,” *Proc. of SC*, Oregon, Nov. 2009.

[58] M. M. Khan, D. R. Lester, L. A. Plana, A. D. Rast, X. Jin, E. Painkras, and S B. Furber, ‘‘SpiNNaker: Mapping neural networks onto a massively-parallel chip multiprocessor,’’ *In International Joint Conference on Neural Networks (IJCNN),* 2849-2856, 2008.

[59] D. B. Strukov, G. S. Snider, D. R. Stewart, R. S. Williams, "The missing memristor found," *Nature,* 453, 80-83, 2008.

[60] B. Linares-Barranco and T. Serrano-Gotarredona, “Memristance can explain Spike-Time-Dependent-Plasticity in Neural Synapses,” [Online], Available from Nature Proceedings, http://hdl.handle.net/10101/npre.2009.3010.1, 2009

[61] C. Gao and D. Hammerstrom. "Cortical Models Onto CMOL and CMOS—Architectures and Performance/Price," *IEEE Transactions on Circuits and Systems,* 54(11), 2502-2515, Nov. 2007.

[62] H. Shayani, P.J. Bentley, and A. M. Tyrrell, “Hardware Implementation of a Bio-plausible Neuron Model for Evolution and Growth of Spiking Neural Networks on FPGA”, *in NASA/ESA Conference on Adaptive Hardware and Systems 2008 (AHS ’08),* 236-243, Jun. 2008.

[63] A. Upegui, C.A. Pena-Reyes, E. Sanchez, “A hardware implementation of a network of functional spiking neurons with hebbian learning,” *in International Workshop on Biologically Inspired Approaches to Advanced Information Technology 2004 (BioAdit 04)*, 399-409, Jan. 2004.

[64] E.Izhikevich, “Which Model to Use for Cortical Spiking Neurons?” *IEEE Transactions on Neural Networks*, 15(5), 1063-1070, 2004.

[65] M. Rosa, E.Caruso, L. Fortuna, M. Frasca, L. Occhipinti, and F. Rivoli, “Neuronal dynamics on FPGA: Izhikevich's model,” *Proceedings of the SPIE*, 5839, 87-94, 2005.

[66] K. L. Rice, M. A. Bhuiyan, T. M. Taha, C. N. Vutsinas, and M. C. Smith, “FPGA Implementation of Izhikevich Spiking Neural Networks for Character Recognition”, *Reconfig 09*, Cancun, Mexico, Dec. 2009.

[67] R. K. Weinstein, M. S. Reid, and R. H. Lee, "Methodology and design flow for assisted neural-model implementations in FPGAs," *IEEE Transactions on Neural Systems and Rehabilitation Engineering*, 15(1), 83-93, March 2007.

[68] M. Gschwind, H. P. Hofstee, B. Flachs, M. Hopkins, Y. Watanabe, T. Yamazaki, “Synergistic Processing in Cell’s Multicore Architecture,” *IEEE Micro*, 26(2), 10-24, Mar. 2006.

[69] http://www.nvidia.com/object/personal-supercomputing.html

[70] “ATI Mobility Radeon HD 5870 GPGPU Specifications”, http://www.amd.com/us/products/notebook/graphics/ati-mobility-hd-5800/Pages/hd-5870-specs.aspx

[71] http://www.srccomp.com/

[72] E.M. Izhikevich, "Polychronization: Computation With Spikes," *Neural Computation*, 1, 8245-282, 2006.

[73] A. L. Hodgkin and A. F. Huxley, “A quantitative description of membrane current and application to conduction and excitation in nerve,” *Journal of Physiology*, 117, 500–544, 1952.

[74] C. Morris and H. Lecar, “Voltage oscillations in the barnacle giant muscle fiber,” *Biophys. J.*, 35, 193–213, 1981.

[75] H. R. Wilson, “Simplified dynamics of human and mammalian neocortical neurons,” *J. Theor. Biol.*, 200, 375–388, 1999.

[76] E. Izhikevich, “Simple Model of Spiking Neurons,” *IEEE Transaction on Neural Networks*, 14(6), 1569-1572, Nov. 2003.

[77] A. Gupta, L. Long, “Character Recognition using Spiking Neural Networks,” *Proceedings of IJCNN*, 53-58, Aug. 2007.

[78] M. A. Bhuiyan, V. K. Pallipuram, M. C. Smith, “Acceleration of Spiking Neural Networks in Emerging Multicore and GPGPU Architectures,” *IEEE proceedings of HiCOMB*, IPDPS, Atlanta, GA, 1-8, April 2010.

[79] “ATI Stream Profiler”, http://developer.amd.com/GPGPU/StreamProfiler/Pages/default.aspx

[80] "Stream KernelAnalyzer", http://developer.amd.com/GPGPU/ska/pages/default.aspx

[81] https://computing.llnl.gov/tutorials/pthreads/

[82] http://openmp.org/wp/

[83] http://msdn.microsoft.com/en-us/library/dd504870.aspx

[84] http://www.khronos.org/opencl/

[85] http://www.nvidia.com/object/cuda_home_new.html

[86] P. J. Cagnard, "The parallel cellular programming model," *8th Euromicro workshop on Parallel and Distributed Processing*, 2000. DOI: 10.1109/EMPDP.2000.823384

[87] M. Vance, "A migration-based parallel programming model with architectural support structures," *Presented at 2009 DoD High Performance Computing Modernization Program Users Group Conference*.

[88] R. B. Brightwell , M. A. Heroux , Z. Wen , "J. Wu, Parallel Phase Model: A Programming Model for High-End Parallel Machines with Manycores," *International Conference on Parallel Processing*,, Austria, 2009.

[89] F. E. Fich, The complexity of computation on the parallel random access Machine. *Synthesis of Parallel Algorithms*, Morgan Kaufmann Publ., San Mateo, CA, 1993

[90] http://www.lam-mpi.org/

[91] http://www.bsp-worldwide.org/

[92] L. Prechelt, "A parallel programming model for irregular dynamic neural networks," *Presented at Third Working Conference on Massively Parallel Programming Models*, 214-219, 1997.

[93] B. Corda and K. H. Warren, "PFP: A scalable parallel programming model" *Presented at Scalable High Performance Computing Conference*, 170-173, 1992.

[94] W. Hwu, S. Ryoo, S. Ueng, J. H. Kelm, I. Gelado, S. S. Stone, R. E. Kidd, S. S. Baghsorkhi, A. A. Mahesri, S. C. Tsao, N. Nvarro, S. S. Lumetta, M. I. Frank, and S. J. Patel, "Implicitly parallel programming models for thousand-core microprocessors" *In* Proceedings of the 44th annual Design Automation Conference *(DAC '07)*. ACM, New York, NY, USA, 754-759. DOI=10.1145/1278480.1278669

[95] M. Sawley, J. Tegner, "A comparison of parallel programming-models for multiblock flow computations," *J Comput Phys*, 122(2), 280-90, 1995.

[96] J. A. Keane, A. J. Grant, M. Q. Xu, "Comparing distributed memory and virtual shared memory parallel programming models" *Future Generation Computer Systems*, 11(2), 233-43, March 1995.

[97] R. Rabenseifner and G. Wellein, "Communication and optimization aspects of parallel programming models on hybrid architectures," *International Journal of High Performance Computing Applications*, 17(1), 49-62, Feb. 2003

[98] L. Hochstein, V. R. Basili, U. Vishkin, J. Gilbert, “A pilot study to compare programming effort for two parallel programming models,” *Journal of Systems and Software*, 8(11), 1920-1930, Nov. 2008

[99] A. Marowka, “Towards high-level parallel programming models for multicore systems,” *International Conference on Advanced Software Engineering & Its Applications*, 226-9, 2008

[100] I. Patel, J. R. Gilbert, "An empirical study of the performance and productivity of two parallel programming models," *Parallel and Distributed Processing, 2008. IPDPS 2008,* 14-18 April 2008

[101] SARC European Project, "Parallel Programming Models for Heterogeneous Multicore Architectures," *Micro, IEEE* , 30(5), 42-53, Sept.-Oct. 2010

[102] K. Fatahalian, J. Sugerman and P. Hanrahan, "Understanding the efficiency of GPU algorithms for matrix-matrix multiplication," *in Proceedings of the ACM SIGGRAPH/EUROGRAPHICS conference on Graphics hardware*, 133-137, 2004.

[103] R. A. Van De Geijn and J. Watts, "SUMMA: Scalable universal matrix multiplication algorithm," *Journal of Concurrency and Computation: Practice and Experience*, 9, 255-274, 1997

[104] K. Goto and R.A. Geijn, "Anatomy of high-performance matrix multiplication," *ACM Transactions on Mathematical Software (TOMS)*, 34, 12, 2008

[105] http://software.intel.com/en-us/articles/intel-mkl/

[106] http://developer.amd.com/libraries/acml/pages/default.aspx

[107] http://developer.download.nvidia.com/compute/cuda/1_0/CUBLAS_Library_1.0.pdf

[108] https://www.ibm.com/developerworks/mydeveloperworks/blogs/powerarchitecture/entry/ibomb_blas_sdk30_2?lang=en

[109] F. Xu and K. Mueller, “RapidCT: Acceleration of 3D Computed Tomography on GPUs,” *ACM Workshop on General-Purpose Computing on Graphics,* 2004.

[110] A. Snavely, N.Wolter, and L. Carrington, “Modeling Application Performance by Convolving Machine Signatures with Application Profiles. *In WWC ’01: Proceedings of the Workload Characterization, 2001,* Washington DC, 149–156, 2001.

[111] L. C. Carrington, X. Gao, N. Wolter, A. Snavely, and R. L. Jr. Campbell, “Performance Sensitivity Studies for Strategic Applications,” *In DOD UGC ’05: Proceedings of the 2005 Users Group Conference* , page 400, Washington DC, 2005.

[112] V. K. Pallipuram, M. A. Bhuiyan, M. C. Smith, “A Comparative Study of GPU Programming Models and Architectures using Neural Networks,” *Journal of SuperComputing*, Springer Publications, DOI 10.1007/s11227-011-0631-3.

[113] G. Khanna and J. McKennon, “Numerical modeling of gravitational wave sources accelerated by OpenCL”, *Computer Physics Communications*,181(9), 1605-1611, 2010.

[114] K. Karimi, N.G. Dickson, and F. Hamze, “A Performance Comparison of CUDA and OpenCL”, *The Computing Research Repository (CoRR)*, abs/1005.2581, 2010.

[115] P. Du, R. Weber, S. Tomov, G. Peterson, and J. Dongarra, “From CUDA to OpenCL: Towards a Performance-portable Solution for Multi-platform GPU Programming”, *Parallel Computing (Submitted), Elsevier Publications*, 2010.

[116] V. K. Pallipuram, “Acceleration of Spiking Neural Networks on Single-GPU and Multi-GPU systems”, Master’s Thesis, Clemson University, May 2010.

[117] http://threadingbuildingblocks.org/

[118] W. Mendenhall, T. Sincich, A Second Course in Statistics: Regression Analysis, Sixth Edition, *Pearson Education*, New Jersey, 2003.

[119] http://www.sas.com/

APPENDIX

APPENDIX A

JMP software output for each of the eight accelerators and six programming models

A1. Nvidia Fermi

Response Runtime

Summary of Fit

RSquare	0.962364
RSquare Adj	0.955307
Root Mean Square Error	60.50031
Mean of Response	167.6846
Observations (or Sum Wgts)	20

Analysis of Variance

Source	DF	Sum of Squares	Mean Square	F Ratio
Model	3	1497513.5	499171	136.3748
Error	16	58564.6	3660	Prob > F
C. Total	19	1556078.1		<.0001*

Parameter Estimates

Term	Estimate	Std Error	t Ratio	Prob>|t|
Intercept	36.223629	16.65145	2.18	0.0449*
Total flops	-1.211e-8	2.733e-9	-4.43	0.0004*
Total Bytes	8.7684e-8	1.661e-8	5.28	<.0001*
(Total Bytes-7.5e+9)*(Total flops-4.3e+10)	-3.41e-21	5.32e-21	-0.64	0.5307

A2. Intel Xeon

Response Runtime

Summary of Fit

RSquare	0.914967
RSquare Adj	0.910038
Root Mean Square Error	917.2831
Mean of Response	1367.911
Observations (or Sum Wgts)	74

Analysis of Variance

Source	DF	Sum of Squares	Mean Square	F Ratio
Model	4	624707015	156176754	185.6135
Error	69	58057169	841408.25	Prob > F
C. Total	73	682764184		<.0001*

Lack Of Fit

Source	DF	Sum of Squares	Mean Square	F Ratio
Lack Of Fit	64	58049181	907018	567.7381
Pure Error	5	7988	1598	Prob > F
Total Error	69	58057169		<.0001*
				Max RSq
				1.0000

Parameter Estimates

Term	Estimate	Std Error	t Ratio	Prob>|t|
Intercept	3033.8719	318.3726	9.53	<.0001*
Total flops	2.6175e-8	9.671e-9	2.71	0.0086*
Total Bytes	1.2332e-7	5.996e-8	2.06	0.0435*
Cores Used	-479.0659	44.75155	-10.71	<.0001*
(Cores Used-6.37838)*(Total flops-3.2e+10)	-1.414e-8	1.086e-9	-13.02	<.0001*

A3. AMD Opteron

Response Runtime

Summary of Fit

RSquare	0.990042
RSquare Adj	0.989465
Root Mean Square Error	284.7947
Mean of Response	1124.099
Observations (or Sum Wgts)	74

Analysis of Variance

Source	DF	Sum of Squares	Mean Square	F Ratio
Model	4	556410225	139102556	1715.029
Error	69	5596452	81108.005	Prob > F
C. Total	73	562006678		<.0001*

Lack Of Fit

Source	DF	Sum of Squares	Mean Square	F Ratio
Lack Of Fit	64	5596229.3	87441.1	1959.765
Pure Error	5	223.1	44.6	Prob > F
Total Error	69	5596452.4		<.0001*
				Max RSq
				1.0000

Parameter Estimates

Term	Estimate	Std Error	t Ratio	Prob>|t|
Intercept	3181.2476	234.2097	13.58	<.0001*
Total flops	4.0678e-8	4.301e-9	9.46	<.0001*
Total Bytes	1.3013e-7	1.783e-8	7.30	<.0001*
Cores	-495.5322	39.07628	-12.68	<.0001*
(Total flops-2.6e+10)*(Cores-6.37838)	-1.963e-8	1.928e-9	-10.18	<.0001*

A4. PS3

Response Runtime

Summary of Fit

RSquare	0.991945
RSquare Adj	0.991521
Root Mean Square Error	126.8124
Mean of Response	527.2285
Observations (or Sum Wgts)	41

Analysis of Variance

Source	DF	Sum of Squares	Mean Square	F Ratio
Model	2	75255164	37627582	2339.823
Error	38	611092	16081.377	Prob > F
C. Total	40	75866256		<.0001*

Parameter Estimates

Term	Estimate	Std Error	t Ratio	Prob>|t|
Intercept	45.708852	23.14433	1.97	0.0556
Total flops	2.3041e-8	1.166e-9	19.76	<.0001*
Total Bytes	5.1863e-8	1.287e-8	4.03	0.0003*

A5. Intel 32 core

Response Runtime

Summary of Fit

RSquare	0.977242
RSquare Adj	0.975699
Root Mean Square Error	2037.539
Mean of Response	4012.625
Observations (or Sum Wgts)	64

Analysis of Variance

Source	DF	Sum of Squares	Mean Square	F Ratio
Model	4	1.0518e+10	2.6295e+9	633.3778
Error	59	244942452	4151567	Prob > F
C. Total	63	1.0763e+10		<.0001*

Parameter Estimates

Term	Estimate	Std Error	t Ratio	Prob>|t|
Intercept	31654.395	1975.783	16.02	<.0001*
Total flops	3.692e-8	2.172e-8	1.70	0.0945
Total Bytes	5.0634e-8	1.328e-7	0.38	0.7045
Cores Used	-33781.67	2082.808	-16.22	<.0001*
(Cores Used-0.93446)*(Total flops-8.5e+10)	-3.88e-7	1.124e-8	-34.50	<.0001*

A6. AMD 32 core

Response Runtime

Summary of Fit

RSquare	0.985541
RSquare Adj	0.984561
Root Mean Square Error	2257.19
Mean of Response	5557.98
Observations (or Sum Wgts)	64

Analysis of Variance

Source	DF	Sum of Squares	Mean Square	F Ratio
Model	4	2.0489e+10	5.1222e+9	1005.362
Error	59	300599536	5094907.4	Prob > F
C. Total	63	2.079e+10		<.0001*

Parameter Estimates

Term	Estimate	Std Error	t Ratio	Prob>|t|
Intercept	43595.437	2188.776	19.92	<.0001*
Total flops	4.5518e-8	2.407e-8	1.89	0.0635
Total Bytes	1.0981e-7	1.472e-7	0.75	0.4585
Cores Used	-46664.03	2307.339	-20.22	<.0001*
(Total flops-8.5e+10)*(Cores Used-0.93446)	-5.397e-7	1.246e-8	-43.33	<.0001*

A7. IBM 16 core

Response Runtime

Summary of Fit

RSquare	0.987974
RSquare Adj	0.987158
Root Mean Square Error	1630.643
Mean of Response	4943.58
Observations (or Sum Wgts)	64

Analysis of Variance

Source	DF	Sum of Squares	Mean Square	F Ratio
Model	4	1.2888e+10	3.222e+9	1211.721
Error	59	156880720	2658995.3	Prob > F
C. Total	63	1.3045e+10		<.0001*

Parameter Estimates

Term	Estimate	Std Error	t Ratio	Prob>|t|
Intercept	32394.884	1599.549	20.25	<.0001*
Total flops	3.6727e-8	1.739e-8	2.11	0.0389*
Total Bytes	1.1103e-7	1.063e-7	1.04	0.3006
Cores Used	-34784.94	1698.092	-20.48	<.0001*
(Total flops-8.5e+10)*(Cores Used-0.92823)	-4.04e-7	9.168e-9	-44.07	<.0001*

A8. Sun 32 core

Response Runtime

Summary of Fit

RSquare	0.969886
RSquare Adj	0.967844
Root Mean Square Error	50630.05
Mean of Response	86749.55
Observations (or Sum Wgts)	64

Analysis of Variance

Source	DF	Sum of Squares	Mean Square	F Ratio
Model	4	4.871e+12	1.218e+12	475.0490
Error	59	1.5124e+11	2.5634e+9	Prob > F
C. Total	63	5.0222e+12		<.0001*

Parameter Estimates

Term	Estimate	Std Error	t Ratio	Prob>|t|
Intercept	786328.74	49233.42	15.97	<.0001*
Total flops	1.6402e-6	1.129e-6	1.45	0.1517
Total Bytes	9.4354e-7	6.94e-6	0.14	0.8923
Cores Used	-841157.7	51745.42	-16.26	<.0001*
(Cores Used-0.9345)*(Total flops-4.8e+10)	-1.735e-5	5.616e-7	-30.89	<.0001*

A9. CUDA on Fermi

Response Runtime

Summary of Fit

RSquare	0.962364
RSquare Adj	0.955307
Root Mean Square Error	60.50031
Mean of Response	167.6846
Observations (or Sum Wgts)	20

Analysis of Variance

Source	DF	Sum of Squares	Mean Square	F Ratio
Model	3	1497513.5	499171	136.3748
Error	16	58564.6	3660	Prob > F
C. Total	19	1556078.1		<.0001*

Parameter Estimates

Term	Estimate	Std Error	t Ratio	Prob>|t|
Intercept	36.223629	16.65145	2.18	0.0449*
Total flops	-1.211e-8	2.733e-9	-4.43	0.0004*
Total Bytes	8.7684e-8	1.661e-8	5.28	<.0001*
(Total Bytes-7.5e+9)*(Total flops-4.3e+10)	-3.41e-21	5.32e-21	-0.64	0.5307

A10. OpeCL on Fermi

Response Runtime

Summary of Fit

RSquare	0.979972
RSquare Adj	0.976216
Root Mean Square Error	45.54474
Mean of Response	162.4022
Observations (or Sum Wgts)	20

Analysis of Variance

Source	DF	Sum of Squares	Mean Square	F Ratio
Model	3	1623914.4	541305	260.9549
Error	16	33189.2	2074	Prob > F
C. Total	19	1657103.6		<.0001*

Parameter Estimates

Term	Estimate	Std Error	t Ratio	Prob>|t|
Intercept	28.802217	12.53524	2.30	0.0354*
Total flops	-1.166e-8	2.057e-9	-5.67	<.0001*
Total Bytes	8.4965e-8	1.251e-8	6.79	<.0001*
(Total Bytes-7.5e+9)*(Total flops-4.3e+10)	-2.17e-21	4e-21	-0.54	0.5959

A11. POSIX-threading on Intel i7

Response Runtime

Summary of Fit

RSquare	0.999998
RSquare Adj	0.999997
Root Mean Square Error	57.46672
Mean of Response	11078.36
Observations (or Sum Wgts)	20

Analysis of Variance

Source	DF	Sum of Squares	Mean Square	F Ratio
Model	4	2.1595e+10	5.3989e+9	1634819
Error	15	49536.3589	3302.4239	Prob > F
C. Total	19	2.1596e+10		<.0001*

Parameter Estimates

Term	Estimate	Std Error	t Ratio	Prob>|t|
Intercept	111.01574	31.05858	3.57	0.0028*
Total flops	3.1293e-7	6.736e-9	46.46	<.0001*
Total Bytes	-3.267e-7	4.451e-8	-7.34	<.0001*
(Total Bytes-7.5e+9)*(Total Bytes-7.5e+9)	-3.27e-17	1.15e-17	-2.85	0.0122*
(Total Bytes-7.5e+9)*(Total flops-4.3e+10)	5.425e-18	1.91e-18	2.85	0.0122*

A12. OpenMP on Intel i7

Response Runtime

Summary of Fit

RSquare	0.999895
RSquare Adj	0.999883
Root Mean Square Error	232.6851
Mean of Response	7251.401
Observations (or Sum Wgts)	20

Analysis of Variance

Source	DF	Sum of Squares	Mean Square	F Ratio
Model	2	8785928185	4.393e+9	81137.27
Error	17	920420.33	54142.372	Prob > F
C. Total	19	8786848605		<.0001*

Parameter Estimates

Term	Estimate	Std Error	t Ratio	Prob>|t|
Intercept	70.341639	62.04045	1.13	0.2726
Total flops	1.3872e-7	1.05e-8	13.21	<.0001*
Total Bytes	1.6819e-7	6.357e-8	2.65	0.0170*

A13. Concurrency Runtime on Intel i7

Response Runtime

Summary of Fit

RSquare	0.99942
RSquare Adj	0.999351
Root Mean Square Error	566.1976
Mean of Response	7699.202
Observations (or Sum Wgts)	20

Analysis of Variance

Source	DF	Sum of Squares	Mean Square	F Ratio
Model	2	9384759107	4.6924e+9	14637.17
Error	17	5449856.1	320579.77	Prob > F
C. Total	19	9390208963		<.0001*

Parameter Estimates

Term	Estimate	Std Error	t Ratio	Prob>|t|
Intercept	214.68954	147.0649	1.46	0.1626
Total flops	1.37e-7	1.515e-8	9.04	<.0001*
Total Bytes	2.1297e-7	9.187e-8	2.32	0.0332*

A14. OpenCL on Intel i7

Response Runtime

Summary of Fit

RSquare	0.999815
RSquare Adj	0.999793
Root Mean Square Error	140.8991
Mean of Response	3385.7
Observations (or Sum Wgts)	20

Analysis of Variance

Source	DF	Sum of Squares	Mean Square	F Ratio
Model	2	1822312771	911156385	45896.14
Error	17	337493.685	19852.57	Prob > F
C. Total	19	1822650264		<.0001*

Parameter Estimates

Term	Estimate	Std Error	t Ratio	Prob>|t|
Intercept	89.308168	37.5677	2.38	0.0294*
Total flops	5.3676e-8	6.358e-9	8.44	<.0001*
Total Bytes	1.3415e-7	3.849e-8	3.49	0.0028*

APPENDIX B

JMP software runtime predictions and error calculations for accelerators

Table B1. SNN model characteristics for verification of accelerator regression models

Observation	SNN Model	Network Size	Total Flops	Total bytes
1	Izhikevich	1680x1680	4.4E+08	8.13E+08
2	Wilson	1680x1680	3.11E+09	3.60E+09
3	Morris-Lecar	1680x1680	5.59E+09	1.19E+09
4	HH	1680x1680	2.79E+11	4.63E+10

Table B2. Predicted runtime for Intel Xeon

Observation	Intercept Term	flop term	byte term	core term	flop*core term	Predicted runtime (ms)
1	3033.8719	1.15E+01	1.00E+02	-3832.5272	727.3944066	40.50462
2	3033.8719	8.14E+01	4.44E+02	-3832.5272	665.8559339	392.7348
3	3033.8719	1.46E+02	1.46E+02	-3832.5272	605.6113447	99.41567
4	3033.8719	7.30E+03	5.71E+03	-3832.5272	-5663.18192	6552.802

Table B3. Verification of results for Intel Xeon

Observation	Actual runtime (ms)	% of Error
1	45.19	10.37
2	406.16	3.30
3	113.31	12.26
4	6632.00	1.19

Table B4. Predicted runtime for AMD Opteron

Observation	Intercept Term	flop term	byte term	core term	flop*core term	Predicted runtime (ms)
1	3181.2476	1.79E+01	1.06E+02	3964.2576	813.626787	154.3034
2	3181.2476	1.27E+02	4.69E+02	3964.2576	728.634583	540.7926
3	3181.2476	2.27E+02	1.54E+02	3964.2576	649.751756	248.3219
4	3181.2476	1.13E+04	6.03E+03	3964.2576	8052.96477	8540.16

Table B5. Verification of results for AMD Opteron

Observation	Actual runtime (ms)	% of Error
1	132.21	-16.72
2	576.60	6.21
3	227.76	-9.03
4	8517.07	-0.27

Table B6. Predicted runtime for IBM PS3

Observation	Intercept Term	flop term	byte term	Predicted runtime (ms)
1	45.708842	1.01E+01	4.22E+01	98.0115
2	45.708842	7.17E+01	1.87E+02	304.0732
3	45.708842	1.29E+02	6.17E+01	236.225
4	45.708842	6.43E+03	2.40E+03	8875.405

Table B7. Verification of results for IBM PS3

Observation	Actual runtime (ms)	% of Error
1	115.01	-14.7726
2	396.7	-23.3493
3	251.20	-5.88645
4	8224.21	7.920778

Table B8. Predicted runtime for Nvidia Fermi

Observation	Intercept Term	flop term	byte term	flop*byte term	Predicted runtime (ms)
1	36.223629	-5.33E+00	7.13E+01	-0.970496517	101.1952119
2	36.223629	-3.77E+01	3.16E+02	-0.530305283	313.8113892
3	36.223629	-6.77E+01	1.04E+02	-0.80557599	71.68442536
4	36.223629	-3.38E+03	4.06E+03	-31.23914143	688.1657636

Table B9. Verification of results for Nvidia Fermi

Observation	Actual runtime for Nvidia Fermi (ms)	% of Error for Nvidia Fermi
1	105.5568	4.310121
2	300.1	-4.36931
3	78.6187	9.673279
4	623.98	-9.32703

Table B10. Predicted runtime for Intel 32 core

Observation	Intercept Term	flop term	byte term	core term	flop*core term	Predicted runtime (ms)
1	31654.395	1.62E+01	4.12E+01	-33781.67	2.15E+03	80.45545
2	31654.395	1.15E+02	1.82E+02	-33781.67	2.08E+03	252.252
3	31654.395	2.06E+02	6.03E+01	-33781.67	2.02E+03	158.7204
4	31654.395	1.03E+04	2.34E+03	-33781.67	-4.93E+03	5584.432

Table B11. Verification of results for Intel 32 core

Observation	Actual runtime (ms)	% of Error
1	98.52	-18.3359
2	230.69	9.346739
3	131.89	20.34305
4	5231.6	6.744253

Table B12. Predicted runtime for AMD 32 core

Observation	Intercept Term	flop term	byte term	core term	flop*core term	Predicted runtime (ms)
1	43595.437	2.00E+01	8.93E+01	-46664.03	2.99E+03	31.76153
2	43595.437	1.42E+02	3.95E+02	-46664.03	2.90E+03	364.892
3	43595.437	2.54E+02	1.31E+02	-46664.03	2.81E+03	125.4121
4	43595.437	1.27E+04	5.08E+03	-46664.03	-6.86E+03	7852.976

Table B13. Verification of results for AMD 32 core

Observation	Actual runtime (ms)	% of Error
1	41.33	-23.1514
2	356.36	2.394203
3	114.94	9.110942
4	6917.79	13.51857

Table B14. Predicted runtime for IBM 16 core

Observation	Intercept Term	flop term	byte term	core term	flop*core term	Predicted runtime (ms)
1	32394.884	1.62E+01	9.03E+01	-34784.94	2.45E+03	168.1952
2	32394.884	1.14E+02	4.00E+02	-34784.94	2.37E+03	498.2801
3	32394.884	2.05E+02	1.32E+02	-34784.94	2.30E+03	249.8729
4	32394.884	1.02E+04	5.14E+03	-34784.94	-5.63E+03	7372.42

Table B15. Verification of results for IBM 16 core

Observation	Actual runtime (ms)	% of Error
1	139.49	20.5787
2	451.19	10.43686
3	269.39	-7.24491
4	6982.59	5.582892

Table B16. Predicted runtime for Sun 32 core

Observation	Intercept Term	flop term	byte term	core term	flop*core term	Predicted runtime (ms)
1	786328.74	7.22E+02	7.67E+02	-841157.7	5.40E+04	708.199
2	786328.74	5.10E+03	3.40E+03	-841157.7	5.10E+04	4682.924
3	786328.74	9.17E+03	1.12E+03	-841157.7	4.82E+04	3658.355
4	786328.74	4.58E+05	4.37E+04	-841157.7	-2.63E+05	183958.6

Table B17. Verification of results for Sun 32 core

Observation	Actual runtime (ms)	% of Error
1	984.93	-28.0965
2	4223.45	10.87912
3	4101.79	-10.8108
4	187331.3	-1.80039

APPENDIX C

JMP software runtime predictions and error calculations for programming models

Table C1. SNN model characteristics for verification of software regression models

Observation	SNN Model	Network Size	Total Flops	Total bytes
1	Izhikevich	1200x1200	2.25E+08	4.15E+08
2	Wilson	1200x1200	1.59E+09	1.17E+09
3	Morris-Lecar	1200x1200	2.85E+09	6.05E+08
4	HH	1200x1200	1.42E+11	2.36E+10

Table C2. Predicted runtime for CUDA (Fermi)

Observation	Intercept	flop term	byte term	flop*byte term	Predicted runtime
1	36.22	-2.72	36.36	-1.03	68.83
2	36.22	-19.22	102.53	-0.89	118.64
3	36.22	-34.53	53.03	-0.94	53.78
4	36.22	-1723.70	2072.26	-5.46	379.32

Table C3. Verification of results for CUDA (Fermi)

Observation	Actual time	% of Error
1	62.24	-10.59
2	99.86	-18.80
3	60.68	11.36
4	365.19	-3.86

Table C4. Predicted runtime for OpenCL (Fermi)

Observation	Intercept	flop term	byte term	flop*byte term	Predicted runtime
1	28.8	2.62E+00	3.53E+01	-6.58E-01	60.77933
2	28.8	1.85E+01	9.94E+01	-5.69E-01	109.1008
3	28.8	3.32E+01	5.14E+01	-6.01E-01	46.37209
4	28.8	1.66E+03	2.01E+03	-3.46E+00	374.7952

Table C5. Verification of results for OpenCL (Fermi)

Observation	Actual time	% of Error
1	69.53	-12.59
2	93.66	16.49
3	55.73	-16.79
4	362.4991	3.39

Table C6. Predicted runtime for POSIX-threading (Intel i7)

Observation	Intercept	flop term	byte term	flop*byte term	byte*byte term	Predicted runtime
1	111.01574	7.04E+01	-1.36E+02	1.64E+03	-1.64E+03	48.50048
2	111.01574	4.98E+02	-3.82E+02	1.42E+03	-1.31E+03	338.1122
3	111.01574	8.92E+02	-1.98E+02	1.50E+03	-1.55E+03	752.447
4	111.01574	4.44E+04	-7.71E+03	8.65E+03	-8.48E+03	37007.7

Table C7. Verification of results for POSIX-threading (Intel i7)

Observation	Actual time	% of Error
1	38.20808	26.94
2	298.916	13.11
3	775.5006	-2.97
4	37147.13	-0.38

Table C8. Predicted runtime for OpenMP (Intel i7)

Observation	Intercept	flop term	byte term	Predicted runtime
1	70.34	3.12E+01	6.98E+01	171.35085
2	70.34	2.21E+02	1.97E+02	487.6871
3	70.34	3.95E+02	1.02E+02	567.44695
4	70.34	1.97E+04	3.97E+03	23737.864

Table C9. Verification of results for OpenMP (Intel i7)

Observation	Actual time	% of Error
1	141.11	-17.6485
2	414	-15.0398
3	629.94	11.01302
4	26000.2	9.530537

Table C10. Predicted runtime for Concurrency Runtime (Intel i7)

Observation	Intercept	flop term	byte term	Predicted runtime
1	214.69	30.78	88.32	333.79
2	214.69	217.40	249.02	681.11
3	214.69	390.61	128.80	734.11
4	214.69	19500.14	5033.18	24748.01

Table C11. Verification of results for Concurrency Runtime (Intel i7)

Observation	Actual time for Concurrency Runtime	% of Error for Concurrency Runtime
1	296.1	12.76497
2	610.43	11.67447
3	792.82	-7.4208
4	24724	-0.11818

Table C12. Predicted runtime for OpenCL (Intel i7)

Observation	Intercept	flop term	byte term	Predicted runtime
1	89.308168	1.21E+01	5.57E+01	157.057518
2	89.308168	8.53E+01	1.57E+02	331.608508
3	89.308168	1.53E+02	8.12E+01	323.445518
4	89.308168	7.62E+03	3.17E+03	10877.24017

Table C13. Verification of results for OpenCL (Intel i7)

Observation	Actual time	% of Error
1	127.522575	23.16056
2	277.090734	19.67506
3	364.681648	-11.3074
4	11029.60253	-1.38139