8-2014

Verifying a Systematic Application to Accelerator Roadmap using Shallow Water Wave Equations

RANAJEET ANAND
Clemson University, rj.anand@gmail.com

VERIFYING A SYSTEMATIC APPLICATION TO ACCELERATOR ROADMAP USING SHALLOW WATER WAVE EQUATIONS

A Thesis
Presented to
the Graduate School of
Clemson University

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

---

by
Ranajeet Pankaj Anand
August 2014

Accepted by:
Dr. Melissa Crawley Smith, Committee Chair
Dr. Walter B. Ligon
Dr. Richard R. Brooks

ABSTRACT

With the advent of parallel computing, a number of hardware architectures have become available for data parallel applications. Every architecture is unique with respect to characteristics such as floating point operations per second, memory bandwidth and synchronization costs. Data parallel applications possess inherent parallelism that needs to be studied and the hardware that can best exploit this parallelism can be identified and selected for large-scale implementation.

The application that I have considered for my thesis is - numerical solution of shallow water wave equations using finite difference method. These equations are a set of partial differential equations that model the propagation of disturbances in water and other incompressible liquids. This application fits in the category of a Synchronous Iterative Algorithm (SIA) and hence, the Synchronous Iterative GPGPU Execution (SIGE) model can be directly applied for performance modeling.

In the high performance computing community, Graphical Processing Units (GPUs) and Field Programmable Gate Arrays (FPGAs) have become highly popular architectures. Homogeneous clusters comprising of multiple processors and heterogeneous clusters that have nodes consisting of both CPU and GPU, are the architectures of interest for this thesis. An initial or high level comparison between the two architectures is performed with regards to the chosen application using a technique known as the Initial Application to Accelerator (A2A) mapping which ranks which architecture delivers the best performance with respect to execution time for large scale implementation.

The subsequent part of the thesis will focus on a low level abstraction of the application of interest to accurately predict the runtime using the multi-level SIGE performance-modeling suite. Through this abstraction, performance modeling of the computation and communication portion of the application is undertaken. The behavior of the computation and communication portions is captured through several instrumented iterations of the application and regression analysis is performed on the execution times. The predicted run time is the sum of the computation and communication run time predictions and is validated by executing the application at higher data sizes.

The thesis concludes with the pros and cons of applying the A2A fitness model and the low level abstraction for run time prediction to the chosen application. A critique of the SIGE model is presented and a Strength, Weakness, Opportunities (SWO) analysis is presented.

DEDICATION

I dedicate this thesis to my family and close relatives and friends for their immense support and encouragement.

ACKNOWLEDGEMENTS

First of all, I would like to express my sincere gratitude to my advisor, Dr. Melissa C. Smith for her invaluable guidance and support throughout this thesis. I am grateful for her advice and her inspiring excellence and knowledge, which have shown me the path to excel.

I would like to thank Dr. Walter B. Ligon and Dr. Richard R. Brooks for being on my thesis committee.

I would like to extend a special thanks to Vivek Pallipuram for providing me valuable guidance and for showing me the right direction in the course of this research.

I am grateful to my family and close relatives who have stood by me like pillars throughout this research. My friends - Nikhil, Nilim, and Kanak, and my sisters - Chaitali and Prajakta have been the moral support for me and have played an important role in my life.

Lastly, I am immensely indebted to Clemson University that has provided me with the tools, access to the Palmetto Cluster, and a platform to showcase my work and helped me enrich my knowledge in the field of computer engineering.

TABLE OF CONTENTS

Page

TITLE PAGE i

ABSTRACT ii

DEDICATION iv

ACKNOWLEDGEMENTS v

LIST OF TABLES viii

LIST OF FIGURES x

CHAPTER

I. INTRODUCTION 1

II. LITERATURE REVIEW 6

Performance Modeling Studies for GPGPU systems 6
SIA Applications on GPGPU systems 11
Summary 14

III. APPLICATION BACKGROUND AND EXPERIMENTAL SETUP 15

Application Description 15
GPGPU Architecture and Memory 20
Palmetto Cluster Configuration 25
Summary 26

IV. VERIFICATION OF THE INITIAL A2A FITNESS MODEL AND SIA MAPPING 28

Overview of the Application to Accelerator (A2A) Fitness Model 28
Application of the A2A Fitness Model 29
SIA to Accelerator Mapping 36
Comparison of Accelerators 41
Summary 42

TABLE OF CONTENTS (continued)

Page

V. LOW LEVEL ABSTRACTION USING THE SIGE MODEL 43

Overview of the Low Level Abstraction of the SIGE Model 43
Application of Low Level Abstraction 44
Summary 56

VI. RESULTS AND ANALYSIS 57

Runtime Predictions from the SIGE Model 57
Insights 69
Effects of Variation in Parameters 75
SWO Analysis of the SIGE Model 78
Summary 81

VII. CONCLUSION 82
Summary 82
Conclusions 85
Future Work 87

REFERENCES 85

LIST OF TABLES

Table Page

4.1 Results of A2A Fitness Model on accelerators of interest 34

4.2 Comparison of Performance to verify A2A mapping 40

6.1 Observed and Predicted Values for Computation Component (ms) 56

6.2 Observed and Predicted Values for Scatter Component (ms) 57

6.3 Observed and Predicted Values for Gather Component (ms) 58

6.4 Observed and Predicted Values for One Time Host to Device Transfer (ms) 59

6.5 Observed and Predicted Values for One Time Device to Host Transfer (ms) 60

6.6 Observed and Predicted Values for Send-Receive Component (ms) 61

6.7 Observed and Predicted Values for Iterative Host to Device Transfer (ms) 62

6.8 Observed and Predicted Values for Iterative Host to Device Transfer (ms) 63

6.9 Observed and Predicted Values for Overall Computation Runtimes (ms) 64

6.10 Observed and Predicted Values for Overall Communication Runtimes (ms) 65

6.11 Observed and Predicted Values for Overall Application Execution Runtimes (ms) 66

6.12 Execution Time of Application over different Y-dimension and K-values with a Ranking of Best Performing Node Configuration and Best G values 68

6.13 Observed and Predicted Best G Value for $Y = 512$, $K = 1000$ 71

6.14 Observed and Predicted Best G Value for $Y = 4096$, $K = 2000$ 71

LIST OF TABLES (continued)

Page

6.15 Variation in Problem Dimension Y on a 8-Node Configuration with $K = 1000$, $G = 20$ 72

6.16 Variation in Number of Nodes P with $Y = 8000$, $K = 1000$, $G = 20$ 73

6.17 Variation in Set Number of Iterations K with $Y = 4000$, $G = 20$ on a 16-Node Configuration 74

6.18 Variation in Number of Ghost Rows G with $Y = 4000$, $K = 800$ on a 2-Node Configuration 74

LIST OF FIGURES

Figure Page

3.1 Grid with vector valued solution at center 18

3.2 Grid with nodes to represent Equation 3.9 and 3.10 18

3.3 Evolution of Water Surface at regular intervals 20

3.4 Grid of CUDA Threads and Blocks 21

3.5 SMX in Kepler GK110 24

3.6 Palmetto Cluster Node 26

4.1 SIA mapping on Heterogeneous CPU-GPGPU cluster 38

4.2 Shared Memory Usage for Equations 3.9, 3.10, 3.11 39

5.1 Kernel Runtime (ms) vs *EDGE-DATA* (bytes) 46

5.2 Kernel Runtime (ms) vs *COMP-DATA* (bytes 46

5.3 Scatter Bandwidth (Megabytes/sec) vs *DATA* (Megabytes) 48

5.4 Gather Bandwidth (Megabytes/sec) vs *DATA* (Megabytes) 49

5.5 One time Host-Device transfer bandwidth (Megabytes/sec) vs *DATA* (Megabytes) 50

5.6 One time Device-Host transfer bandwidth (Megabytes/sec) vs *DATA* (Megabytes) 51

5.7 Send-Receive Bandwidth (Megabytes/sec) vs *DATA* (Megabytes) 52

5.8 Iterative Host-Device transfer bandwidth (Megabytes/sec) vs *DATA* (Megabytes) 53

5.9 Iterative Device-Host transfer bandwidth (Megabytes/sec) vs *DATA* (Megabytes) 54

LIST OF FIGURES (continued)

Figure Page

6.1 Mapping of problem dimensions with number of hardware nodes 69

6.2 Prediction of G value with $Y = 512$, $K = 1000$ for 2, 4, 8 and 16 nodes 70

6.3 Prediction of G value with $Y = 4096$, $K = 2000$ for 2, 4, 8 and 16 nodes 71

CHAPTER 1

INTRODUCTION

In recent times, parallel computing has become the preferred way for application development in the scientific community. Early microprocessors based on single core central processing units (CPU) made rapid advances in terms of fixed point and floating point operations per second and operating frequencies. Such CPUs with complex control logic for branch predictions and hazard prevention were highly conducive for serial application development. Based on Moore's Law, for many years the CPU speeds were projected to increase and sequential software was predicted to perform better as the hardware improved, thus, preempting the need for a change in the software development paradigm.

This trend, however, changed in 2003 when the power wall was hit. As CPU frequencies rose, the energy consumption and heat dissipation in the processors reached extremely high levels and this limited the maximum operating frequencies of processors. This led to the evolution of many core and multicore architectures. However, traditional sequential software is executed primarily on single core CPUs and is incapable of harnessing the power of multicore processors. This called for a change in the software development paradigm and parallel software development became necessary.

Several multicore and many core hardware architectures have evolved for parallel computing, the prominent ones being multicore processors, Graphical Processing Units (GPUs) and Field Programmable Gate Arrays (FPGAs). Intel introduced the Xeon E5-2600 series of processors [1] having up to 16 cores. GPUs used traditionally for graphics

software are highly conducive for parallel applications since GPU devices support thousands of hardware threads useful for processing massively data parallel applications. GPUs used this way are generally termed as General Purpose Graphical Processing Units (GPGPUs). FPGAs are in demand where reconfigurable hardware is necessary. Multiple computing nodes or processors can be coupled via fast interconnect networks to form cluster systems that are highly conducive to parallel computing. Homogeneous clusters that have nodes with many core processors and heterogeneous clusters having nodes equipped with several multicore processors and one or more GPGPU devices are popular hardware accelerators.

These architectures have unique processing capabilities such as floating point operations per second, memory access times, and inter host-device data bandwidth. These characteristics are important considerations when selecting a particular architecture for the application of interest. Characteristics of data parallel software such as the number of data parallel computations, memory access, and data transfers between different architectures become critical. These characteristics are important performance factors for the application because the speedup and execution time is dependent on the ability of the application to exploit the maximum degree of parallelism from the architecture. An architecture selected without such a study may not be fully utilized by the application and therefore, deliver sub-optimal performance. Similarly, the chosen algorithm must expose enough inherent data parallelism to occupy the hardware. Applications that are inherently serial may perform poorly if implemented on parallel architectures. The identification of near optimal hardware architectures and mapping of application to accelerators becomes a non-trivial process. Therefore, there is a need for a roadmap to guide application

developers in identification and ranking of architectures that would be best suited for a particular application. The Application to Accelerator (A2A) roadmap given in [23] has been studied and applied in this thesis.

In my thesis, the application of interest is the numerical solution of shallow water wave equations using finite difference method [25]. In this application, the computation of the shallow water surface takes place over a square grid on which each point of the surface is computed using finite difference over two stages. This is a highly computationally intensive and data parallel step that takes place over multiple iterations, reflecting the water surface as it evolves over time. The application can be classified as a Synchronous Iterative Algorithm (SIA) since several processing units compute the entire water surface over multiple iterations. Therefore, the Synchronous Iterative GPGPU Execution (SIGE) model [16] has been used for performance modeling and runtime prediction.

One significant focus of the thesis is the systematic verification of the A2A roadmap. A homogenous multiprocessor cluster and a heterogeneous CPU-GPGPU cluster are the architectures studied and the A2A roadmap is used to identify the accelerator that can deliver the optimal performance. Strengths, Weaknesses and Opportunities (SWO) of the A2A roadmap with respect to this application are discussed. The accelerator identified using this roadmap is selected and low-level abstraction of the application as per the SIGE model is carried out.

The verification of the performance prediction framework consisting of he low-level abstraction described in the SIGE model is the second significant focus of the thesis. In this analysis, performance modeling of the computation and communication

parts of the application is carried out. Through several instrumented executions of the application, runtimes at smaller data sizes are captured. Using this data, regression analysis is carried out to accurately predict the runtimes at larger data sizes. For the regression analysis, parameters such as data bytes transferred, compute data size, number of floating point operations are typically used as independent predictor variables with the runtime as the dependent variable. The predictor variables are selected based on regression parameters such as high R^2 (greater than 0.90) and low p-values (less than 0.10). The overall predicted runtime is the sum of the computation and communication runtime predictions and this is verified by executing the application at higher data sizes. The experimental results were observed to be within 10 percent of the predicted runtimes. The ease of use of the SIGE model is also discussed.

An important comment should be made regarding the application choice. Since the finite difference method is used, at a given instant of the algorithm, data at a location in the water surface depends on the values of its adjacent points. For points along the grid borders of a particular process, the adjacent points may reside along the grid borders of a neighboring process and vice versa. Such points are typically called “ghost data” and must be exchanged between neighboring processes during each iteration. On a heterogeneous CPU-GPU cluster, this leads to inter CPU host – GPU device ghost data transfers. The combined effect over multiple iterations is that there are a large number of inter process and inter host-device communications, making the application highly communication sensitive; this is a good contrast from the four Spiking Neural Network (SNN) models – Hodgkin-Huxley, Izhikevich, Morris-Lecar, and Wilson and the Anisotropic Digital Filter (ADF) algorithms used in [31] and further studied in [32] that

are mostly pleasingly parallel with lesser communication complexity and higher computation sensitivity. The chosen application qualifies to have sufficient computation and communication complexity. Verification of the SIGE model on such an application marks a significant contribution of this thesis.

The cluster systems used in this research belong to the Palmetto Cluster at Clemson University [30]. The cluster configuration consists of up to 16 nodes Intel Xeon ES-2665 HP SL250s nodes with 16 cores, 64GB memory and inter-connected over Infiniband. Each node is coupled with NVIDIA Tesla K20 GPU cards. The configuration of the Palmetto Cluster is described in further chapters.

The remainder of the thesis is organized as follows. Chapter 2 describes the literature study pertaining to performance modeling studies and development of applications on GPGPUs, similar to the application of interest. Chapter 3 describes the application in depth and elucidates the available accelerators. The chapter also describes the Compute Unified Device Architecture (CUDA) and discusses the NVIDIA Tesla K20 GPU architecture. Chapter 4 conducts the A2A fitness study and application mapping on the chosen accelerator is discussed. The chapter concludes with a runtime analysis of both accelerators and sheds light on the A2A verification. Chapter 5 details the low-level abstraction using the SIGE model. Chapter 6 consists of the results and analysis, and a verification of the SIGE model. The chapter also consists of certain insights concerning to execution of SIAs on parallel accelerators and are developed by using the prediction framework. The thesis concludes in Chapter 7 with a summary, conclusions, and Strengths, Weaknesses and Opportunities (SWO) analysis of the SIGE model and future challenges that will consolidate the application to accelerator mapping.

CHAPTER 2

LITERATURE REVIEW

In this chapter, we bring to light the recent developments with regards to performance prediction and architecture selection for different data parallel applications. Section 2.1 discusses the advances made in the field of performance modeling and section 2.2 explores various SIA applications that have been implemented on GPGPUs and that typically use finite difference methods. The chapter closes with a summary in section 2.3

2.1 Performance Modeling Studies for GPGPU systems.

In this section, we explore the performance modeling studies that have been conducted in the realm of GPGPU.

In [6], the authors have designed a Scalable HeterOgeneous Computing benchmark (SHOC) suite that focuses on the performance and stability of scalable heterogeneous computing systems such as GPUs and multicore processors. Their work consists of benchmarks that test the performance of low level hardware characteristics such as device memory, bus speed download and readback, kernel compilation, queueing delay, and resource contention using a set of parallel applications. Heterogeneous architecture comprising of devices such as the NVIDIA 8800 GTX, NVIDIA Tesla C1060, ATI Radeon HD5770 and multicore CPUs like Intel Gainestown and Harpertown are considered. The authors discuss performance of CUDA and OpenCL and contention for system resources observable during inter-device and inter-node communications. Although the SHOC suite provides results with a high degree of accuracy, it is restricted

to measurement of hardware parameters and cannot be used for overall performance prediction of the application.

In [7], a performance analysis framework that identifies root cause of performance bottlenecks and an estimation of the degree of benefit of applying optimization strategies using static and dynamic profiling and a suite of micro-benchmarks is discussed. The prediction framework uses inter-thread instruction-level parallelism, memory-level parallelism, computing efficiency and serialization effects to estimate the performance benefits. The authors use the NVIDIA Fermi C2050 and the performance model builds upon an existing MWP-CWP model by using parameters like cache effect and SFU instructions. The speedup resulting from optimizations such as use of shared memory, loop unrolling, data layout, eliminating divergent branches and reduction of idle threads are discussed and are within 10% of the predicted results. However, the framework makes use of extremely intricate hardware and software parameters that may not be easily available therefore, making the model highly complex to use.

In [8], the authors present the Multi2C simulation framework through which different heterogeneous devices can be evaluated based on different performance or reliability criteria. The authors build upon the existing Multi2Sim framework to translate OpenCL and CUDA kernels to an LLVM representation. The compilation infrastructure models hardware and software timings based on instruction queues, divergent branches, and functional units. The framework provides for memory coherency modeling by allowing for dynamic cache block transitions between coherent and non-coherent modes and achieve up to 1.8x speedup on the AMD Radeon 5870 using OpenCL. The

framework also includes fault injection into the execution to test the architectural vulnerability. However, the parameters considered in this study are also difficult to access to rendering the model difficult for performance modeling.

In [9], the authors identify a set of important GPU application characteristics and use those to predict performance of an arbitrary application by determining its most similar proxy benchmarks using a range of prediction suites such as the Rodinia suite, GPGPU-SIM and NVIDIA-SDK suite. Speedup for a particular benchmark is obtained by taking a weighted sum of the speedups of the proxy benchmarks. The application metrics considered include but are not limited to, instruction throughput, computation-to-memory access ratio, memory efficiency and warp occupancy. The authors base their experiments on the NVIDIA Tesla C205 and the Kepler K20 GPUs. For applications that match the benchmarked applications, the prediction results have an accuracy of 13% to 15% but the error increases for outlier applications. The framework makes use of existing benchmarks, but cannot be used for a novel application for which benchmarks may not exist along with the fact that the prediction errors are high.

In [10], authors propose the Eiger modeling framework for automating the generation of performance prediction models by profiling workloads using micro benchmarks and regression techniques. The framework constructs performance models and evaluates performance sensitivity to processor configurations using Principal Components Analysis (PCA). The application metrics are independent of the device on which it is running and machine metrics describe the hardware. A wide range of application metrics like memory efficiency, SIMD execution, static and dynamic memory and machine metrics like bandwidth and streaming multiprocessors are considered.

Although the framework predicts the performance faithfully, the application and machine parameters in the PCA are not readily available.

In [11], the authors extend an existing GROPHECY framework to project the overall GPU speedup from abstract CPU code and project the overhead of data transfer between CPU and GPU using a data usage analyzer and a PCIe bus model. The framework estimates the performance gained by GPU acceleration by modeling the data transfer overhead. The authors employ an NVIDIA Quadro FX 5600 GPU and use the CFD, HotSpot, SRAD and Stassuij applications as benchmarks. They achieve a prediction error of 8% on the data transfer overhead and 9% on the overall GPU speedup, but only at significantly high number of iterations. At lower iterations, the data transfer overhead is significant and the framework detects higher error rates, thus exposing a drawback of the framework. Moreover, the framework is restricted to modeling data transfers for pinned CPU memory rather than the more common pageable CPU memory.

In [12], the authors extend the PMaC performance-modeling framework for prediction of large-scale HPC applications by profiling application and machine characteristics. The model identifies compute and memory access patterns for scatter/gather, stream, reduction, etc operations on different hardware and projects the obtainable speedup by optimizing the same. The architectures used are NVIDIA Fermi C2070 GPU and Convey FPGA co-processor and the authors could achieve an average accuracy for bandwidth prediction within 3.16% and 2% for the FPGA and GPUs respectively. Although fairly accurate, the model requires the knowledge of memory access patterns and projects speedup of individual patterns instead of the application as a whole, which can be a concern if the application is iterative in nature. Further, the model

does not illustrate the effect of optimizations on multiple patterns and how the overall performance would be improved.

In [13], the authors present the Boat Hull model in which performance is predicted for GPU and multi-core architectures by creating instances of the roofline model for different algorithm classes. The model doesn't require code but uses off-chip memory accesses and coalesced and uncoalesced accesses, data size and number of threads to predict the computes and data transfer times. The NVIDIA GeForce GTX 470 GPU was used and the performance prediction is within 8% of the measured performance. However, the selected SIGE model has better accuracy since better modeling could be accomplished through code study.

In [14], the Bulk Synchronous Parallel (BSP) model is proposed that aims in the mapping and structuring of iterative parallel applications on heterogeneous architectures. But the model is highly theoretical and provides the performance at near optimal processor utilization and cannot be directly applied for performance prediction. The Heterogeneous BSP model [15] increases the applicability of BSP by incorporating parameters that reflect the relative speeds of heterogeneous computing components. However, these models aim to guide the design of applications for optimal performance on a given machine.

In [16], the Synchronous Iterative GPGPU Execution (SIGE) model for the performance prediction of Synchronous Iterative Algorithms (SIA) is presented. This model uses a regression-based approach to predict the computation and communication sections of the application by collecting micro benchmarks. The model makes use of predictor variables like the number of floating point operations, total bytes consumed,

data transfer size and processor count. The authors considered four spiking neural network SIAs and with the NVIDIA Tesla M2070 GPU, they could achieve performance prediction accuracy of over 90%.

2.2 SIA Applications on GPGPU systems

In this chapter we discuss the evolution regarding the implementation of applications that require finite difference methods on GPGPUs. Each study comprises of a brief description of the application, optimization strategies, experimental setup, and results obtained.

In [17], the authors use a 2D problem for computation of electric field values caused by the light scattering due to a transverse magnetic wave and implement it on a single GPGPU. In this inherently data parallel application, a finite difference time domain method is used, where the value of each cell depends on the previous two time steps and the values of its directly neighboring cells. This is an SIA with a high number of iterations (~100000) and ghost rows used for computing edge elements are exchanged with the host CPU in every iteration. The authors make use of shared memory and global memory is accessed in a coalesced manner. A PC with AMD Athlon 4000+ with a 2.4GHz CPU and 2GB RAM and one NVIDIA GeForce 8800 GTX GPGPU was used to obtain the benchmarks. For large input data sets (4 Million data elements or 128x128 grid) the authors observed a speedup of up to 50x.

In [18], the authors perform seismic modeling and reverse time migration (RTM) using a finite difference method on a 2D and 3D mesh. Seismic waves that are reflected and/or refracted at the interface of geological interfaces are used as boundary conditions. Asynchronous MPI communications are used to exchange ghost rows and are performed

at every iteration in this SIA. Shared memory is used and optimizations are performed to increase occupancy. Scalability and speedup comparison for constant, variable density and RTM is discussed. The authors used a GPGPU cluster testbed composed of 10 Xeon bi-socket quad-core nodes coupled with 5 NVIDIA TESLA S1070 servers. The TESLA server is composed of 4 T10 GPGPUs and speedups up to 10x for RTM and up to 30x for seismic modeling were observed.

In [19], the authors implement versions of scattering of acoustic waves in non-homogeneous media on GPGPUs using shared memory and texture memory approaches. The application is discretized into a set of finite difference equations by replacing partial derivatives with central differences. The authors concluded that the shared memory approach performed better since the computation time was significantly lower and the CUDA occupancy was higher. For the texture memory approach, data is copied between device global memory and texture memory and saved into texture memory in every iteration, leading to slowing down of the kernel. The shared memory approach used 2 kernels - one to load data into the shared memory, and other to compute values of the next time step. The authors conducted the experiments on a Tesla C1060 GPGPU composed of 30 multiprocessors; 4GB DDR3 memory 16KB shared memory per block and 2D Texture memory with 216 width × 215 height.

In [20], the authors accelerate a 3D finite difference wave propagation application on a single GPGPU and heterogeneous CPU-GPU cluster using CUDA-MPI. The authors use a 2D mesh along with a sliding computation window to account for the lack of sufficient memory for a 3D grid. Shared memory and register optimization techniques are used to hold data of the 2D grid and ghost elements. There is an effective overlap of

computations and communications by exchanging ghost elements with neighboring processes using non-blocking MPI communications and computing inner points on the GPGPUs. The experimental setup consists of a cluster of 48 NVIDIA Tesla S1070, each having four GT200 GPGPUs and two PCIe-2 buses and connected to BULL Novascale R422 E1 nodes. The authors could achieve a speedup of 37x for a single GPU over the serial version. The authors further conclude that the application has weak scalability but not strong scalability because of the stalls for non-blocking MPI communications. With different configurations of the application, the speedup was between 20x and 60x for the CUDA-MPI version.

In [21], the authors perform simulation of room acoustics with a finite-difference time-domain model in real-time, up to a geometry of $100m^3$. With a 10% maximum dispersion error limit, the system could be used for real-time auralization up to 1.5kHz. The authors choose a low sampling rate of 7kHz since at higher frequencies the computational load can be excessive. 3D GPGPU grids are used to model the finite difference equations and ghost elements are present to compute boundary elements. The CPU was used to perform the required sampling rate conversions. Two GPU kernels are executed: one, to update the mesh points and the other, for the boundary filters. Computation of 1 time step requires information from the previous two time steps and GPU L1 and L2 caches are used. Issues such as memory coalescing and occupancy are addressed to obtain the maximum performance. The data for the impedance filters are pre-computed and stored in constant memory. The experimental setup consisted of an NVIDIA Quadro FX 5800 with 4GB of global DRAM and a commodity PC having Intel Pentium Dual CPU E2180 running at 2GHz and 2GB of RAM. The authors finally

perform a comparison of different schemes of the application with regards to computation size and performance.

In [22], the authors assess the performance improvement of a GPGPU-based implementation of elliptical or steady heat conduction. A five-point finite difference scheme using a Point Over Successive Relaxation (PSOR) method, which uses computation over two schemes in an iterative manner. The authors perform padding of memory so that the global memory reads and writes are aligned. Only global memory is used. Further, to ensure synchronization between threads, each computation kernel is launched iteratively. The authors use an NVIDIA GTX260 GPGPU and analyze the performance of the application on coarse, medium, and fine grids with the fine grid performing the best and having the best occupancy out of the three. The use of padded global memory led to 26% faster execution for the GPU kernels. The authors further concluded that the speedup reached a constant value at higher number of iterations.

2.3 Summary

In this chapter, we present the performance modeling studies for GPGPU systems as and applications that use finite difference methods on GPGPUs. We also discussed in brief, the SIGE model that targets SIAs for performance prediction. The models presented in section 2.1 are sufficiently accurate but consist of complex procedures to model the performance and require a detailed knowledge of the GPU architecture. The SIGE prediction framework requires easily available application and hardware parameters and makes the modeling task straightforward. We select this model for performance prediction for the SIA of our choice – shallow water wave equations and aim to provide a verification of this model.

CHAPTER 3

APPLICATION BACKGROUND AND EXPERIMENTAL SETUP

In this chapter, we elucidate the application in depth. The computation and communication portion of the application is discussed and data parallelism and communication complexity is exposed. In subsequent sections, the accelerators of interest namely, the homogeneous multiprocessor cluster and heterogeneous CPU- GPGPU systems are discussed. The microarchitecture of NVIDIA GPUs and characteristics of the NVIDIA Tesla K20 GPU are also discussed.

3.1 Application Description

As introduced in previous chapters, the shallow water wave equations are a set of partial differential equations that model the propagation of disturbances in water and other incompressible fluids. The finite difference method is used to find numerical solutions of these partial differential equations. These equations are typically used for incompressible fluids with the underlying assumption being that the depth of the fluid is small compared to the wavelength of the disturbance. Shallow water can store and release energy by locally varying its height within certain limits.

The partial differential equations are derived from Moler's model [25] of shallow water that uses conservation of mass and momentum. The independent variables are time t, and motion in two space coordinates, x and y. The dependent variables are the fluid height h, and the two-dimensional fluid velocity, u and v. Here, u implies the motion in x direction and v in y direction. As state variables, the set of h, $h.u$ and $h.v$ is chosen. With consistent units, the conserved quantities are mass, that is proportional to h, and

momentum that is proportional to *u.h* and *v.h*. The force acting on the fluid is gravity, represented by the gravitational constant *g*. The partial differential equations then take the following form as shown in Equations 3.1, 3.2 and 3.3.

$$\frac{\partial h}{\partial t} + \frac{\partial (uh)}{\partial x} + \frac{\partial (vh)}{\partial y} = 0 \tag{3.1}$$

$$\frac{\partial (uh)}{\partial t} + \frac{\partial (u^2 h + \frac{1}{2} g h^2)}{\partial x} + \frac{\partial (uvh)}{\partial y} = 0 \tag{3.2}$$

$$\frac{\partial (vh)}{\partial t} + \frac{\partial (uvh)}{\partial x} + \frac{\partial (v^2 h + \frac{1}{2} g h^2)}{\partial y} = 0 \tag{3.3}$$

Equations 3.4, 3.5 and 3.6 represent the above equations in a compact form using the three vectors as shown below:

$$U = \begin{pmatrix} h \\ uh \\ vh \end{pmatrix} \tag{3.4}$$

$$F(U) = \begin{pmatrix} uh \\ u^2 h + \frac{1}{2} g h^2 \\ uvh \end{pmatrix} \tag{3.5}$$

$$G(U) = \begin{pmatrix} vh \\ uvh \\ v^2 h + \frac{1}{2} g h^2 \end{pmatrix} \tag{3.6}$$

Here, *F* and *G* are intermediate vectors that are computed at half time steps and assist in the calculation of the position vector *U* at the end of a complete time step. Using the above notation, the shallow water wave equations become an instance of a hyperbolic conservation law as shown in Equation 3.7:

$$\frac{\partial U}{\partial t} + \frac{\partial F(U)}{\partial x} + \frac{\partial G(U)}{\partial y} \tag{3.7}$$

A square region is chosen to represent the shallow water surface or vector *U*. Boundary conditions must be considered to model a real world situation [26] therefore, the reflective boundary conditions, $u = 0$ on the vertical sides of the regions and $v = 0$ on the horizontal sides, are applied. Furthermore, at the left and right vertical edges, the condition $u = -u$ and at the top and bottom horizontal edges, $v = -v$ are applied. With these boundary conditions, any wave that reaches the boundary is reflected back into the region.

The Lax-Wendroff method is used to compute a numerical approximation to the solution. For this, a regular square finite difference grid with a vector-valued solution centered in the grid is introduced as shown in Figure 3.1. The quantity vector:

$$U_{i,j}^{n} \tag{3.8}$$

represents a three component vector at each grid cell i,j that evolves with time step n.

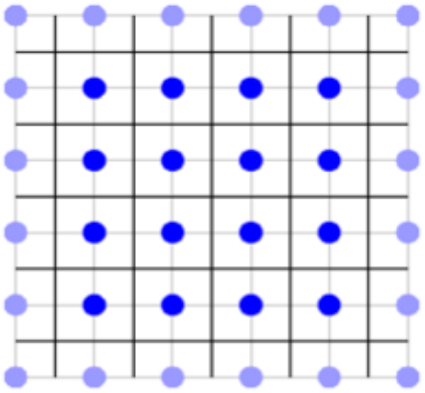


Figure3.1: Grid with vector valued solution at the center. Light blue points handle boundary conditions

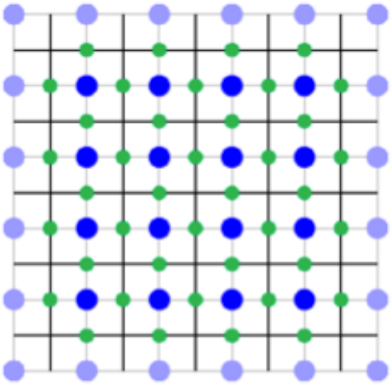


Figure 3.2: Green nodes represent calculations resulting from Equations 3.9 and 3.10

In the Lax-Wendroff method, a time step is covered over two stages. In the first stage, known as a half step, values of *U* at time step *n*+1/2 and the midpoints of the edges of the grid are defined. Equations 3.9 and 3.10 describe the first stage.

$$U_{i+\frac{1}{2},j}^{n+\frac{1}{2}} = \frac{1}{2}(U_{i+1,j}^{n} + U_{i,j}^{n}) - \frac{\Delta t}{2\Delta x}(F_{i+1,j}^{n} - F_{i,j}^{n}) \tag{3.9}$$

$$U_{i,j+\frac{1}{2}}^{n+\frac{1}{2}} = \frac{1}{2}(U_{i,j+1}^{n} + U_{i,j}^{n}) - \frac{\Delta t}{2\Delta y}(G_{i,j+1}^{n} - G_{i,j}^{n}) \tag{3.10}$$

Figure 3.2 shows how the above equations are computed. In the second stage, the time step is completed by using the values from the first stage to compute the new values at the centers of the cells as shown in Equation 3.11.

$$U_{i,j}^{n+1} = U_{i,j}^{n} - \frac{\Delta t}{\Delta x}\left(F_{i+\frac{1}{2},j}^{n+\frac{1}{2}} - F_{i-\frac{1}{2},j}^{n+\frac{1}{2}}\right) - \frac{\Delta t}{\Delta y}\left(G_{i,j+\frac{1}{2}}^{n+\frac{1}{2}} - G_{i,j-\frac{1}{2}}^{n+\frac{1}{2}}\right) \tag{3.11}$$

The significant pieces of the overall algorithm are the computations performed in the first and second time steps, that is, Equations 3.9, 3.10, and 3.11. These represent the compute intensive and data parallel nature of the application. The three equations are computed iteratively and each iteration represents a smoothening of the water surface. This application fits the category of a Synchronous Iterative Algorithm (SIA) [16] since several iterations of the compute intensive parts of the application are required.

The initial conditions are chosen as $h = 1$, $u = 0$, $v = 0$ over the entire region. This calm water surface is disturbed by a water droplet hitting the surface, which is represented by adding a two dimensional Gaussian shaped peak to h. After this impulsive disturbance, the resulting waves propagate back and forth over the region. The initial conditions only affect the nature of the water surface and not the computations that use the finite difference method steps. Therefore, a sharper Gaussian peak would result in greater number of iterations being required for the water surface to eventually smooth out. However, the performance modeling carried out in this thesis does not depend on the number of iterations that are used for the application and hence, is independent of the initial conditions. The performance is modeled for a single iteration and is scaled with the number of iterations to calculate the total runtime.

During the execution of the iterative algorithm, at regular intervals, the output representing the water surface is collected in a file. This file is used to plot snapshots of the wave behavior at regular intervals of time with a MATLAB program. In Figure 3.3, some of the sample outputs are shown with the propagation of disturbances visible from Figures 3.3a to 3.3d.

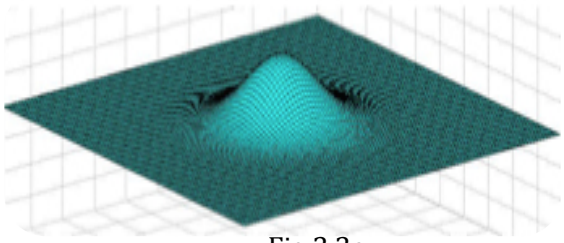


Fig 3.3a

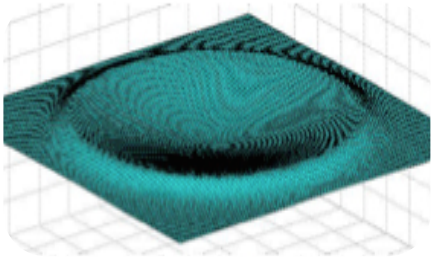


Fig 3.3b

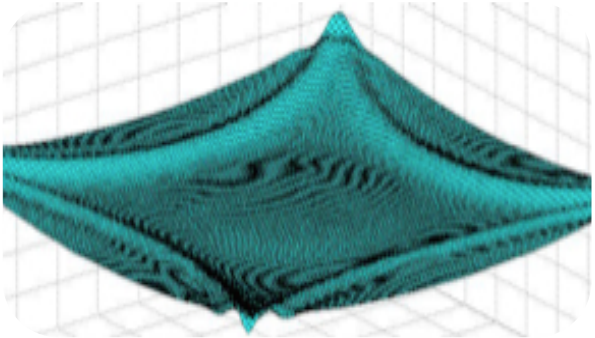


Fig 3.3c

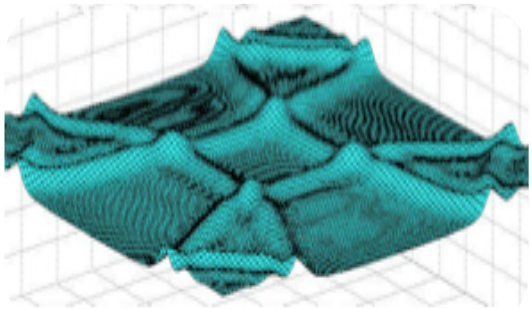


Fig 3.3d

Figure 3.3: Evolution of the water surface at regular intervals of the application

3.2 GPGPU Architecture and Memory

In this thesis, the NVIDIA Tesla K20 GPU is considered. In this section, we describe the NVIDIA CUDA framework and GPGPU architecture and specific characteristics of the K20 GPU.

3.2.1 NVIDIA CUDA framework

The GPGPU technology is based on Compute Unified Device Architecture (CUDA) [2] that was introduced by NVIDIA in 2007. The CUDA architecture consists of thousands of floating-point processing units or CUDA cores and memories such as global, constant and texture memory. These GPU cores consist of processing units called threads and execute the parallel sections of the program called CUDA kernels. CUDA

kernel execution takes place in a Single Program Multiple Data (SPMD) manner since all of the threads execute the same code.

The GPU threads are organized into a two level hierarchy of blocks and grids. The threads are arranged in a 1D, 2D or 3D structure called blocks. The *threadIdx* variable specifies the number of the thread with respect to the block. The number of threads in each dimension in a block is specified with the *blockDim* variable. A CUDA block can have a maximum of 1024 threads. The blocks are further arranged in a 3D manner to form a grid, where each block has a unique index *blockidx*. Together with the *blockDim*, *blockIdx* and *threadIdx* variables, the global thread coordinate can be uniquely determined. An execution configuration containing information about the number of blocks and the number of threads per block, is specified when a CUDA kernel is launched. Figure 3.4 shows the arrangement of blocks and threads.

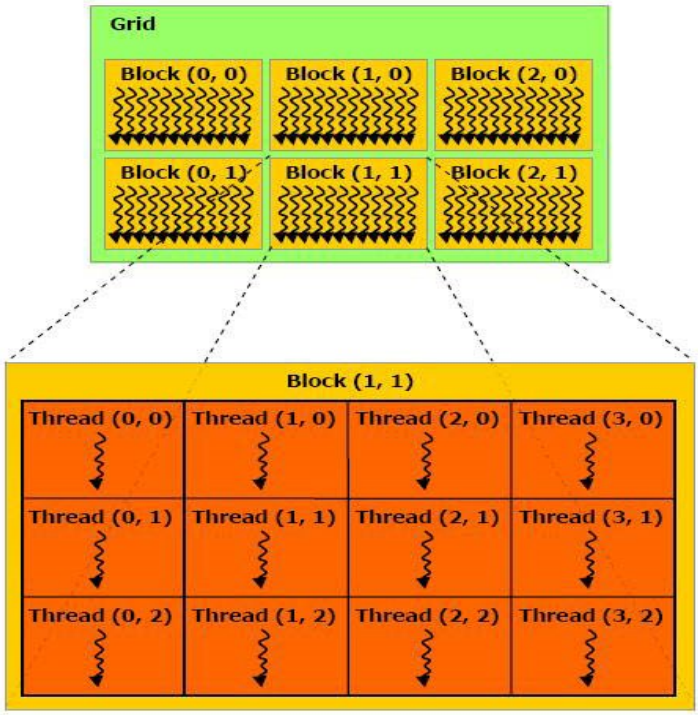


Figure 3.4: Grid of CUDA threads and blocks

CUDA runtime organizes the execution resources into Streaming Multiprocessors (SM). During the kernel execution, threads are assigned to SMs on a block-by-block basis. A limited number of blocks can be assigned to each SM and the CUDA runtime assigns new blocks to SMs as they complete previous block executions. Each SM consists of multiple Streaming Processors (SPs) that share control logic and instruction cache. Thread blocks in SMs are divided into 32 consecutive threads called warps. Threads in a block are executed by groups of 16, called as half-warps and are executed in a Single Instruction Multiple Data (SIMD) manner. Each SM can execute a limited number of warps at any instant. Accesses to global memory can be aligned or coalesced by the GPU hardware into a single efficient transaction per half-warp, thereby increasing the memory performance.

GPU threads access data from different GPU memories. Global memory is off-chip memory implemented with DRAM technology that is accessible to all threads in the device. Although it supports L1 and L2 caches, it has the highest access latencies and the lowest bandwidth. Registers are on-chip memories and have negligible access times. These are accessible to individual threads and are used to hold automatic variables. However, each block has a limited number of registers. Shared memory is slower than registers but is still accessible to all threads in a block. It is commonly used for thread collaboration and synchronization. The amount of shared memory per block is also limited. Local memory is used to hold automatic array variables. It has the same access latency as global memory since it resides in global memory. The scope of local memory is also per thread. Constant memory is used to provide read-only values to the kernel. It is

stored in global memory but is cached for faster access. The scope of this memory is for all threads in the GPU device.

3.2.2 NVIDIA Tesla K20 GPU (Kepler GK110)

The NVIDA Fermi microarchitecture [3] was a significant leap forward since the G80 architecture. The Fermi architecture has salient features such as 512 CUDA cores with 32 cores per SM, 64 KB of memory configurable for use as shared memory and L1 cache and a total of 6 GB of GDDR5 DRAM. It supports Error Correction Code (ECC) and has dual warp scheduler that simultaneously schedules and dispatches instructions from two independent warps. The cores are organized in 16 SMs and each core has a pipelined integer arithmetic logic unit (ALU) and floating point unit (FPU). Each SM has 16 load/store units. It is capable of performing fused multiply-add (FMA) instruction for single and double precision arithmetic.

NVIDIA introduced Kepler GK110 microarchitecture [5] – a huge improvement over the Fermi architecture and focuses on compute performance and reduction in power dissipation. The Tesla K20 and K20X GPUs are derivatives of this architecture. The Kepler GK110 supports CUDA compute capability 3.5. Each of the Kepler GK110 Streaming Multiprocessor (SMX) units have 192 single-precision CUDA cores and retain the single and double-precision arithmetic introduced in Fermi. The Kepler family can support up to 16 SMX per block. With 13 SMX per block, the K20 GPU supports 2496 CUDA cores [4]. It supports a memory width of 320-bits. The global memory is up to 5 GB. The Kepler GK110's SMX provide up to 8x the number of SFUs of the Fermi GF110 SM. Like the Fermi, Kepler has a warp size of 32 threads and supports up to 64

warps per SMX. Each SMX of the Kepler has a quad warp scheduler each with dual instruction dispatch units, allowing four warps to be issued and executed concurrently. Each thread can access up to 255 registers for the Kepler GK110. Shuffle instruction is introduced that allows threads within a warp to share data.

Figure 3.5: SMX in Kepler GK110 [5]

Kepler's memory hierarchy is similar to Fermi's. Each SMX has 64 KB of on chip memory that is configurable as 48 KB of shared memory with 16 KB of L1 cache or vice versa. It is possible to configure a 32KB / 32KB split between the allocation of shared memory and L1 cache. Shared memory bandwidth for 64b and larger load operations is doubled to 256B per core clock. In addition to L1 cache, a 48 KB read-only cache is available. The dedicated L2 cache up to 1536 KB is available and supports up to 2x the bandwidth compared to the Fermi. ECC for memory protection is present. Figure 3.5 shows the Kepler GK110 architecture.

The Kepler GK110 has further salient features such as Dynamic Parallelism, Hyper Q, and NVIDIA GPUDirect. Using dynamic parallelism, more parallel code in an application can be directly launched by the GPU onto itself, thus, performing load balancing. Using Hyper Q, up to 32 simultaneous hardware work queues between the host and the CUDA work distributer logic to overcome effects of serialization. GPUDirect aims to reduce compute latencies through DMA between NIC and GPU and better MPI communications between GPU and nodes in a network.

3.3 Palmetto Cluster Configuration.

For the purposes of this thesis, we have used the Clemson University Palmetto Cluster computing system [30]. The cluster provides a homogeneous CPU-only configuration as well as a heterogeneous CPU-GPGPU configuration. Each node used in the cluster node is of a HP SL250s make including up to 16 cores of Intel Xeon E5-2665 processors and up to 64GB of RAM memory. We use the Message Passing Interface (MPI) standard [27] for application development on the homogeneous cluster and CUDA-MPI for the heterogeneous cluster. The processors are capable of performing

double-precision floating-point operations, even though only single precision operations are considered. Each node is further equipped with 2 NVIDIA Tesla K20 devices. The communication between the CPU and GPU device takes place over the PCI-Ex bus. The inter-node interconnect network is 56g Infiniband. The GPU devices are present such that 2 CPU cores in every node are connected to 2 GPU devices in a 1:1 node packing fashion. Therefore, only 2 out of 16 CPU cores are used in each node by the application. Figure 3.6 shows an instance of the cluster node with 2 cores being used in a node.

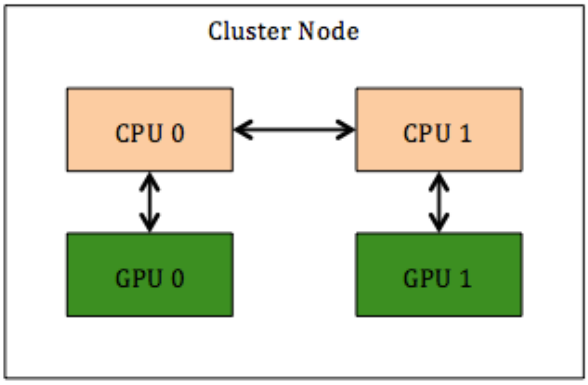


Figure 3.6: Palmetto Cluster node.

Ultimately, each node supports 2 CPU cores and 2 GPU devices. At the time of selecting a configuration, we reserve all 16 cores in each node (although only 2 CPU cores are actually used) to ensure that the traffic across that node belongs entirely to our application and minimize communication interference caused by other applications. We use the heterogeneous CPU-GPU configuration for both, the MPI-only and the CUDA-MPI versions to maintain consistency across cluster configurations.

3.4 Summary

In this chapter, we described the application of interest. The CUDA framework used for GPGPUs and the architecture of the NVIDIA Tesla K20 GPU used in this thesis

was also described. We followed this up with an overview of the Palmetto cluster and the configurations used for executing the application.

CHAPTER 4

VERIFICATION OF THE INITIAL A2A FITNESS MODEL AND SIA MAPPING

We begin this chapter, by applying an initial A2A Fitness model [23] to the architectures of interest – a homogeneous multiprocessor cluster and a heterogeneous CPU-GPGPU cluster. In the subsequent sections, we first discuss the initial Application to Accelerator (A2A) fitness model and perform a careful mapping of the application components on the cluster systems. The chapter concludes with an experimental validation of the fitness model – the performance of the application on both clusters is compared and the results of the fitness model are verified.

4.1 Overview of the Application to Accelerator (A2A) Fitness model

In this section, we provide an overview of the Application to Accelerator (A2A) [23] fitness model and an in depth discussion for use with our application. Through the use of this model, we aim to establish a preliminary runtime prediction of the application on each accelerator by calculating a scalar product of two vectors – the application vector and the accelerator vector. In this fitness model, significant computation or communication transactions are identified and considered as directions or unit vectors. If i, j, k … are identified as the unit vectors, then with $a1$, $a2$, $a3$… as the application vector coefficients and $b1$, $b2$, $b3$ … as the accelerator vector coefficients, we have the application and accelerator vectors as shown in Equations 4.1 and 4.2:

$$\text{Application vector} = a1\ i + a2\ j + a3\ k + \ldots \tag{4.1}$$

$$\text{Accelerator vector} = b1\ i + b2\ j + b3\ k + \ldots \tag{4.2}$$

The application and accelerator vectors have units of *parameter* and *seconds/parameter* where parameter can be bytes transferred or floating-point operations (FLOPs). The scalar product of these vectors has the unit of seconds and conveys the execution time of the application on that device. The scalar product is given by Equation 4.3.

$$\text{Scalar Product} = a1.b1 + a2.b2 + a3.b3 + \ldots \quad (4.3)$$

We can rank the performance of the systems based on their scalar product. The accelerator with the smallest execution time, i.e. scalar product, is deemed the best fit for that application. More information about the A2A fitness model can be found in [24].

4.2 Application of the A2A Fitness Model

We apply the A2A fitness model by identifying the application and accelerator vector components. Each accelerator has a unique accelerator vector that consists of the FLOPs per second (FLOPS), the data transfer time over Infiniband or PCI-Ex bus, and the per byte data access time by processing cores. Each accelerator has a corresponding unique application vector that consists of the FLOPs count, the bytes of data transferred, and the data bytes consumed by processing cores. As discussed above, the application vectors have units of *parameter* and accelerator vectors are in *seconds/parameter*.

Each component of the vector consists of a direction vector or a unit vector corresponding to either a communication component or a computation component across both the architectures. We identified the following 9 unit vector components.

i Processor or CPU FLOPs

j Data transferred in scatter operation

k Data transfer from CPU host to GPGPU device over PCI-Ex bus

l Data transfer between CPU hosts over Infiniband

m Data transfer from GPGPU device to CPU host over PCI-Ex bus

n CPU global memory accesses

o GPGPU global memory accesses

p GPGPU FLOPs

q Data transferred in gather operation

We consider the variables - *x* as the vertical dimension per process, *Y* as the horizontal dimension common for all processes, and *G* as the number of ghost rows. Since the application of interest is an SIA, the number of iterations executed in the application plays an important role in the execution time. We denote the "set" number of iterations as *"K"*. However, since the application makes use of rows of ghost data, the "effective" number of iterations performed in the application is reduced by a factor of *G* since *G* rows are computed in a single iteration. Hence,

$$\text{Effective number of iterations} = K/G \tag{4.4}$$

In what follows, we elucidate the two vectors for the multiprocessor cluster and the GPGPU-enabled cluster. For both clusters, the steps of scattering the initial waveform vector (vector *j*) and gathering the final waveform vector (vector *q*) from all processes are common. The steps for exchanging the ghost row data with neighboring processes are also common (vector *l*). Therefore, we conveniently eliminate these transactions from consideration since they will be identical on both cluster configurations. Nonetheless, we evaluate the application and accelerator components for these transactions before safely eliminating them.

If $BW_{scatter}$, BW_{gather}, and $BW_{send\text{-}recv}$ represent the bandwidth for scatter, gather and send-receive operations respectively, the accelerator vector for these operations can be computed as shown in Equation 4.5.

$$\text{Accelerator vector} = j / BW_{scatter} + q / BW_{gather} + l / BW_{send\text{-}recv} \quad (4.5)$$

The data size transferred in bytes for both the scatter and gather operations is $36.x.(Y+2)$ and that the send-receive operation is $36.G.(Y+2)$ per iteration. The send-receive operation takes place for K/G iterations. Therefore, the application vector for these operations is shown in Equation 4.6 and the scalar product of these components is computed as shown in Equation 4.7:

$$\text{Application vector} = 36.x.(Y+2)\, j + 36.x.(Y+2)\, q + 36.(Y+2).K\, l \quad (4.6)$$

$$\text{Scalar Product} = 36.x.(Y+2) / BW_{scatter} + 36.x.(Y+2)/ BW_{gather} + (36.(Y+2).K)/BW_{send\text{-}recv}. \quad (4.7)$$

As explained, these components are common for both clusters and are henceforth, excluded from consideration.

4.2.1 Multiprocessor Cluster:

For this architecture, all of the floating-point calculations are performed by the processors implying that the number of FLOPS is the accelerator parameter that has the highest impact. Although the processor must fetch data from memory, the memory bandwidth is assumed to be high enough to not incur any data access latency (vector *n*). Therefore, only the unit vector *i* is featured in both the vectors. With an Intel Xeon E5410 processor, the benchmark performance is taken as 153.6 Giga FLOPS. We use the unit vector *i* and Equation 4.8 shows the accelerator vector:

$$\text{Accelerator vector} = i / (153.6 * 10^9) \text{ second/FLOPs} \quad (4.8)$$

For the application vector, the processors need to compute two intermediate vectors, each with 32 FLOPs per data element. The application performs calculations over G ghost rows at a time leading to a total data size of $(Y+2)*(x+G+1)*G$ over a set of G ghost rows. The final vector requires 44 FLOPs per data element over the same data size. Equation 4.9 shows the application vector for a single iteration:

$$\text{Application vector} = (32.(Y+2).(x+G+1).G + 32.(Y+2).(x+G+1).G + 44.(Y+2).(x+G+1).G)i \text{ FLOPs}$$

$$= 108.(Y+2).(x+G+1).G \text{ FLOPs} \quad (4.9)$$

Over K/G iterations, the final application vector is shown in Equation 4.10:

$$\text{Application vector} = 108.(Y+2).(x+G+1).G * K/G \text{ FLOPs}$$

$$= 108.(Y+2).(x+G+1).K \text{ FLOPs} \quad (4.10)$$

Equation 4.11 shows the scalar product of the two vectors:

$$\text{Scalar Product} = (108.(Y+2).(x+G+1).K) / (153.6 * 10^9) \text{ seconds} \quad (4.11)$$

4.2.2 Heterogeneous GPGPU architecture

For this architecture, we consider the host to device transfer (vector k), device to host transfer (vector m), global memory accesses (vector o, vector q) and GPGPU FLOPs (vector p) as the contributing components. For the NVIDIA Tesla K20 GPGPU, the following benchmarks are considered:

Host to device data rate	3.28 Gigabytes / second
Device to host data rate	2.83 Gigabytes / second
Global memory bandwidth	208.11 Gigabytes / second
GPGPU FLOPs per second (FLOPS)	1160.5 Giga FLOPS

Using this data, the accelerator vector can be computed as shown in Equation 4.12:

$$\text{Accelerator vector} = [k / (3.28*10^9) + m / (2.83*10^9) + o / (320*10^9) + q / (147.14*10^9)] \text{ second/GBytes} + p / (1160.5*10^9) \text{ second/GFLOPs} \quad (4.12)$$

To compute the application vector, we construct the application vectors piecewise, by considering each communication and computation transaction.

4.2.2.1 One Time Host-Device Data Transfers

Each host process has to transfer its portion of the waveform vector to the GPGPU device before proceeding to the iterative stage (vector k). Similarly, after completing all iterations of the algorithm, each host process receives the final processed waveform vector from the GPGPU device (vector m). The data size transferred in both the cases is $36.(x+2.G).(Y+2)$ where x, Y and G are as explained above. Equation 4.13 shows the application vector resulting from the initial and final host to device and device to host transfers:

$$\text{One time Transfers: } 36.(x+2.G)(Y+2).k + 36.(x+2.G)(Y+2)m \text{ bytes} \quad (4.13)$$

4.2.2.2 Iterative Host-Device Data Transfers

During each iterative step, the GPGPU device first, receives top and bottom ghost row data from its host processor (vector k). Each process undergoes ghost row data exchange with its neighboring process prior to this step (vector l) but as explained above, we eliminate this data transaction from consideration. After receiving this data, the GPGPU device executes the kernel and at the end of the iteration, transfers the freshly computed top and bottom edge data also comprising of G rows, to the host process

(vector *m*). The host processes then exchange this data at the start of the next iteration. The ghost data size is a function of *G* and *Y* and is expressed as 36.*G*.(*Y*+2). Using the effective number of iterations, Equation 4.14 shows the application vector components corresponding to the iterative data transfers:

$$\text{Iterative Transfers} = (72.G.(Y+2).k + 72.G.(Y+2)m) * K/G \text{ bytes}$$
$$= K.(72.(Y+2).k + 72.(Y+2)m) \text{ bytes} \quad (4.14)$$

4.2.2.3 Iterative Computation Component

In the computing step, each GPGPU thread requires 98 FLOPs (vector *p*) to compute a vector element. Using a Block size of 256 threads, the total number of threads (N_{th}) are calculated as shown in Equation 4.15:

$$N_{th} = ceil((Y+2)/16) * ceil((x+2.G)/16) * 256 \quad (4.15)$$

On computing over *G* ghost rows, the total number of FLOPs are (98.N_{th}.*G*) *p* where *p* is the corresponding unit vector

Equation 4.16 shows this component over *K/G* iterations:

$$\text{Number of FLOPs} = 98.N_{th}.K\ p \text{ FLOPs} \quad (4.16)$$

Each GPGPU thread makes 32 global memory accesses (vector *o*) threads per *G* iterations. Over the total number of threads N_{th} and the effective *K/G* iterations, the global memory accesses are as shown in Equation 4.17:

$$\text{GPGPU global memory accesses} = 32.N_{th}.K\ o \text{ bytes} \quad (4.17)$$

Equation 4.18 shows the final application vector by using Equations 4.13, 4.14, 4.16 and 4.17:

Application vector = [36.(x+2.G)(Y+2).k + 36.(x+2.G)(Y+2).m + K.(72.(Y+2).k + 72.(Y+2).m)] bytes + K. Nth.(98.p FLOPs + 32.o bytes) (4.18)

Equations 4.12 and 4.18 are used to calculate the scalar product of the two vectors as shown in Equation 4.19:

Scalar product = (36.(x+2.G)(Y+2) + 72.K.(Y+2)) / (3.28*10^9) + (36.(x+2.G)(Y+2) + 72.K.(Y+2)) / (2.83*10^9) + 32.K / (320*10^9) + 36.K.(Y+2)(x+G+1) / (147.14*10^9) + (98. Nth. K) / (1160.5*10^9) (4.19)

4.2.3 Results

The following table 4.1 shows the scalar products of both the architectures with the independent variables being the application dimension in one direction Y, number of ghost rows G, number of processes P, and the set number of iterations K. For different combinations of these parameters, the fitness model predicts the approximate runtime on the two accelerators and ranks them. The fitness model is not responsible for predicting the accurate runtime but is only used to rank the accelerators. From the table, it is evident that the GPGPU cluster is bound to perform better and is predicted to be considerable faster than its counterpart. Therefore, it is chosen as the best-fit architecture.

Table 4.1: Results of using the Fitness Model on the accelerators of interest

Parameters				**Execution Time (sec)**		**Relative Speedup** ($T_{CPU}/T_{CPU\text{-}GPGPU}$)
Dimensions *YxY*	**Processes *P***	**Ghost Rows *G***	**Set Iterations *K***	**Heterogeneous CPU-GPGPU cluster ($T_{CPU\text{-}GPGPU}$)**	**Homogeneous multi CPU cluster (T_{CPU})**	
2000x2000	4	8	1000	0.313	7.165	22.909
8000x8000	8	12	2000	3.936	113.991	28.965
12000x12000	16	16	1000	2.503	64.726	25.861
16000x16000	32	20	2000	4.889	117.240	23.987

4.3 SIA to Accelerator mapping

This section describes how the computation and communication components of the application are mapped to the CPU and GPGPU cores in the heterogeneous cluster. Certain optimizations are performed in this process. We describe the application mapping in section 4.2.1 and the optimizations in section 4.2.2.

4.3.1 Application Mapping

The key computation step is the finite difference method; therefore this can be performed on the GPGPUs. To simulate initial conditions, the master host CPU initializes the u and v velocities of the entire square surface and the height is initialized to a two dimensional Gaussian wave that represents a water droplet.

After this initial processing, the master process scatters the square water surface region, also known as the surface vector 1, to all other processes in a rectangular row striped fashion. Each process initializes its respective GPGPU device by allocating memory for the surface vector 1 and two intermediate vectors - vector 2 and vector 3 required for the first half step shown in Equations 3.9 and 3.10. Further, for each process, a host to device transfer of only the surface vector takes place; the F and G vectors get initialized during the kernel computation step. This is a one-time host to device transfer step.

As mentioned in a previous chapter, the application is designed to operate using ghost data. This data consists of auxiliary points that reside along all four edges of the per process surface vector. These points are used to calculate the values of the intermediate and surface vector points along the boundaries of the surface in the finite difference method step. Along the left and right vertical edges, there is a single column of ghost

points but along the top and bottom edges, there can be multiple configurable number of ghost rows. Since the surface vector is scattered in a row striped manner, each process can send the top and bottom valid surface data and receive the top and bottom ghost data from its neighbor. The valid data for one process acts as the ghost data for its neighbor and vice versa.

The iterative step commences with each process participating in a send-receive of ghost data. Each process then transfers this data to the GPGPU device it is coupled with. Three GPGPU kernels are designed that perform the following tasks.

Kernel 1 is used to update the boundary conditions. For the GPGPU device coupled with the master process, the reflections at the topmost row is updated and the condition $v = -v$ is applied. Similar is the case for bottommost row for the device coupled with the last process. For all other processes, the GPGPU threads along the right and left edges update the velocity u with the condition, $u = -u$. Only the surface vector U is updated in this kernel. No floating point operations take place.

Kernel 2 evaluates the intermediate vectors 2 and 3 in this first half step of the Lax-Wendroff method. To compute vector 2, the U and F components of surface vector 1 are loaded in shared memory. Equation 3.9 is then solved by all GPGPU threads and vector 2 is updated. To compute vector 3, the U and G components of surface vector 1 are loaded in shared memory. All GPGPU threads compute equation 3.10 and update vector 3.

Kernel 3 uses the intermediate vectors computed in kernel 2 in the calculation of the surface vector 1. The GPGPU threads compute equation 3.11 and the time step is completed. The F and G components of the intermediate vectors 2 and 3 respectively, are

loaded into shared memory since Equation 3.11 requires both these components. GPGPU threads work in parallel and the U, F and G components of vector 1 are updated at the end of this kernel.

After the kernel computations, the freshly computed surface vector data near the top and bottom edges are transferred from GPGPU device to respective host processes. This data acts like the ghost data that the processes exchange with their neighbors at the start of a new iteration. The steps of exchanging ghost data with neighboring processes, host to device transfer of the ghost data, execution of the three kernels and a device to host transfer of the freshly computed edge data take place for a *K/G* number of iterations. A value of *K* can be specified by the SIA whereas the *G* value can be selected to obtain the best runtime.

At the end of all iterations, the entire surface vectors present on each GPGPU device is transferred to its host. This is also a one-time transaction. Each process participates in a gather operation and the complete water surface vector is collected on to the master process. The application can be summarized in the following steps.

1. Master process initializes the two dimensional Gaussian peak, vector velocities and scatters the surface vector 1.
2. Each process transfers the received scattered vector to its GPGPU device.
3. For each iteration,

a. Neighboring processes exchange ghost data in a send-receive operation

b. Each process transfers the ghost data to GPGPU device

c. GPGPU device executes kernels 1, 2 and 3

d. GPGPU device transfers fresh edge data to its host process

4. GPGPU device transfers the final surface vector 1 to host process
5. The entire surface vector is gathered on to the master process

Figure 4.1 summarizes the SIA flow on the heterogeneous CPU-GPGPU cluster.

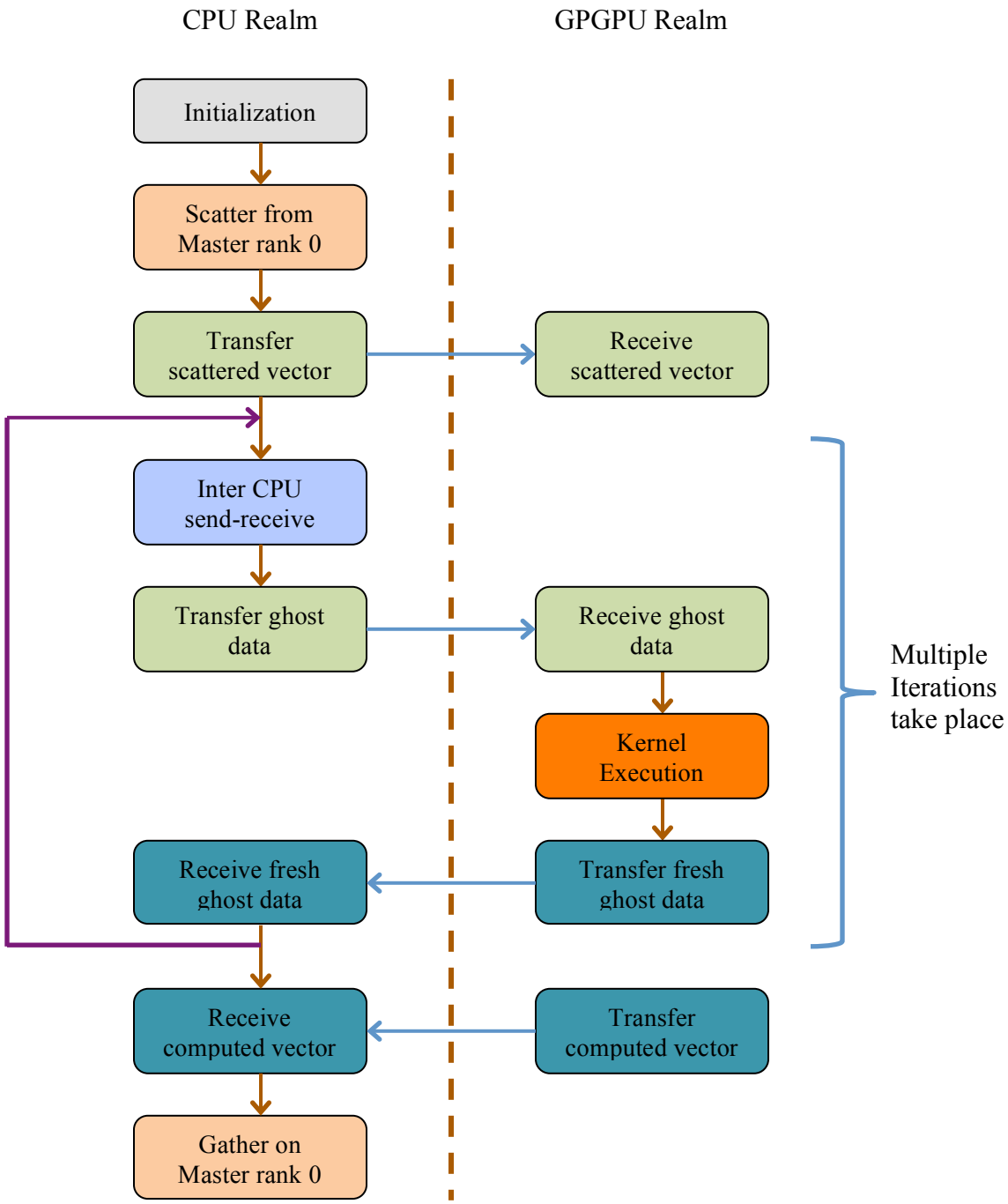


Figure 4.1: SIA Mapping on Heterogeneous CPU-GPGPU Cluster

4.3.2 Optimizations

Optimization techniques make the kernels highly efficient. Two such techniques are elaborated - the use of shared memory and data structure access by using registers:

4.3.2.1 Shared Memory

This memory is used to reduce the number of global memory accesses by threads since global memory incurs long access latencies. The use of shared memory significantly increases the Compute to Global Memory Access (CGMA) ratio. As discussed above, shared memory is used in kernels 2 and 3 and has dimensions of (TILE_WIDTH+1).(TILE_WIDTH+1). A TILE_WIDTH of 16 is used and the size of (TILE_WIDTH+1) is justified for border elements in the finite difference step. In kernel 2, each thread *t(i,j)* requires vectors from the neighboring threads *t(i,j+1)* and *t(i+1,j+1)* to calculate the F vector component and from threads *t(i+1,j)* and *t(i+1,j+1)* to calculate the G component. In kernel 3, to calculate the final vector at the end of the half steps, each thread *t(i+1,j+1)* requires F vector component from the neighboring threads *t(i+1,j)* and *t(i,j)* and the G vector component from threads *t(i,j+1)* and *t(i,j)* to calculate the G component. Figure 4.2a, 4,2b and 4,2c show how shared memory locations are used to compute equations 3.9, 3.10 and 3.11 respectively.

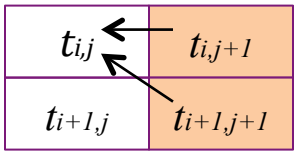


Figure 4.2a

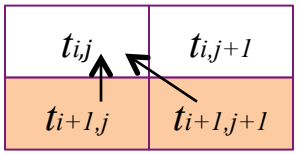


Figure 4.2b

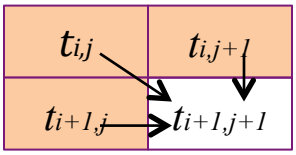


Figure 4.2c

4.3.2.2 Data Structure Access and Use of Registers

The initial versions of kernel 2 and 3 had the vectors implemented as arrays. Since local array variables reside on the global memory, those accesses would have incurred long latencies. The implementation was modified such that the *U*, *F* and *G* vectors were structures of 3 elements. Local vector variables could thus, fit into registers that have negligible access latencies. This also allowed for loop unrolling and each component of the *U*, *F* or *G* vectors could be independently computed using registers. Use of structures also allowed updating a vector element in global memory in a single operation. After this modification, kernel 2 had occupancy of 100% whereas kernel 3 had occupancy of 83%.

4.4 Comparison of Accelerators

In this section, we validate the findings of the A2A mapping by comparing the performance of the application on both accelerators. We keep the number of iterations *K* fixed at 400 and use 8 ghost rows *G*. We consider performance over smaller problem sizes since we only need to verify the A2A mapping. Table 4.2 provides this data.

Table 4.2: Comparison of performance to verify A2A mapping

Parameters		**Execution Time (sec)**		**Relative Speedup ($T_{CPU}/T_{CPU\text{-}GPGPU}$)**
Configuration	**Dimensions *YxY***	**Heterogeneous CPU-GPGPU cluster ($T_{CPU\text{-}GPGPU}$)**	**Homogeneous multi CPU cluster (T_{CPU})**	
2-Node	512x512	0.435	126.093	289.868
4-Node	1024x1024	1.925	249.139	129.422
8-Node	2048x2048	6.011	508.811	84.646
16-Node	4096x4096	12.413	1035.038	83.383

The relative speedup indicates that the heterogeneous CPU-GPGPU cluster performs far better than the homogeneous cluster. Therefore, we can ascertain that the

A2A mapping provides an accurate ranking of available accelerators for a particular application.

4.5 Summary

In this chapter, we described the A2A mapping process in brief and used it to rank the available accelerators for the selected application. The A2A mapping indicates that the heterogeneous CPU-GPGPU cluster is expected to outperform the homogeneous cluster and this is verified by the short scale implementation of the application on both accelerators. We also described the application mapping on the heterogeneous CPU-GPGPU cluster and the optimizations performed. The problem dimensions and number of iterations specify the application whereas, the number of ghost rows and number of process are flexible and should be selected in order to attain the best performance of the application

CHAPTER 5

LOW LEVEL ABSTRACTION USING THE SIGE MODEL

In this chapter, the low level abstraction of the SIGE model [16] is used to perform regression analysis for runtime prediction. The chapter is organized as follows - an introduction to the SIGE model and low level abstraction is presented. Thereafter, the details of the low level abstraction are included. The runtime prediction is carried out independently for the computation and communication sections of the algorithm.

5.1 Overview of the Low-Level Abstraction of the SIGE model

The low-level abstraction methodology presented in the SIGE model aims to put forth a model for performance prediction using limited algorithm implementation details. The model aims to abstract the underlying system architecture by measuring the performance of the application under different workloads on the architecture of interest. The runtime prediction framework models the computation and communication sections of the algorithm independently. The computation section is further broken down into computations carried out on the CPU host and that on the GPGPU device. Similarly, the communication section comprises of components like the inter host-device transfers over PCI-Ex bus and the inter CPU host communications over Infiniband.

The computation component is modeled using readily available algorithm characteristics such as the number of FLOPs (floating point operations), amount of data required for computations and the communication component depends on characteristics like amount of data transferred, bandwidth offered by the architecture, number of processes, etc. Since the algorithm of interest is an SIA, the total computation and

communication time is a function of the number of iterations that the algorithm runs for. The overall execution time is the sum of the individual computation and communication runtimes. In order to accurately carry out runtime prediction, several instrumented executions of the algorithm are carried out and regression analysis is performed on the collected data using the R tool-chain for regression analysis [29]. The parameters are chosen based upon their high R^2 values (greater than 0.95) and low p-values (less than 0.10) in order to yield high prediction accuracy. The samples are collected using an appropriate range of the problem size (up to 6000 x 6000). If *x1*, *x2*, *x3* … are the independent variables with coefficients *a1*, *a2*, *a3* … , then the dependent variable *t* can be determined by the following equation with *e* as the error difference as shown in Equation 5.1.

$$t = a1.x1 + a2.x2 + a3.x3 + ... + e \quad (5.1)$$

5.2 Application of Low-Level Abstraction

In this section, the low level abstraction of the SIGE model discussed above is applied to the algorithm of interest. The application is partitioned into computation and communication sections for this purpose. Since no computation operations take place on the CPU host, the computation component only depends on those carried out on the GPGPU device. The prediction framework uses the measured runtime data for the computation component runtime for a single iteration. The total computational runtime is calculated by scaling this prediction with the total number of iterations. The communication component comprises of several host-device data transfers and inter-host data transfers. These transfers can be classified as one-time and iterative data transfers. The one-time transfers include a scatter operation of the surface vector, a host-device

transfer of the scattered vector, a device-host transfer of the final processed vector, and finally a gather operation of the processed vector. The inter CPU host transfers of the ghost data and host-device and device-host transfer of the ghost data constitute the iterative transfers. Similar to the computation component runtime, the total iterative communication component runtime is obtained by scaling the prediction for a single iteration with the total number of iterations. The following Equations 5.2 to 5.6 express the different runtime components.

$$T_{execution\text{-}time} = T_{computation} + T_{communication} \quad (5.2)$$

$$T_{computation} = T_{GPGPU\text{-}kernel} * \text{Set } number\ of\ iterations \quad (5.3)$$

$$T_{communication} = T_{one\text{-}time\text{-}communication} + T_{iterative\text{-}communication}. \quad (5.4)$$

$$T_{one\text{-}time\text{-}communication} = T_{scatter} + T_{gather} + T_{host\text{-}device\text{-}once} + T_{host\text{-}device\text{-}once} \quad (5.5)$$

$$T_{iterative\text{-}communication} = (T_{send\text{-}recv} + T_{host\text{-}device\text{-}iterative} + T_{device\text{-}host\text{-}iterative}) * Effective\ number\ of\ iterations \quad (5.6)$$

As discussed in chapter 4, the effective number of iterations is given by *K/G* where *K* is the set number of iterations and *G* is the number of ghost rows. Equations 5.7 and 5.8 show the total computation and iterative communication timings:

$$T_{computation} = T_{GPGPU\text{-}kernel} * (K) \quad (5.7)$$

$$T_{iterative\text{-}communication} = (T_{send\text{-}recv} + T_{host\text{-}device\text{-}iterative} + T_{device\text{-}host\text{-}iterative}) * (K/G). \quad (5.8)$$

In what follows, the details of the computation and communication performance modeling is elucidated.

5.2.1 Computation component

As discussed, the computation component modeling only depends upon the computations carried out on the GPGPU device. The computation kernels perform the tasks of updating the reflections at the boundaries of the surface and compute the value of each element using finite difference method. Therefore, the data points that constitute the perimeter of the surface (*EDGE-DATA*) and the total data points used for computations (*COMP-DATA*) are considered as predictor variables.

The number of FLOPs required for computation are also considered as predictors but this parameter is abstracted whilst considering the total data size. The coefficients of the regression equations capture the impact of FLOPs on data size and therefore, it is not explicitly considered. This simplifies the regression analysis and should be noted as strength of the SIGE model.

The *COMP-DATA* and *EDGE-DATA* parameters are functions of the problem size dimensions (*Y*), number of ghost rows (*G*) and number of processes (*P*). Equations 5.9 to 5.12 below represent the runtime predictions for 2, 4, 8, and 16-node configurations. Both the data components are in bytes.

$$T_{2\text{-}node} = -1.43e\text{-}5 * EDGE\text{-}DATA + 4.5e\text{-}6 * COMP\text{-}DATA + 0.02618 \quad (5.9)$$

$$T_{4\text{-}node} = -1.76e\text{-}5 * EDGE\text{-}DATA + 4.39e\text{-}6 * COMP\text{-}DATA + 0.02716 \quad (5.10)$$

$$T_{8\text{-}node} = -2.05e\text{-}5 * \text{EDGE-DATA} + 4.77e\text{-}6 * COMP\text{-}DATA + 0.0298 \quad (5.11)$$

$$T_{16\text{-}node} = -3.28e\text{-}5 * EDGE\text{-}DATA + 4.51e\text{-}6 * COMP\text{-}DATA + 0.03622 \quad (5.12)$$

The following Figures 5.1 and 5.2 show the behavior of the kernel runtime with respect to the predictor variables.

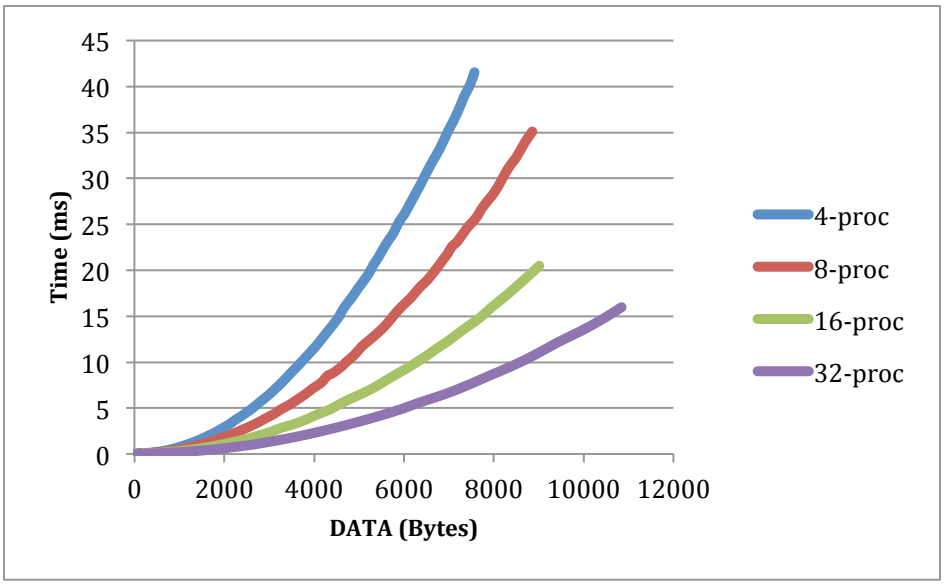


Figure 5.1: Kernel Runtime (ms) vs *EDGE-DATA* (bytes)

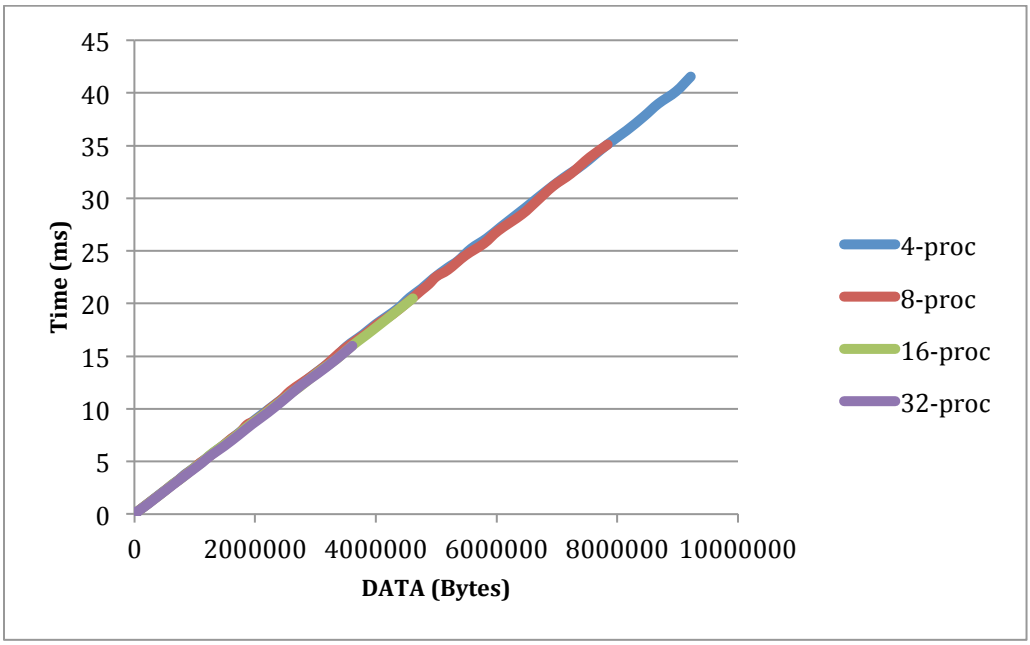


Figure 5.2: Kernel Runtime (ms) vs *COMP-DATA* (bytes)

5.2.2 Communication Component.

All the communication micro-benchmarks can be adequately represented by a Michaelis Menten curve [33]; therefore these components are modeled using the Michealis Menten kinetics. In this model, the rate of reaction (*BW*) is a function of the substrate concentration (*DATA*) along with the constants V_{max} and K_m. The general form of this model is described by the following Equation 5.13.

$$v = \frac{V_{max}\,[S]}{K_m + [S]} \tag{5.13}$$

Here, V_{max} represents the maximum rate achievable by the reacting system and K_m is the substrate concentration when reaction rate is half of that of the maximum. In this case, the data bandwidth offered by the communication link can be represented by the reaction rate *BW* and the data size by *DATA*. The type of communication link – PCI-Ex or Infiniband, is abstracted by this model and therefore, this model can be conveniently applied for all communication components. The units for data size and bandwidth are megabytes (MB) and megabytes per second (MB/sec) respectively.

Equation 5.14 shows how the execution time (*T*) is calculated using Equation 5.13:

$$T = (DATA + K_m) \,/\, V_{max} \tag{5.14}$$

In this section, we model the one-time and iterative communication components independently.

5.2.2.1 One Time Communication Components

These communication components consist of the scatter and gather of the entire surface vector, and host to device and device to host transfer of scattered surface vector.

The data size for the scatter and gather operations depend on the dimensions of the problem size. The following Equations 5.15 to 5.18 represent the communication time for this operation on 2, 4, 8, and 16 nodes, respectively.

$$T_{2\text{-}node} = (DATA + 0.0109) / 230.92 \quad (5.15)$$

$$T_{4\text{-}node} = (DATA - 0.0331) / 156.32 \quad (5.16)$$

$$T_{8\text{-}node} = (DATA + 0.8064) / 134.38 \quad (5.17)$$

$$T_{16\text{-}node} = (DATA + 1.0986) / 125.71 \quad (5.18)$$

The scatter micro-benchmarks are displayed in the Figure 5.3 below.

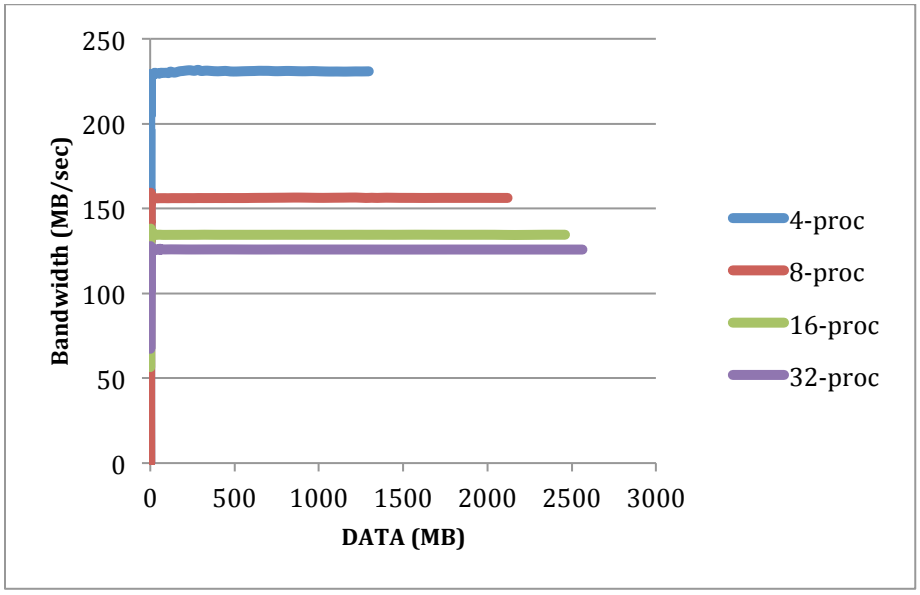


Figure 5.3: Scatter Bandwidth (Megabytes/sec) vs *DATA* (Megabytes)

The following Equations 5.19 to 5.22 represent the communication time for the gather operation on 2, 4, 8, and 16 nodes, respectively.

$$T_{2\text{-}node} = (DATA + 0.0769) / 230.24 \quad (5.19)$$

$$T_{4\text{-}node} = (DATA + 48.8688) / 155.25 \quad (5.20)$$

$$T_{8\text{-}node} = (DATA + 0.1773) / 134.49 \quad (5.21)$$

$$T_{16\text{-}node} = (DATA + 155.60) / 123.26 \quad (5.22)$$

The gather micro-benchmarks are displayed in the Figure 5.4 below.

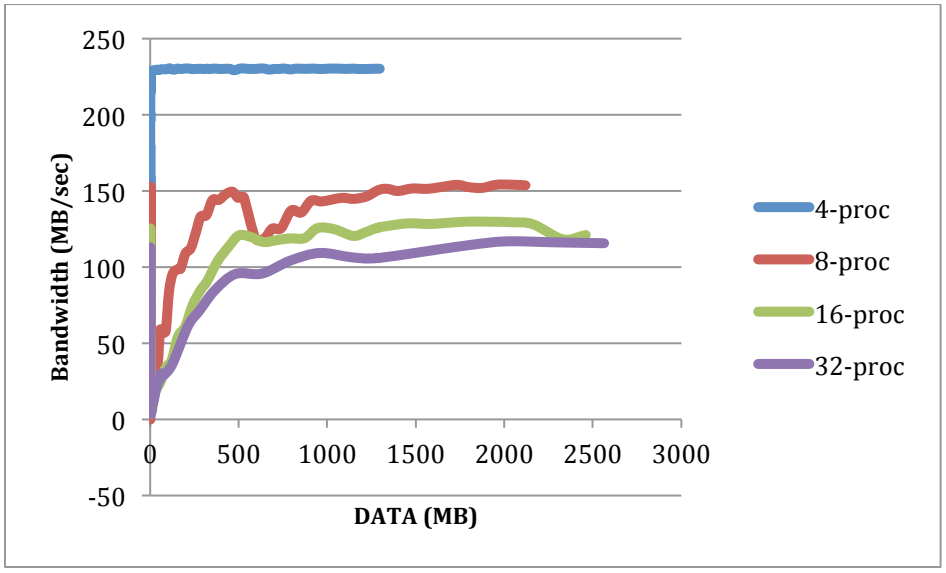


Figure 5.4: Gather Bandwidth (Megabytes/sec) vs *DATA* (Megabytes)

The data size parameter *DATA* for the host to device and device to host operations depend on the problem size dimensions, number of processes, as well as the number ghost rows employed by the algorithm. Since each node constitute two CPU host - GPGPU device pairs, a single equation is sufficient to represent the communication time for the CPU host to GPGPU device communication. However, we observed a distinct

behavior for the 16-node configuration and hence, model it separately. The following Equation 5.23 is used for the 2-node, 4-node and 8-node configurations on account of similarity in the bandwidth behavior.

$$T_{host\text{-}to\text{-}device} = (DATA + 0.4103) / 2446.34 \quad (5.23)$$

Equation 5.24 below is used to model the 16-node configuration.

$$T_{host\text{-}to\text{-}device\text{-}16\text{-}node} = (DATA + 0.6698) / 1485.77 \quad (5.24)$$

The host-to-device micro-benchmarks are displayed in the Figure 5.5 below.

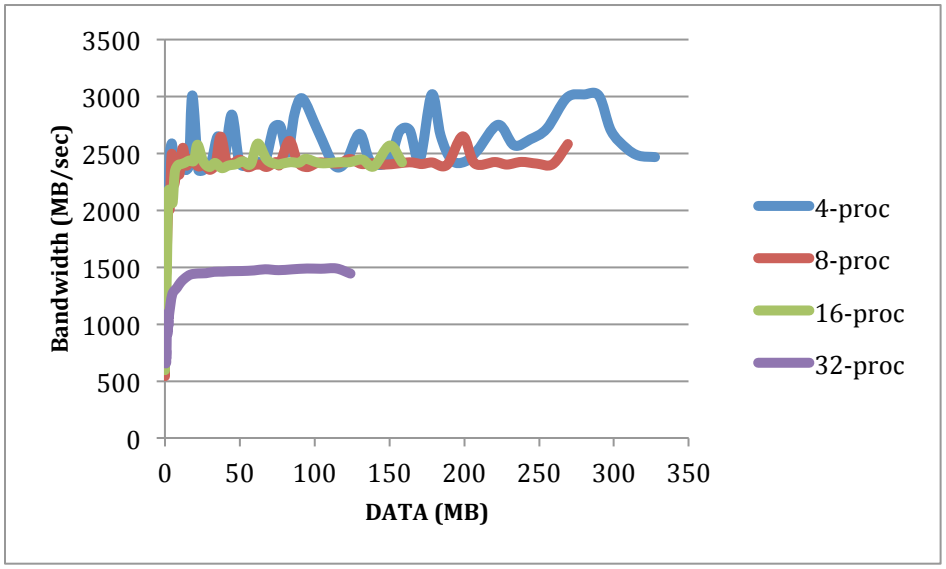


Figure 5.5: One time Host-Device transfer bandwidth (Megabytes/sec) vs *DATA* (Megabytes)

For the GPGPU device to CPU host communication, a similar behavior is observed with the bandwidth for the 16-node configuration being different from the other configurations. The following Equation 5.25 represents the behavior for 2-node, 4-node and 8-node configurations.

$$T_{device\text{-}to\text{-}host} = (DATA + 0.2423) / 2733.948 \quad (5.25)$$

Equation 5.26 below is used to model the 16-node configuration.

$$T_{device\text{-}to\text{-}host\text{-}16\text{-}node} = (DATA + 0.3924) / 1568.68 \quad (5.26)$$

The device to host micro-benchmarks are displayed in the Figure 5.6 below.

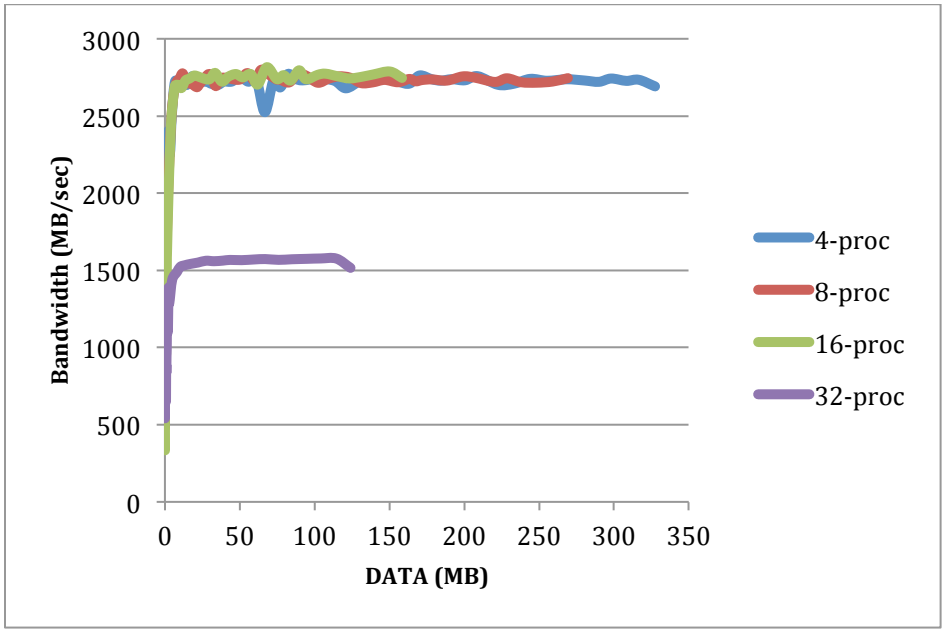


Figure 5.6: One time Device-Host transfer bandwidth (Megabytes/sec) vs *DATA* (Megabytes)

5.2.2.2 Iterative Communication Components

These communication components consist of the host to device, device to host, and the inter CPU host transfer of the ghost data. The number of nodes and the number of ghost rows decide the payload and therefore, the latencies for the iterative communication components. Similar to the computation component, the total communication time of each component is directly proportional to the number of iterations of the algorithm.

The data size for all the iterative communication components depend on the problem size dimension and the number of ghost rows. The data size for the send-receive communications is twice of that for the host-device transfers. This is because, while each GPGPU device transfers the top and bottom data to their respective hosts in each iteration, each CPU host has to send as well as receive the top and bottom ghost data from its neighbor. The regression coefficients, V_{max} and K_m abstract this notion.

The following Equations 5.27 to 5.30 represent the communication time for the send-receive operation for a single iteration for 2, 4, 8, and 16 nodes, respectively. The total time is obtained by scaling this time with the effective number of iterations (K/G)

$$T_{2\text{-}node} = (K/G) * (DATA + 0.0489) / 56.059 \quad (5.27)$$

$$T_{4\text{-}node} = (K/G) * (DATA + 0.0591) / 47.851 \quad (5.28)$$

$$T_{8\text{-}node} = (K/G) * (DATA + 0.0431) / 44.676 \quad (5.29)$$

$$T_{16\text{-}node} = (K/G) * (DATA + 0.0415) / 55.705 \quad (5.30)$$

The send-receive micro-benchmarks are displayed in the Figure 5.7 below.

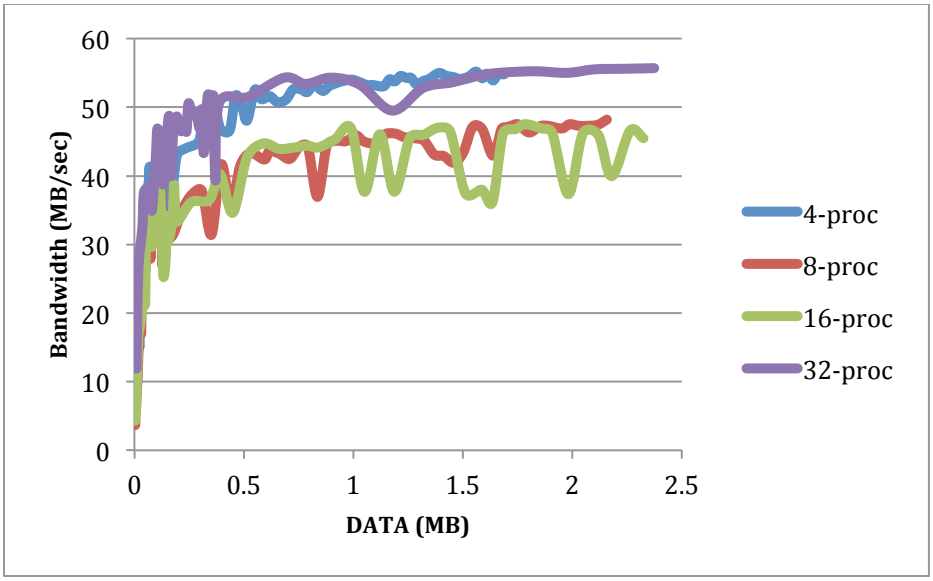


Figure 5.7: Send-Receive Bandwidth (Megabytes/sec) vs *DATA* (Megabytes)

The following Equations 5.31 to 5.34 represent the communication time for the host to device and device to host transfer for a single iteration for 2, 4, 8, and 16 nodes. The total time is obtained by scaling this time with the effective number of iterations (K/G). For the iterative cases too, we model the 16-node and other configuration separately. The following Equations 5.31 and 5.32 represent behavior for the 2-node, 4-node and 8-node configurations.

$$T_{host\text{-}to\text{-}device} = (K/G) * (DATA + 0.1861) / 2137.511 \quad (5.31)$$

$$T_{device\text{-}to\text{-}host} = (K/G) * (DATA + 0.1853) / 2500.843 \quad (5.32)$$

The following equations 5.33 and 5.34 are used to model the 16-node configuration.

$$T_{host\text{-}to\text{-}device\text{-}16\text{-}node} = (K/G) * (DATA + 0.2577) / 1212.814 \quad (5.33)$$

$$T_{device\text{-}to\text{-}host\text{-}16\text{-}node} = (K/G) * (DATA + 0.1789) / 1413.621 \quad (5.34)$$

The host to device and device to host micro-benchmarks are displayed in the Figure 5.8 and Figure 5.9 below.

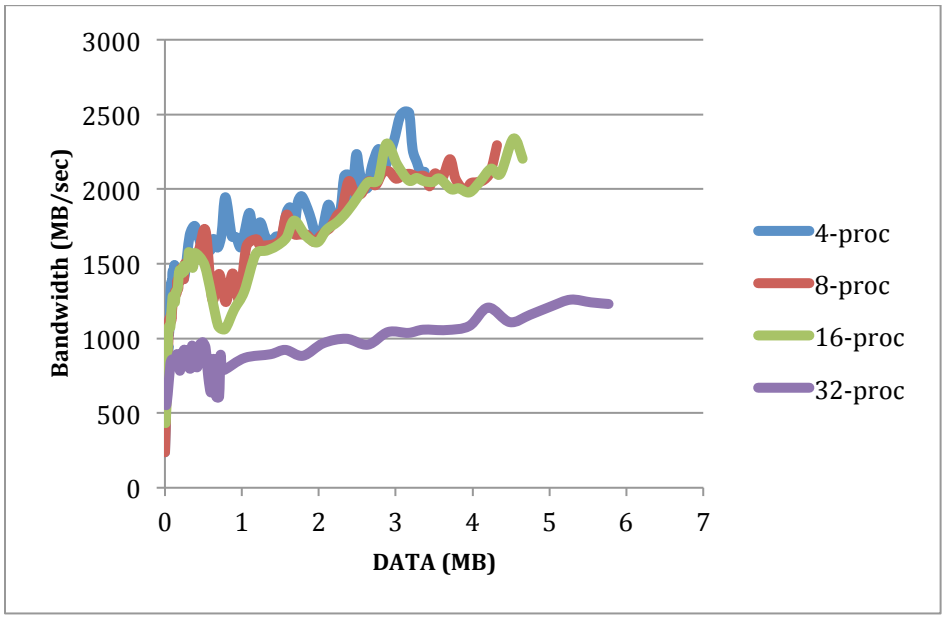


Figure 5.8: Iterative Host-Device transfer bandwidth (Megabytes/sec) vs *DATA* (Megabytes)

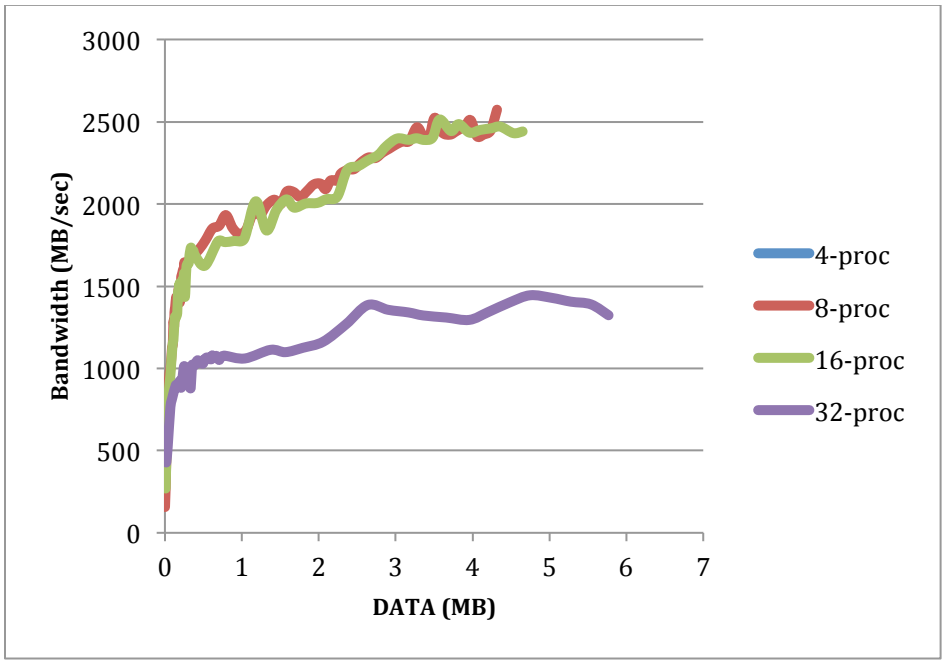


Figure 5.9: Iterative Device-Host transfer bandwidth (Megabytes/sec) vs *DATA* (Megabytes)

5.3 Summary

In this chapter, we elucidate how the low level abstraction of the SIGE model is carried out on the computation and communication sections of the algorithm. For the computation component, the FLOPs parameter is abstracted by the coefficients of the regression equations when the total data size is considered. For the communication component, we model the iterative and one-time components. The CPU host – GPGPU device communications are one-time as well as iterative but these are modeled separately. These transfers are independent of the node configuration but we model the behavior for the 16-node configuration independently from the other configurations on account of a distinct change in behavior. For all iterative communication components, the data transferred in the inter-CPU communication is twice that of the inter host-device communications and is represented in by the regression equation parameters.

CHAPTER 6

RESULTS AND ANALYSIS

In this chapter, we present the experimental verification of the low level abstraction of the SIGE model that was applied on the shallow water wave application in the previous chapter. In section 6.1, the predicted and actual runtimes of the computation and communication components as well as the overall application as a whole is enumerated along with the errors observed. This is followed by section 6.2 in which we draw certain insights based on the runtime predictions provided by the low level abstraction. Section 6.3 consists of a Strength, Weakness, and Opportunities (SWO) analysis of the SIGE model. The chapter concludes with a summary in section 6.4.

6.1 Runtime Predictions from the SIGE model.

In this section, the runtime predictions for the computation and communication components are presented. The application is executed on 2, 4, 8, and 16 nodes at larger problem sizes, for different number of ghost rows. For each case, we consider problem sizes up to the maximum limit that the hardware resources can support, such as the amount of memory on the Tesla K20 GPGPUs. These problem sizes are significantly higher than those considered for benchmarking purposes. For each of the predictions, we have identified four independent parameters that control the application runtime – problem size dimension (Y), number of processes (P), set number of iterations (K), number of ghost rows (G). The runtimes presented in the sections below are for a single iteration of the computation kernels as well as the communication components. We vary Y, P, and G and compare the predicted and observed runtimes.

6.1.1 Computation Component

Table 6.1 presents the predicted and experimental runtimes for the computational component for different node configurations. We predict the runtimes using Equations 5.9, 5.10, 5.11 and 5.12.

Table 6.1: Observed and Predicted Values for Computation Component (ms)

Configuration	**Dimensions** ***Y x Y***	**Ghost Rows** ***G***	**Predicted** **T**computation	**Observed** **T**computation	**Error in** **T**computation **(%)**
2-Node	8000x8000	10	29034.843	28849.719	0.63759
	10000x10000	20	45648.772	46667.927	-2.2326
	12000x12000	40	66437.66	70645.17	-6.33302
	13000x13000	80	79693.126	92164.315	-15.64901
4-Node	12000x12000	10	32682.005	32724.455	-0.12989
	14000x14000	20	44917.472	46295.484	-3.06787
	16000x16000	40	59667.759	63301.7	-6.09029
	18200x18200	80	79476.638	92583.812	-16.49186
8-Node	18000x18000	10	36665.523	36760.762	-0.25975
	20000x20000	20	45917.403	47242.27	-2.88533
	22000x22000	40	56990.84	61779.433	-8.40239
	25360x25360	80	78827.484	90824.97	-15.21993
16-Node	28000x28000	10	44733.22	44779.879	-0.10431
	30000x30000	20	52436.121	53750.225	-2.5061
	32000x32000	40	61774.496	64895.918	-5.05293
	35104x35104	80	78945.92	89168.992	-12.94946

The observed runtimes are in good agreement with the predicted runtimes for medium problem sizes and have a low error rate (about 6%). The deviation increases up to 16% at the largest problem size. The problem dimensions are chosen with a view to maximize the memory usage and verify the prediction models at the highest possible memory usage allowed by the hardware. For each configuration, the highest problem size corresponds to the maximum possible memory usage of as high as 4.5 GB out of the 5GB global memory available on the K20 GPU. At such high memory usage, the global memory access times of the GPU could be larger leading to an increase in the observed computation time, and therefore, greater errors in the prediction. We conclude that the

prediction accuracy for such high problem dimensions could be improved by considering certain missing predictor variables as the memory usage approaches the hardware limits.

6.1.2 Communication Component

The one-time and iterative communication components are modeled and verified separately.

6.1.2.1 One-Time Communication Components

In this section, we compare the predicted and experimental runtimes for the scatter, gather, one time host-to-device transfers, and one time device-to-host transfers.

6.1.2.1.1 Scatter

Table 6.2 presents the predicted and experimental runtimes for the scatter operations on different processor configurations. The application is executed with different problem sizes Y as this is the only parameter that affects the scattered data. The Equations 5.15, 5.16, 5.17 and 5.18 are used to predict runtimes. We observe that the observed runtimes follow the predicted runtimes almost accurately with the error being less than 1% for most cases.

Table 6.2: Observed and Predicted Values for Scatter Component (ms)

Configuration	**Dimensions** ***Y x Y***	**Predicted** **T**scatter	**Observed** **T**scatter	**Error in** **T**scatter **%**
2-Node	8000x8000	9517.417	9528.746	-0.11903
	10000x10000	14870.209	14877.85	-0.05138
	12000x12000	21412.386	21441.354	-0.13529
	13000x13000	25129.504	25156.546	-0.10761
4-Node	12000x12000	31630.715	31634.236	-0.01113
	14000x14000	43051.848	43057.778	-0.01377
	16000x16000	56229.886	56246.557	-0.02965
	18200x18200	72755.032	72775.39	-0.02798
8-Node	18000x18000	82791.321	82671.332	0.14493
	20000x20000	102208.968	102048.402	0.1571
	22000x22000	123670.484	123499.148	0.13854
	25360x25360	164326.924	168017.512	-2.24588
16-Node	28000x28000	214129.864	214072.547	0.02677
	30000x30000	246072.125	245858.385	0.08686
	32000x32000	279674.636	279456.884	0.07786
	35104x35104	336559.326	336322.22	0.07045

6.1.2.1.2 Gather

Table 6.3 presents the predicted and experimental runtimes for the gather operations on different processors. Like the scatter operation, the gathered data depends only on problem size *Y*, so only this parameter is varied and the application is executed. We use Equations 5.19, 5.20, 5.21 and 5.22 for the predictions. The observed runtimes have sufficient agreement with the predicted runtimes with the maximum error rate being under 5%.

Table 6.3: Observed and Predicted Values for Gather Component (ms)

Configuration	**Dimensions** ***Y x Y***	**Predicted** **T**gather	**Observed** **T**gather	**Error in** **T**gather **%**
2-Node	8000x8000	9545.912	9535.996	0.10388
	10000x10000	14914.569	14908.224	0.04254
	12000x12000	21476.135	21458.582	0.08173
	13000x13000	25204.27	25183.428	0.08269
4-Node	12000x12000	32371.932	31644.119	2.24828
	14000x14000	43946.388	43098.333	1.92975
	16000x16000	57301.334	56281.039	1.78058
	18200x18200	74048.323	72773.282	1.7219
8-Node	18000x18000	82719.299	86100.994	-4.08816
	20000x20000	102121.151	103856.987	-1.69978
	22000x22000	123565.209	124105.824	-0.43751
	25360x25360	164188.578	166078.437	-1.15103
16-Node	28000x28000	219641.036	228493.122	-4.03025
	30000x30000	252218.437	251974.938	0.09654
	32000x32000	286489.101	287535.079	-0.3651
	35104x35104	344504.887	355330.003	-3.14222

6.1.2.1.3 One Time Host to Device Transfer

The predicted and experimental runtimes for the one time host to device transfers are given in table 6.4 below. The transferred data depends on the problem size *Y*, number of ghost rows *G* and number of processes *P*, therefore, we compare the runtimes for different combinations of these parameters. Equations 5.23 and 5.24 are used for the predictions. We observe higher error rates of up to 17% as the problem size and number of processes increase. However, since these are one time transfers, we do not expect this component to have a significant deteriorating effect on the overall runtime of the application.

Table 6.4: Observed and Predicted Values for One Time Host to Device Transfer (ms)

Configuration	**Dimensions** ***Y x Y***	**Ghost Rows** ***G***	**Predicted** **T**host-device-once	**Observed** **T**host-device-once	**Error in** **T**host-device-once **(%)**
2-Node	8000x8000	10	207.204	185.44	10.50403
	10000x10000	20	325.36	350.781	-7.81303
	12000x12000	40	473.184	522.879	-10.50236
	13000x13000	80	567.426	490.962	13.4756
4-Node	12000x12000	10	256.161	258.577	-0.94325
	14000x14000	20	351.866	363.354	-3.26484
	16000x16000	40	467.218	460.714	1.39212
	18200x18200	80	622.102	604.008	2.90846
8-Node	18000x18000	10	288.892	324.188	-12.21788
	20000x20000	20	361.587	400.953	-10.88699
	22000x22000	40	448.573	518.072	-15.49315
	25360x25360	80	620.061	702.184	-13.24429
16-Node	28000x28000	10	579.562	571.852	1.33032
	30000x30000	20	678.824	581.565	14.32745
	32000x32000	40	799.091	660.539	17.33867
	35104x35104	80	1020.133	880.107	13.72628

6.1.2.1.3 One Time Device to Host Transfer

The following table 6.5 indicates the predicted and experimental runtimes for the one time device to host transfers. In this case too, we vary the problem size *Y*, number of ghost rows *G* and number of processes *P* while comparing the runtimes. We use Equations 5.25 and 5.26 for the predictions. In this case as well, we observe high error rates of up to 16%, but the one time nature of this transaction prohibits it from having a significant impact on the total runtime.

Table 6.5: Observed and Predicted Values for One Time Device to Host Transfer (ms)

Configuration	**Dimensions** ***Y x Y***	**Ghost Rows** ***G***	**Predicted** **T**device-host-once	**Observed** **T**device-host-once	**Error in** **T**device-host-once **(%)**
2-Node	8000x8000	10	203.074	203.127	-0.0259
	10000x10000	20	319.119	317.471	0.51633
	12000x12000	40	464.301	524.535	-12.97298
	13000x13000	80	556.86	553.208	0.65572
4-Node	12000x12000	10	228.901	245.478	-7.24213
	14000x14000	20	314.452	331.902	-5.54911
	16000x16000	40	417.567	462.163	-10.67999
	18200x18200	80	556.018	624.829	-12.37562
8-Node	18000x18000	10	255.139	284.488	-11.5031
	20000x20000	20	319.335	330.402	-3.46582
	22000x22000	40	396.15	421.137	-6.30741
	25360x25360	80	547.588	636.844	-16.29985
16-Node	28000x28000	10	548.755	548.936	-0.03309
	30000x30000	20	642.77	562.293	12.52034
	32000x32000	40	756.68	677.178	10.50668
	35104x35104	80	966.041	845.629	12.46449

6.1.2.2 Iterative Communication Components

In this section, we compare the predicted and experimental runtimes for the iterative components - send-receive, and iterative host-to-device transfers and iterative device-to-host transfers. As discussed, all runtimes are for single communication iteration.

6.1.2.2.1 Send-Receive

Table 6.6 below consists of the predicted and experimental runtimes for the send-receive operation between CPU hosts on different processors. The data transferred depends on the problem size Y and number of ghost rows G; therefore these parameters are varied to compare the runtimes. Equations 5.27, 5.28, 5.29 and 5.30 are used for the

runtime predictions. We observe higher error rates of up to 19% for the 16-node configuration.

The higher errors could be because of missing predictors in the performance modeling process. Predictors related to the number of processes or per process data size or the node interconnect could be considered in the modeling process. Since this component is iterative, greater number of iterations introduces a greater error in the overall runtime. By including the missing predictors, it would be possible to consider higher node configurations such as 32-node or 64-node for the application. Also, we observed that the error rates are higher for the highest problem dimensions for some node configurations. This problem could be addressed by considering a wider range of problem dimensions in the benchmarking process. This would also allow greater problem dimensions to be considered for the application.

Table 6.6: Observed and Predicted Values for Send-Receive Component (ms)

Configuration	**Dimensions** ***Y x Y***	**Ghost Rows** ***G***	**Predicted** **T**send-receive	**Observed** **T**send-receive	**Error in** **T**send-receive **(%)**
2-Node	8000x8000	10	49.929	50.102	-0.346
	10000x10000	20	123.423	126.194	-2.24458
	12000x12000	40	294.85	317.393	-7.64564
	13000x13000	80	637.881	717.44	-12.47237
4-Node	12000x12000	10	87.513	88.409	-1.02434
	14000x14000	20	202.475	178.493	11.84416
	16000x16000	40	461.506	441.056	4.43118
	18200x18200	80	1048.044	1060.406	-1.17952
8-Node	18000x18000	10	139.293	157.788	-13.27828
	20000x20000	20	308.286	292.633	5.07738
	22000x22000	40	677.384	672.84	0.67078
	25360x25360	80	1560.177	1802.143	-15.5089
16-Node	28000x28000	10	173.261	201.54	-16.32199
	30000x30000	20	370.729	434.511	-17.20462
	32000x32000	40	789.72	912.6	-15.55991
	35104x35104	80	1731.644	2047.974	-18.26764

6.1.2.2.1 Iterative Host to Device Transfer

The following table 6.7 consists of the predicted and experimental runtimes for the iterative host to device transfers for a single iteration. The transferred data depends on the problem size Y and number of ghost rows G, so we compare the runtimes for different values of these parameters. We use Equations 5.31 and 5.33 for the predictions. We observe that the observed runtimes are in good agreement with the predicted runtimes with the maximum error being under 15%. The error contributed by this component is also significant since it is an iterative communication and significantly impacts the overall communication runtime.

Table 6.7: Observed and Predicted Values for Iterative Host to Device Transfer (ms)

Dimensions ***Y x Y***	**Ghost Rows** ***G***	**Predicted** **T**host-device-iter	**Observed** **T**host-device-iter	**Error in** **T**host-device-iter **(%)**
8000x8000	10	2.554	2.265	11.31227
10000x10000	20	6.287	6.22	1.05504
12000x12000	40	14.983	15.248	-1.76881
13000x13000	80	32.385	29.511	8.87549
12000x12000	10	3.942	3.642	7.59889
14000x14000	20	9.084	8.77	3.44735
16000x16000	40	20.648	19.645	4.85823
18200x18200	80	46.866	49.391	-5.38808
18000x18000	10	5.807	6.206	-6.86924
20000x20000	20	12.787	13.258	-3.68241
22000x22000	40	28.013	31.196	-11.36194
25360x25360	80	64.457	71.769	-11.34377
28000x28000	10	16.068	14.073	12.4182
30000x30000	20	34.2	29.417	13.98434
32000x32000	40	72.689	62.103	14.56351
35104x35104	80	159.216	137.821	13.43785

6.1.2.2.2 Iterative Device to Host Transfer

Lastly, the table 6.8 below presents the predicted and experimental runtimes for the iterative device to host transfers for a single iteration. In this case too, the parameters *Y* and *G* are varied to compare the runtimes. We use Equations 5.32 and 5.24 for the predictions. The maximum error rate observed is under 12% implying that the predicted and observed runtimes are in tune. This parameter also affects the overall communication runtime due to its iterative nature.

The iterative host to device and device to host components have a small payload. This makes the modeling process complex since it is difficult to predict the behavior and the timings for the transfers of small amounts of data. This may call for a change in the modeling process and include any missing predictors to improve the prediction models.

Table 6.8: Observed and Predicted Values for Iterative Device to Host Transfer (ms)

Dimensions ***Y x Y***	**Ghost Rows** ***G***	**Predicted** **T**device-host-iter	**Observed** **T**device-host-iter	**Error in** **T**device-host-iter **(%)**
8000x8000	10	2.359	2.26	4.23135
10000x10000	20	5.82	5.377	7.61871
12000x12000	40	13.882	13.326	4.00543
13000x13000	80	30.015	30.488	-1.57386
12000x12000	10	3.369	3.3	2.0513
14000x14000	20	7.764	7.296	6.02654
16000x16000	40	17.648	17.623	0.14065
18200x18200	80	40.057	44.81	-11.86755
18000x18000	10	5.008	4.768	4.79523
20000x20000	20	11.033	10.789	2.21236
22000x22000	40	24.175	25.666	-6.16758
25360x25360	80	55.631	61.451	-10.46206
28000x28000	10	13.73	13.383	2.53018
30000x30000	20	29.286	25.881	11.62582
32000x32000	40	62.307	57.695	7.40206
35104x35104	80	136.542	139.134	-1.89849

6.1.3 Total Application Runtime

In this section, we carry out a comparison of the overall predicted and experimentally observed application runtime at large problem sizes. The set number of iterations K is fixed to 400. We compare the overall predicted and experimental computation and communication runtimes and the total application runtime is the sum of these components. Table 6.9 shows the total predicted and observed computation timing over the set number of iterations K. The overall prediction errors are good (under 8%) for medium problem sizes but are as high as 16% for the largest problem size.

Table 6.9: Observed and Predicted Values for Overall Computation Runtimes (ms)

Configuration	**Dimensions** $Y \times Y$	**Ghost Rows** G	**Predicted** $T_{computation}$	**Observed** $T_{computation}$	**Error in** $T_{computation}$ **(%)**
2-Node	8000x8000	10	29034.843	28849.719	0.63759
	10000x10000	20	45648.772	46667.927	-2.2326
	12000x12000	40	66437.66	70645.17	-6.33302
	13000x13000	80	79693.126	92164.315	-15.64901
4-Node	12000x12000	10	32682.005	32724.455	-0.12989
	14000x14000	20	44917.472	46295.484	-3.06788
	16000x16000	40	59667.759	63301.7	-6.09029
	18200x18200	80	79476.638	92583.812	-16.49186
8-Node	18000x18000	10	36665.523	36760.762	-0.25975
	20000x20000	20	45917.403	47242.27	-2.88533
	22000x22000	40	56990.84	61779.433	-8.40239
	25360x25360	80	78827.484	90824.97	-15.21993
16-Node	28000x28000	10	44733.22	44779.879	-0.10431
	30000x30000	20	52436.121	53750.225	-2.5061
	32000x32000	40	61774.496	64895.918	-5.05293
	35104x35104	80	78945.92	89168.992	-12.94946

Table 6.10 consists of the total predicted and observed communication timing over the effective number of iterations K/G. The SIGE model is effective when the sum of individual components is considered to get the overall timing with the highest error rate being under 4%.

Table 6.10: Observed and Predicted Values for Overall Communication Runtimes (ms)

Configuration	**Dimensions** ***Y x Y***	**Ghost Rows** ***G***	**Predicted** **T**communication	**Observed** **T**communication	**Error in** **T**communication **(%)**
2-Node	8000x8000	10	21667.298	21630.36	0.17048
	10000x10000	20	33139.853	33210.133	-0.21207
	12000x12000	40	47063.156	47407.022	-0.73065
	13000x13000	80	54959.467	55401.336	-0.80399
4-Node	12000x12000	10	68280.658	67596.47	1.00202
	14000x14000	20	92050.993	90742.555	1.42143
	16000x16000	40	119414.034	118233.719	0.98842
	18200x18200	80	153656.305	152605.544	0.68384
8-Node	18000x18000	10	172058.961	176131.489	-2.36694
	20000x20000	20	211653.154	212970.337	-0.62233
	22000x22000	40	255376.141	255881.206	-0.19777
	25360x25360	80	338084.472	347101.79	-2.66718
16-Node	28000x28000	10	443021.58	458046.292	-3.39142
	30000x30000	20	508296.447	511593.376	-0.64862
	32000x32000	40	576966.681	583107.668	-1.06436
	35104x35104	80	693187.399	709502.607	-2.35365

Table 6.11 shows the overall predicted and observed application runtimes.

We observe that even though larger errors were observed in the computational components, the runtimes of this component is not sufficiently large to affect the overall runtime. With the highest error rate being under 10%, we can say that the SIGE model has proved to be effective in the runtime predictions of Synchronous Iterative Algorithms (SIAs).

Table 6.11: Observed and Predicted Values for Overall Application Execution Runtimes (ms)

Configuration	**Dimensions** ***Y x Y***	**Ghost Rows** ***G***	**Predicted** **T**execution-time	**Observed** **T**execution-time	**Error in** **T**execution-time **(%)**
2-Node	8000x8000	10	50702.14	50478.709	0.44067
	10000x10000	20	78788.625	79866.989	-1.36868
	12000x12000	40	113500.817	118056.734	-4.014
	13000x13000	80	134652.593	147538.656	-9.56986
4-Node	12000x12000	10	100962.664	100335.029	0.62165
	14000x14000	20	136968.465	137010.42	-0.03063
	16000x16000	40	179081.792	181493.325	-1.34661
	18200x18200	80	233132.944	245141.838	-5.15109
8-Node	18000x18000	10	208724.485	212937.105	-2.01827
	20000x20000	20	257570.557	260291.05	-1.05621
	22000x22000	40	312366.981	317587.063	-1.67114
	25360x25360	80	416911.957	437948.566	-5.04582
16-Node	28000x28000	10	487754.8	502824.433	-3.08959
	30000x30000	20	560732.568	565163.097	-0.79013
	32000x32000	40	638741.176	647816.579	-1.42083
	35104x35104	80	772133.319	798574.374	-3.42442

6.2 Insights.

The SIGE model can be used to draw certain insights on the characteristics of the application. In the following section, we present these insights based on an analysis of the prediction models.

6.2.1 Performance variation

Since the shallow water wave application is a SIA, two important parameters specify the quality of execution of the SIA – the problem size dimensions Y and the number of iterations to be executed K. The SIA might be executed on a wide range of problem sizes depending on the nature of the SIA and the specification of the target user. Similarly, the number of iterations determines the extent to which an end user would

desire the SIA to evolve. An application that evolves slowly might require a large number of iterations. On the other hand, the target user might observe the application behavior after a small number of iterations. With these two parameters in mind, we have the flexibility of choosing the number of hardware resources (or processes, *P*) and the number of ghost rows *G* that the application should use.

The prediction models that we have developed allow us to determine the expected runtimes of the application at various problem sizes and desired number of iterations, but the application performance for different *G* and *P* values is undetermined. At a particular *G* and *P* value, the application might deliver the best performance. For a specified value of the problem size *Y* and number of iterations *K*, we study the predicted runtime and classify the application performance based on the number of processes. We also identify the *G* value at which best performance is observed. Table 6.12 shows this study that is carried out for different values of *Y* and *K*. We consider a sufficiently wide range for the problem size ranging from 112 to 16384 and the *K* parameter is varied from 500 to 8000. Each cell contains three values – the runtime predicted by the SIGE model, the identified *G* value, and a ranking of the number of hardware nodes on the basis of the performance delivered. The node with the smallest runtime is ranked as the best node that can be selected to execute the application using the *G* value specified.

Table 6.12: Execution Time of Application over different *Y*-dimension and *K*-values with a Ranking of Best Performing Node Configuration and Best *G* values.

Y	*K*-value				
	500	1000	2000	4000	8000
112	T = 97.67 *G* = 20 2 > 8 > 4 > 16	T =190.56 *G* = 50 2 > 8 > 4 > 16	T = 377.56 *G* = 20 2 > 8 > 4 > 16	T = 746.45 *G* = 50 2 > 8 > 4 > 16	T = 1488.12 *G* = 100 2 > 8 > 4 > 16
512	T = 474.13 *G* = 20 8 > 2 > 4 > 16	T =804.9 *G* = 20 8 > 2 > 4 > 16	T = 1466.7 *G* = 20 8 > 2 > 4 > 16	T = 2790 *G* = 20 8 > 4 > 16 > 2	T =5437.2 *G* = 20 8 > 16 > 4 > 2
1024	T = 1221.1 *G* = 10 8 > 2 > 4 > 16	T = 1895 *G* = 10 8 > 4 > 2 > 16	T = 3244.7 *G* = 10 8 > 4 = 16 > 4	T = 5942 *G* = 10 8 > 16 > 4 > 2	T = 10528 *G* = 10 16 > 8 > 4 > 2
2048	T = 3726 G = 8 8 > 4 > 2 > 16	T = 5292 G = 8 8 > 16 > 4 > 2	T = 8129 G = 8 16 > 8 > 4 > 2	T = 12843 G = 8 16 > 8 > 4 > 2	T =22091 *G* = 8 16 > 8 > 4 > 2
4096	T = 13366 *G* = 4 8 > 16 > 4 > 2	T =16176 *G* = 4 16 > 8 > 4 > 2	T = 21797 *G* = 4 16 > 8 > 4 > 2	T = 33038 *G* = 4 16 > 8 > 4 > 2	T = 55520 *G* = 4 16 > 8 > 4 > 2
8192	T =46242 *G* = 4 16 > 8 > 4 > 2	T = 54089 *G* = 4 16 > 8 > 4 > 2	T =69782 *G* = 4 16 > 8 > 4 > 2	T = 101167 *G* = 4 16 > 8 > 4 > 2	T =163939 *G* = 4 16 > 8 > 4 > 2
16384	T =174699 *G* = 4 16 > 8 > 4 > 2	T = 199651 *G* = 4 16 > 8 > 4 > 2	T = 249555 *G* = 4 16 > 8 > 4 > 2	T = 349362 *G* = 4 16 > 8 > 4 > 2	T = 548978 *G* = 4 16 > 8 > 4 > 2

We observe that for lowest problem size, the 2-node configuration dominates and *G* values in the range of 20 to 100 are observed. As the problem size increases, higher node configurations begin to dominate and for the largest problem size, the 16-node configurations outperform the rest for a consistent *G* value of 4. For the intermediate problem sizes, the 8-node configuration typically dominates and *G* values in the range 4 to 20 are identified as best to deliver the optimum performance. In this way, table 6.12 provides significant insights to select the number of hardware resources and configure the application (select a *G* value) so that an optimal performance can be achieved. Figure 6.1 provides a general summary in which the number of hardware nodes that are best suited

to deliver the highest performance for a given problem size dimension is indicated. This summary is independent of the number of iterations.

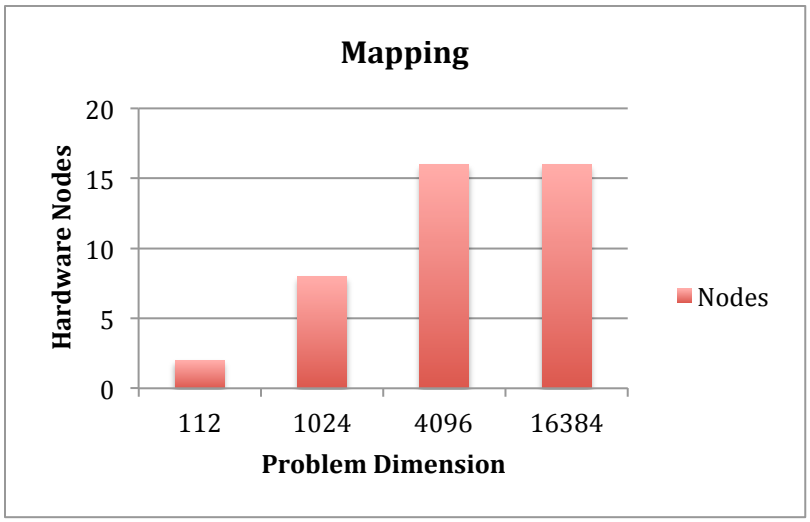
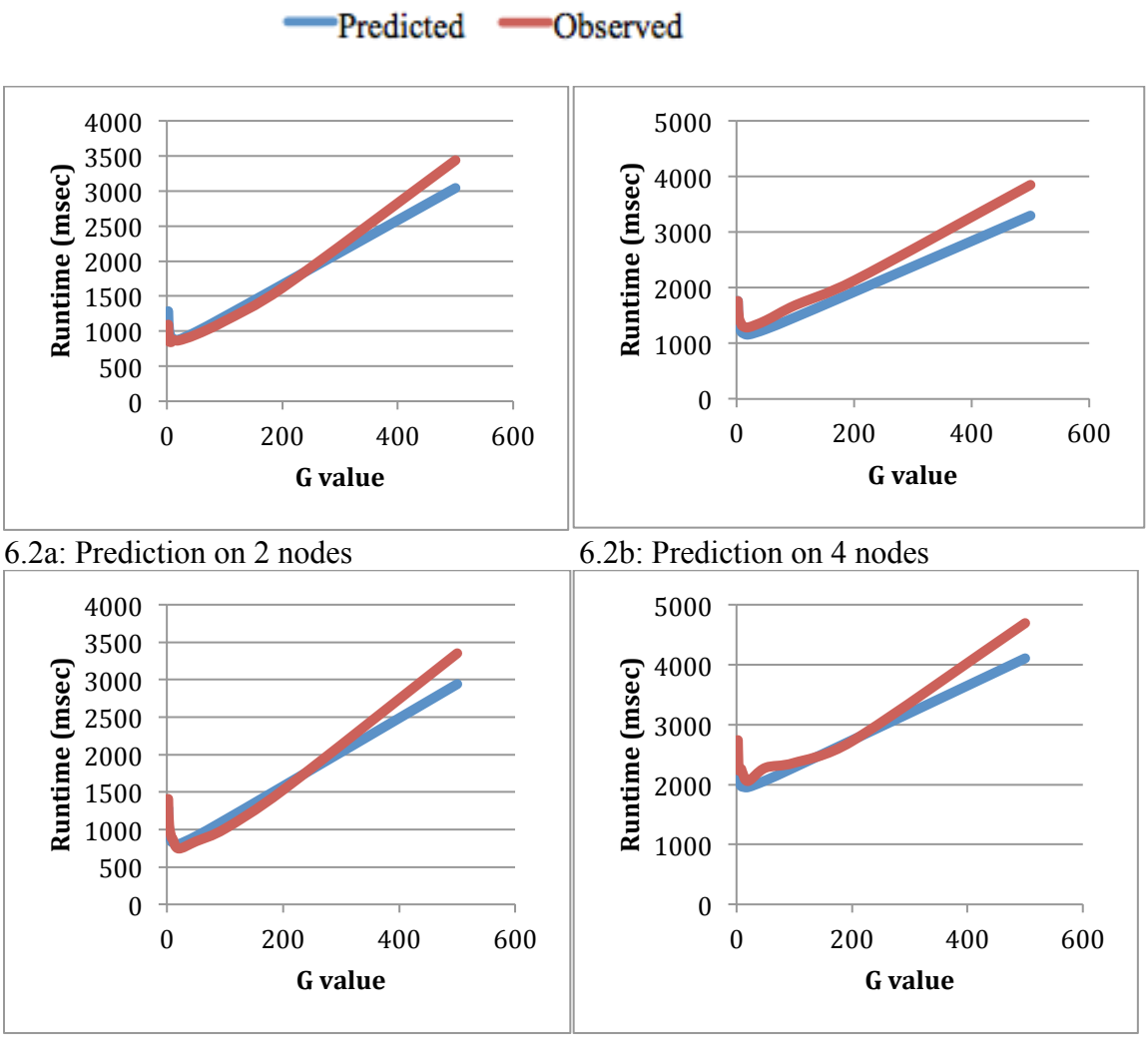


Figure 6.1: Mapping of problem dimensions with number of hardware nodes

6.2.2 Prediction of G value

Since the application has a configurable number of ghost rows G, this parameter could be tuned to provide the best performance for a given problem size, number of iterations, and number of nodes. The runtime predictions provided by the SIGE model can be used to make near accurate prediction regarding which G value will yield the best performance. We consider two such cases as shown in Figure 6.2 and Figure 6.3 with Y = 512, K = 1000 and Y = 4096, K = 2000 respectively. In both cases, we see that for a particular value of G, lowest runtime is achieved. A deviation from this value results in increase in runtime. This is because, the G parameter affects both, the computation and communication components. For the communication component, an increase in G lowers the effective number of iterations for the iterative components, resulting in overall decrease in communication time. But the computation component increases considerably

with an increase in G and dominates the decrease in communication component. Table 6.13 and 6.14 shows the predicted and experimental G values at which best performance is observed for both the cases. We observe that the SIGE model predicts the G value with a good degree of accuracy.



6.2a: Prediction on 2 nodes

6.2b: Prediction on 4 nodes

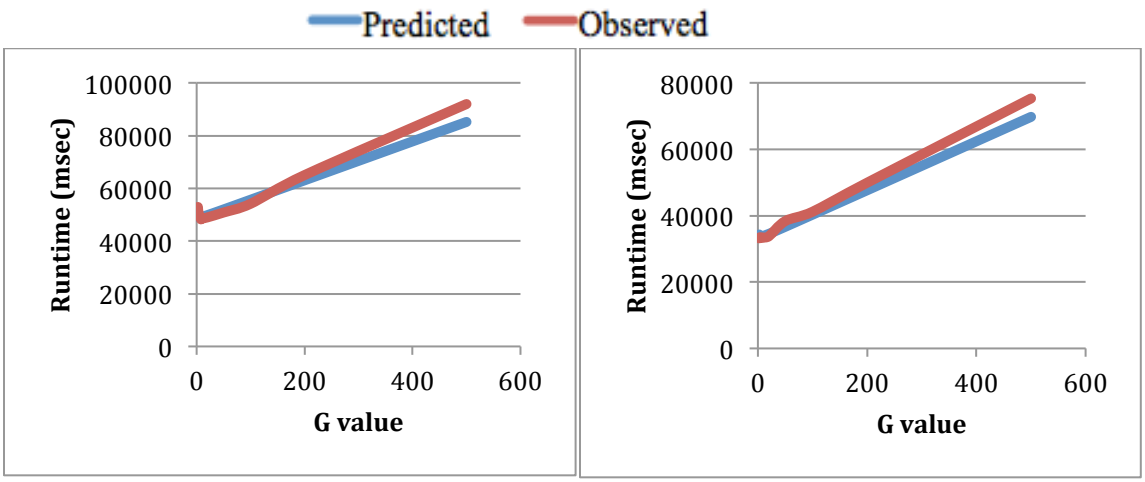
6.2c: Prediction on 8 nodes

6.2d: Prediction on 16 nodes

Figure 6.2: Prediction of G value with $Y = 512$, $K = 1000$ for 2, 4, 8, and 16 nodes

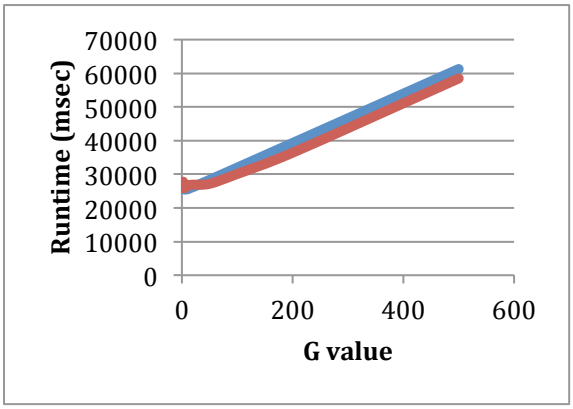
Table 6.13: Observed and Predicted Best *G* Value for *Y* = 512, *K* = 1000.

***Y* = 512, *K* = 1000**				
Configuration	**Runtime (ms)**		***G* value**	
	Predicted	**Observed**	**Predicted**	**Observed**
2-Node	878.176	845.086	8	20
4-Node	1157.536	1288.012	20	20
8-Node	804.914	746.054	20	20
16-Node	1956.521	2059.164	20	20

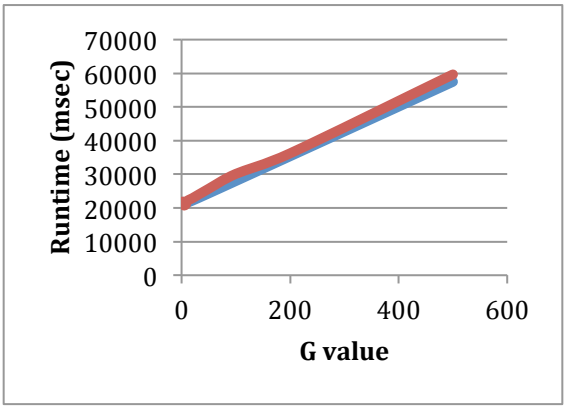


6.3a: Prediction on 2 nodes

6.3b: Prediction on 4 nodes



6.3c: Prediction on 8 nodes



6.3d: Prediction on 16 nodes

Figure 6.3: Prediction of *G* value with *Y* = 4096, *K* = 2000 for 2, 4, 8, and 16 nodes

Table 6.14: Observed and Predicted Best *G* Value for *Y* = 4096, *K* = 2000.

***Y* = 4096, *K* = 2000**				
Configuration	**Runtime (ms)**		***G* value**	
	Predicted	**Observed**	**Predicted**	**Observed**
2-Node	49035.174	48214.763	4	8
4-Node	33867.487	33322.249	8	8
8-Node	25472.338	25898.881	4	4
16-Node	21349.018	20767.353	4	4

6.3 Effects of Variation in Parameters.

The SIGE model is useful in observing the effect of individual parameters on the runtime components. In this section, we explore the effects of the independent parameters on the computation and communication components. In each subsection, we study the effect of one parameter keeping the other parameters constant. We consider only the predicted runtimes for this analysis

6.3.1 Variation in problem size Y

On increasing the *Y* parameter, both, the computation and communication components increase. Table 6.15 shows an instance of varying *Y* for an 8-node configuration with *K* = 1000 and *G* = 20. The computation runtime has a complexity of $O(n^2)$, hence, doubling of the problem size causes the runtime to quadruple. This is confirmed by the predicted runtime. For the communication runtime, the iterative components are doubled as the problem size doubles. However, the scatter and gather components also have a complexity of $O(n^2)$, and therefore, dominate. The net communication component quadruples with some approximation. The predicted communication runtimes verify this.

Table 6.15: Variation in Problem Dimension *Y* on a 8-Node Configuration with *K* = 1000, *G* = 20

***K* = 1000, *G* = 20**		**Runtimes (ms)**	
8-Node configuration	***Y***	**T**computation	**T**communication
	4000	5121.857	11585.937
	8000	19144.703	39492.362
	16000	73980.595	144519.041
	32000	290812.242	551312.851

6.3.2 Variation in number of nodes P

As we increase the number of hardware nodes, the computation runtime decreases whereas the communication runtime increases. Table 6.16 describes the scenario in which the number of nodes are varied for the size *Y* = 8000, *K* =1000 and *G* = 20. The computation data is halved every time the number of nodes are doubled thereby causing the runtime to reduce by half. The computation runtime can be approximated with the following Equation 6.1:

$$T_{compute\text{-}2P} \approx T_{compute\text{-}P} / 2 \quad (6.1)$$

Here, $T_{compute\text{-}2P}$ represents the computation time when the number of processes are doubled from *P* to *2P* and is approximately half of $T_{compute\text{-}P}$ that is the runtime for *P* processes. This can be verified by the predicted computation runtimes. For the communication components, the data transferred is constant for the scatter, gather and iterative components and an increase in number of nodes causes the communication overhead to increase. Hence the communication runtime increases with an increase in nodes. Table 6.16 confirms this observation.

Table 6.16: Variation in Number of Nodes P with $Y = 8000$, $K = 1000$, $G = 20$

$K = 1000$, $G = 20$		Runtimes (ms)	
	Configuration	**T**computation	**T**computation
$Y = 8000$	2-node	73306.668	24903.4115
	4-node	37233.148	35140.3558
	8-node	19144.773	39492.362
	16-node	10201.848	42512.2574

6.3.3 Variation in number of iterations K

The number of iterations has a direct relation to the computation and communication runtimes. We consider a 16-node configuration with $Y = 4000$, $G = 20$ and vary the K parameter in table 6.17 Being an iterative component, the computation runtime is linearly scaled by K and hence, doubles on doubling K. For the communication component, the only the iterative components are scaled by the effective number of iterations K/G and are linearly scaled. Since, the contribution of these components is lesser than the one-time components, a gradual increase in the overall communication runtime should be expected. Table 6.16 confirms both the observations.

Table 6.17: Variation in Set Number of Iterations K with $Y = 4000$, $G = 20$ on a 16-Node Configuration

16-node configuration, $G = 20$		Runtimes (ms)	
	K	**T**computation	**T**computation
$Y = 4000$	500	1434.425	11603.563
	1000	2868.8473	13075.602
	2000	5737.646	16019.481
	4000	11475.3692	21907.138

6.3.4 Variation in the number of ghost rows G

The number of ghost rows determines the iterative communication data and the effective number of iterations. We consider a 2-node configuration with $Y = 4000$, $K = 800$ and vary the G parameter in table 6.18. The computation data size has a very weak

dependence on *G* and therefore, the computation runtime should increase gradually with an increase in *G*. This is confirmed by the predicted computation runtimes. The iterative communication data size has a linear dependence on *G* and it doubles as *G* doubles, but the frequency of these communications is halved on doubling *G*. But these components are shadowed by the scatter and gather operations and the changes on varying *G* are insignificant, as can be seen in table 6.18

Table 6.18: Variation in Number of Ghost Rows *G* with $Y = 4000$, $K = 800$ on a 2-Node Configuration

2-node configuration		**Runtimes (ms)**	
	G	**Tcomputation**	**Tcommunication**
***Y* = 4000** ***K* = 800**	10	14648.302	7098.275
	20	14936.686	7067.962
	40	15511.554	7051.606
	80	16662.339	7047.641

6.4 SWO Analysis of the SIGE model

In this section, we perform a Strength (S), Weakness (W), and Opportunities (O) or SWO analysis of the SIGE model based on the results discussed in section 6.1. The SWO analysis is a subset of the SWOT where T stands for Threats, but the discussion of threats is not applicable for this model.

Strengths – The SIGE model is used to develop equations to predict the computation and communication runtimes of the application. Table 6.11 shows the total application runtime for different configurations and problem sizes with the maximum prediction errors being under 10%. Tables 6.9 and 6.10 provide the overall computation and communication runtimes respectively. The maximum error rates for the computation component is 8% barring a few outlier cases and that for the communication component

is 4%. The SIGE model can provide sufficiently accurate runtime prediction models. An important strength is the ease of use of the model since it makes use of readily available application parameters like data bytes consumed or transferred as predictor variables. It should be noted that although the training data set used a maximum problem dimension of 6000, the results were collected by considering dimensions as large as 35104. Hence, the maximum possible dimensions restricted by only the hardware resources were considered. Therefore, it can be concluded that the SIGE model is effective and provides good prediction results with sufficient accuracy even at the highest problem sizes.

Secondly, the SIGE model enables us to model the runtime for a single iteration of a communication or computation component. With the knowledge of the application, we can obtain the total runtime of any iterative component by scaling it with the number of iterations. This underlines the simplicity of using the SIGE model.

Thirdly, for the computation component, the SIGE model effectively abstracts the number of FLOPS parameter as can be seen from equations 5.1-5.4. The A2A roadmap required the knowledge of the FLOPS capacity of the accelerator but the regression coefficients developed using the training data set doesn't deem this parameter necessary. This enables the prediction mechanism to be truly architecture independent and can be extended to future architectures as well. The only knowledge necessary is of the application data size consumed in the computation. This should be considered as a significant strength of the SIGE model.

Lastly, as discussed in the introduction, the application of interest has sufficient communication complexity because of the exchange of ghost data between CPU host and GPGPU devices and between neighboring CPU nodes at regular iterations. The authors in

[16] and [32] had considered Spiking Neural Network models that did not involve extensive communication operations and were pleasingly parallel. The ability of the SIGE model to provide sufficiently accurate prediction models validates its use for SIAs with sufficient communication complexity. This should also be considered as an important strength of the SIGE model.

Weakness – Tables 6.4, 6.5, 6.6, 6.7 and 6.8 show the one-time and iterative communication runtimes. We observe that the error rates increase as the problem dimensions increase and are as high as 19% in some cases. We also observed a distinct variability in the iterative communication models for the 16-node configurations. The SIGE model is susceptible to parameters such as variability in network protocols and the error rates can be attributed to these missing predictor variables. For very low data transfers (the iterative host-device transfers), the SIGE model predictions show greater errors. Additional predictors could be considered for the send-receive component. The model could consider more accurate methods of modeling such communication transactions. Also, for the chosen application, the scatter and gather operations dominate and overshadow the high error rates in the iterative communication runtimes. If an SIA is chosen in which the iterative components dominate, high error rates in such components may result in higher errors in overall application runtime. The SIGE model should be able to address theses weaknesses.

Opportunities – The SIGE model has potential to improve the communication runtimes explained above by considering additional predictor variables. By improving the performance models for the send-receive component, the application performance at higher configurations such as 32-node and 64-node could be predicted. Further, the SIGE

model can be applied to different SIAs with varying computation and communication complexities for runtime predictions. The model can also be verified on accelerators such as future GPGPUs from NVIDIA and AMD.

6.5 Summary

In this chapter, we presented the predicted and experimentally observed runtimes for the computation and different communication components and the application as a whole for different hardware and problem configurations and compared the same. Thereafter, we used the SIGE model to draw insights on how certain application parameters could be tuned to achieve the best performance. We elaborated that the SIGE model can also be used to predict the value for number of ghost rows to attain the best performance for a particular configuration. The chapter concluded with a SWO analysis of the SIGE model.

CHAPTER 7

CONCLUSION

We conclude the thesis by presenting a summary in section 7.1 and the conclusions and insights in section 7.2. Future challenges are presented in section 7.3.

7.1 Summary

In chapter 1, we discuss the trends in parallel computing and identify certain accelerators for parallel application development. We stress the importance of a roadmap to help map accelerators to application by considering performance parameters such as floating point operations per second and memory access bandwidth. We propose to use the Application to Accelerator (A2A) roadmap [23] to identify an accelerator that is best fit for the concerned application - shallow water wave equations. Secondly, we aim to verify the SIGE model for Synchronous Iterative Algorithms [16] by carrying out a low level abstraction of the chosen accelerator.

In chapter 2, we present the background work in the realm of performance modeling of parallel applications on GPGPUs and discuss certain works in which applications using finite difference method have been implemented on GPGPUs. Although the performance models yield sufficiently high accuracy, most of them require fine knowledge of the accelerator characteristics, making it cumbersome to apply the models directly. Conversely, the SIGE model relies on straightforward application parameters such as the data bytes transferred or consumed in computations. The ease of use coupled with a high accuracy of the model makes it convenient to use it for performance modeling.

In chapter 3, we provide the application and accelerator background. We describe the shallow water wave application. The parallelism of the application lies at the core of the finite difference step that is carried out iteratively. We discuss the available accelerators - homogeneous multiprocessor and heterogeneous CPU- GPGPU clusters. The characteristics of the CUDA architecture and NVIDIA Tesla K20 GPGPU are also discussed.

In chapter 4, we discuss the Application to Accelerator (A2A) roadmap extensively. The roadmap is applied to the application of interest by developing application and accelerator vectors. The heterogeneous CPU-GPGPU cluster is identified as the best-fit accelerator and the results obtained in section 4.3 validate this conclusion. The A2A roadmap is aims in ranking accelerators but does not provide a guarantee regarding relative performance of accelerators. On the recommendation of this roadmap, we choose the heterogeneous CPU-GPGPU cluster for performance modeling.

In chapter 5, we describe the low level abstraction of the SIGE model. We model the computation and communication sections of the applications by performing regression analysis on micro benchmarks. For the communication section, we model the iterative and one-time components separately. The total runtime of the iterative communication components is obtained by scaling it with the effective number of iterations and the total computation runtime is obtained using the set number of iterations.

In chapter 6, we present the experimental validation of the low level abstraction of the SIGE model. We compare the predicted and observed runtimes for the individual communication and computation components and the overall application as a whole. Parameters such as the problem size dimension (Y), number of processes (P), set number

of iterations (K), and number of ghost rows (G) are identified as independent parameters and the prediction models rely on these to obtain the total application runtime. We observed that the maximum error for the computation component was about 8% for the 2, 4 and 8 node configurations. Higher error rates of up to 16% were observed for the 16 node configurations. For the one-time communication components such as the scatter and gather, we achieved sufficiently high accuracy with the maximum error for the scatter and gather operations being under 3% and 5% respectively. The one-time download and read-back transfers between the CPU host-GPGPU device faced higher errors of up to 17%. However, the latencies for these transfers are insignificant when large application sizes are concerned. For the iterative send-receive operations, we observe error rates of up to 15% for 2, 4, 8-node configurations and higher rates of up to 19% for 16-node configurations. For the iterative download and read-back operations, the maximum errors observed are 15% and 12% respectively. On comparing the overall application runtime, we observe that the maximum error rate is under 9%.

We discuss certain insights obtained by the usage of the SIGE model. For the shallow water wave SIA, the runtime prediction model enables us to select the number of hardware nodes and number of ghost rows to execute the application on, for a given problem size dimension and given set number of iterations. This is accomplished by ranking the performance on different nodes and by identifying the adequate value for the number of ghost rows. The runtime model is also able to predict with sufficient accuracy, the G value at which the best performance could be obtained for a fixed problem size dimension and set number of iterations, over different number of nodes. We further discuss the impact of varying the independent parameters on the communication and

computation component runtimes. Chapter 6 is concluded by a discussion of the Strengths, Weaknesses and Opportunities of applying the SIGE model for performance prediction.

7.2 Conclusions

Based on this summary, we draw the following conclusions:

1. The Application to Accelerator roadmap accomplishes the task of ranking potential accelerators and we can select the best-fit accelerators for low-level abstraction and performance modeling. Each accelerator entails an accelerator vector as well as an application vector since the application developed on each accelerator is unique. The A2A roadmap imposes the condition that the user should possess intricate knowledge of the application in order to accurately describe the application vectors. The advantage of this roadmap is that readily available hardware parameters could be used to construct the accelerator vectors.

2. The performance-modeling framework of the SIGE model enables us to predict the application runtimes at larger data configurations within a good degree of accuracy. For a SIA, based on the problem size and number of iterations that are required, we need to select application parameters and accelerators to achieve the best possible performance. Since our application is a SIA, we can further harness the prediction model to identify the application specific number of ghost rows (G value) and the number of hardware nodes (P value) at which the best performance would be delivered for a specific problem size dimension (Y value) and number of iterations (K value). It is also possible to rank the performance across different number of hardware nodes for a specific Y value and K value. Further, the prediction model helps us to tune and identify the value of G at

which best performance could be attained for specific Y value and K value on different number of nodes. Barring a few outlier cases, the predicted G values are in good agreement with the observed values.

3. The performance model enables us to observe the effects of varying each of the independent parameters – Y, P, K and G on the computation and communication runtimes, while keeping the other parameters constant. An increase or decrease in some parameters has different impacts on the communication and computation runtimes. Both runtime components show quadratic increase on increasing the problem size dimension. We observe that the one time communications- scatter and gather, are the most dominant in the communication runtimes and therefore, demand higher accuracy in their prediction models. The SIGE model provides this accuracy. On increasing the number of nodes P, the computation runtime decreases linearly whereas the communication runtime shows a gradual increase due to greater communication overhead. The total computation runtime linearly scales with the number of iterations K but the total communication runtime increases gradually as K is increased. Lastly we observe that the computation runtime increases gradually on increasing the G value due to a weak dependence on the number of ghost rows, whereas the iterative communication runtime decreases as the G value increases. The total communication runtime decreases gradually due to the weak contribution of the iterative components. Tables 6.15, 6.16, 6.17 and 6.17 illustrate the effects of varying the Y, P, K and G parameters.

4. We also conduct a SWO analysis based on the modeling techniques of the SIGE model and the prediction results. Chapter 6.4 describes this analysis in depth. With

this approach, we explore the potential of the SIGE model and its applicability to SIAs on future architectures.

7.3 Future Work.

This thesis aims to verify the Application to Accelerator (A2A) roadmap and the performance prediction framework of the SIGE model on the SIA – shallow water wave equations using finite difference method. We can consider SIAs of varying computation and complexities to perform this verification. For the verification of the A2A roadmap, we can consider additional accelerators such as FPGAs and the Intel Xeon Phi co-processor. We can perform a ranking of the accelerators for the chosen SIA and verify this by comparing small-scale implementations. Additionally, for each accelerator, we can undertake performance modeling using the SIGE model and verification of the same. With this approach, the applicability of the SIGE model can be confirmed across various architectures. The performance modeling could be further extended to larger clusters such as 32-node, 64-node, or even 128-node. However, prior to this, it is essential that the missing predictors discussed in the previous chapter be considered in the performance modeling process. An improvement in the iterative communication components could improve the overall performance prediction at higher node configurations. Further, it is possible to develop insights as described in chapters 6.2 and 6.3 for each accelerator. The SIGE model can also be used for performance prediction on AMD and future NVIDIA GPGPUs. Lastly, we observed certain drawbacks of the SIGE model for the iterative communication runtimes. By considering the suggestions of the SWO analysis, we can enhance the prediction modeling process for these components to yield more accurate prediction models.

REFERENCES

1. Intel Xeon Processor E5-2600 Product Family
   http://download.intel.com/newsroom/kits/xeon/e5/pdfs/Intel_Xeon_E5_Factsheet.pdf

2. NVIDIA CUDA C Programming Guide
   http://docs.nvidia.com/cuda/pdf/CUDA_C_Programming_Guide.pdf

3. NVIDIA's Next Generation CUDA Compute Architecture: Fermi – Whitepaper
   http://www.nvidia.com/content/PDF/fermi_white_papers/NVIDIA_Fermi_Compute_Architecture_Whitepaper.pdf

4. NVIDIA Tesla GPU Accelerators
   http://www.nvidia.com/content/tesla/pdf/NVIDIA-Tesla-Kepler-Family-Datasheet.pdf

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

6. Anthony Danalis, Philip C. Roth, Gabriel Marin, Kyle Spafford, Collin McCurdy, Vinod Tipparaju, Jeremy S. Meredith, Jeffrey S. Vetter. The Scalable HeterOgeneous Computing (SHOC) Benchmark Suite. *GPGPU '10 March 14, 2010. Pittsburgh, PA, USA*

7. Jaewoong Sim, Aniruddha Dasgupta, Hyesoon Kim, Richard Vuduc. A Performance Analysis Framework for Identifying Potential Benefits in GPGPU Applications. *PPoPP'12, February 25–29, 2012, New Orleans, Louisiana, USA.*

8. Rafael Ubal, Dana Schaa, Perhaad Mistry, Xiang Gong, Yash Ukidave, Zhongliang Chen, Gunar Schirner, David Kaeli. Exploring the Heterogeneous Design Space for both Performance and Reliability. *DAC '14, June 01 - 05 2014, San Francisco, CA, USA*

9. Shuai Che, Kevin Skadron. BenchFriend: Correlating the performance of GPU benchmarks. *The International Journal of High Performance Computing Applications 2014, Vol. 28(2) 238–250*

10. Andrew Kerr, Eric Anger, Gilbert Hendry, Sudhakar Yalamanchili. Eiger: A Framework for the Automated Synthesis of Statistical Performance Models.

11. Michael Boyer, Jiayuan Meng, Kalyan Kumaran. Improving GPU Performance Prediction with Data Transfer Modeling. *2013 IEEE 27th International Symposium on Parallel & Distributed Processing Workshops and PhD Forum*

12. Mitesh R. Meswani, Laura Carrington, Didem Unat, Allan Snavely, Scott Baden, Stephen Poole. Modeling and predicting performance of high performance computing applications on hardware accelerators. *The International Journal of High Performance Computing Applications 27(2) 89–108*

13. Cedric Nugteren, Henk Corporaal. The Boat Hull Model: Enabling Performance Prediction for Parallel Computing Prior to Code Development. *CF'12, May 15–17, 2012, Cagliari, Italy.*

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

15. Tiffani L. Williams and Rebecca J. Parsons. The Heterogeneous Bulk Synchronous Parallel Model.

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

17. A. Balevic, L. Rockstroh, A. Tausendfreund, S. Patzelt, G. Goch, S. Simon. Accelerating Simulations of Light Scattering based on Finite-Difference Time-Domain Method with General Purpose GPUs. *2008 11th IEEE International Conference on Computational Science and Engineering*

18. Rached Abdelkhalek, Henri Calandra, Olivier Coulaud, Jean Roman, Guillaume Latu. Fast Seismic Modeling and Reverse Time Migration on a GPU Cluster

19. Diego Brandão, Marcelo Zamith, Esteban Clua, Anselmo Montenegro, André Bulcão, Daniel Madeira, Mauricio Kischinhevsky, Regina C.P. Leal-Toledo. Performance Evaluation of Optimized Implementations of Finite Difference Method for Wave Propagation Problems on GPU Architecture. *2010 22nd International Symposium on Computer Architecture and High Performance Computing Workshops*

20. David Michéa and Dimitri Komatitsch. Accelerating a three-dimensional finite-difference wave propagation code using GPU graphics cards. *Geophys. J. Int.* (2010) 182**,** 389–402

21. Lauri Savioja. Real-time 3D Finite-Difference Time-Domain Simulation of Low- and Mid-Frequency Room Acoustics. *Proceedings of the 13th Int. Conference on Digital Audio Effects (DAFx-10), Graz, Austria , September 6-10, 2010*

22. Lucian Mihai Itu, Constantin Suciu, Florin Moldoveanu, Adrian Postelnicu. GPU Accelerated Simulation of Elliptic Partial Differential Equations. *The 6th IEEE International Conference on Intelligent Data Acquisition and Advanced Computing Systems: Technology and Applications 15-17 September 2011, Prague, Czech Republic*

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

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

25. Chapter 18 – Shallow Water Wave Equation
https://www.mathworks.com/moler/exm/chapters/water.pdf

26. "What makes the Ocean Wave?" Summer seminar ISC5939. John Burkardt, Department of Scientific Computing, Florida State University.
http://people.sc.fsu.edu/~jburkardt/presentations/shallow_water_2012_fsu.pdf

27. The Message Passing Interface (MPI) Standard
http://www.mcs.anl.gov/research/projects/mpi/

28. Intel Xeon Processor E5-2600 Series
http://download.intel.com/support/processors/xeon/sb/xeon_E5-2600.pdf

29. The R Project
http://www.r-project.org/

30. Palmetto Cluster User Guide
http://citi.clemson.edu/palmetto/pages/userguide.html

31. V.K. Pallipuram. Exploring Multiple Levels of Performance Modeling for Heterogeneous Systems. Ph.D. Dissertation, Clemson University, 2013

32. N. Raut. Statistical Regression Methods for GPGPU Design Space Exploration. Master's Thesis, Clemson University, 2013.

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