12-2012

Exploring Computational Chemistry on Emerging Architectures

David Dewayne Jenkins
University of Tennessee - Knoxville, ddj@utk.edu

To the Graduate Council:

I am submitting herewith a dissertation written by David Dewayne Jenkins entitled "Exploring Computational Chemistry on Emerging Architectures." I have examined the final electronic copy of this dissertation for form and content and recommend that it be accepted in partial fulfillment of the requirements for the degree of Doctor of Philosophy, with a major in Computer Engineering.

Gregory D. Peterson, Major Professor

We have read this dissertation and recommend its acceptance:

Robert J. Hinde, Robert J. Harrison, Itamar Arel

Accepted for the Council:

Carolyn R. Hodges

Vice Provost and Dean of the Graduate School

(Original signatures are on file with official student records.)

12-2012

Exploring Computational Chemistry on Emerging Architectures

David Dewayne Jenkins
University of Tennessee - Knoxville, ddj@utk.edu

To the Graduate Council:

I am submitting herewith a dissertation written by David Dewayne Jenkins entitled "Exploring Computational Chemistry on Emerging Architectures." I have examined the final electronic copy of this dissertation for form and content and recommend that it be accepted in partial fulfillment of the requirements for the degree of Doctor of Philosophy, with a major in Computer Engineering.

Gregory D. Peterson, Major Professor

We have read this dissertation and recommend its acceptance:

Robert J. Hinde, Robert J. Harrison, Itamar Arel

Accepted for the Council:
Carolyn R. Hodges

Vice Provost and Dean of the Graduate School

(Original signatures are on file with official student records.)

Exploring Computational Chemistry on Emerging Architectures

A Dissertation

Presented for the

Doctor of Philosophy

Degree

The University of Tennessee, Knoxville

David Dewayne Jenkins

December 2012

This dissertation is dedicated to my beautiful fiancée, Priya, for her love and support.

Acknowledgments

First, I would like to thank my advisors, Dr. Gregory Peterson and Dr. Robert Hinde, for their support and guidance throughout my research. They provided invaluable insight and inspiration that helped shape my research to what it is today. Without their assistance and incessant motivation, there is no telling if I would have ever finished this work.

I would also like to thank Dr. Robert Harrison for serving on my committee and providing some of the computational resources used in this research. In addition, I would like to thank Dr. Itamar Arel for serving on my committee and for allowing me to audit his random processes class so that I could better understand the statistical properties of the application used in this work. I would also like to thank Dr. Harry Richards for his continued financial and educational support.

Without the motivation from friends, I would not have continued to pursue this research. I would like to thank Nicholas Lineback and Dylan Storey for their numerous and long conversations about computers, science, sci-fi movies and TV shows, and everything in between. Their camaraderie has proved to be a helpful release from the trials of research.

Family has played an important role throughout this work. My family put up with listening to me complain about work, not coming home as often as I should, and being a complete zombie when I did come home. Without their understanding and support, there is no telling how my life would have turned out. There is nothing that I could do to ever repay them for their support.

Most of all, I would like to thank my wonderful fiancée, Priya, for standing by me throughout this entire journey with nothing but pure motivation and support. From long talks to making delicious dinners, she has been the cornerstone of my sanity throughout this work. Without her, I would not have continued pursuing my Ph.D. I cannot thank her enough.

Personal funding was supplied by the SCALE-IT fellowship (NSF grant DGE-0801540). Computational resources were supplied through the National Science Foundation grant NSF CHE-0625598.

Everyone you will ever meet knows something you don't. - Bill Nye

Abstract

Emerging architectures, such as next generation microprocessors, graphics processing units, and Intel MIC cards, are being used with increased popularity in high performance computing. Each of these architectures has advantages over previous generations of architectures including performance, programmability, and power efficiency. With the ever-increasing performance of these architectures, scientific computing applications are able to attack larger, more complicated problems. However, since applications perform differently on each of the architectures, it is difficult to determine the best tool for the job. This dissertation makes the following contributions to computer engineering and computational science. First, this work implements the computational chemistry variational path integral application, QSATS, on various architectures, ranging from microprocessors to GPUs to Intel MICs. Second, this work explores the use of analytical performance modeling to predict the runtime and scalability of the application on the architectures. This allows for a comparison of the architectures when determining which to use for a set of program input parameters. The models presented in this dissertation are accurate within 6%. This work combines novel approaches to this algorithm and exploration of the various architectural features to develop the application to perform at its peak. In addition, this expands the understanding of computational science applications and their implementation on emerging architectures while providing insight into the performance, scalability, and programmer productivity.

Contents

List of Tables

List of Figures

Nomenclature

α	Time to execute the VPI algorithm per atom per replica per iteration
β	Percent of times the potential energy of the outer pairs is calculated
$\Delta\tau$	Time Step
ρ	Density
A	Number of Atoms
$AMAT_{lookup}$	Average memory access time for a potential energy lookup
$B_{maxwrite}$	Maximum filesystem write bandwidth
B_{read}	Filesystem read bandwidth
B_{send}	Interprocess communication bandwidth
B_{write}	Filesystem write bandwidth
c_{op}	Number of cycles to execute operation, `op`
D	Speed of writing a file to disk in bytes per second
I	Number of Iterations
L	Interprocess communication latency

M	Number of MPI Masters and work groups
N	Number of MPI Processes
n_tex_pts	Number of points in the texture lookup table
P	Number of Interacting Pairs
P_I	Number of Inner Interacting Pairs
P_O	Number of Outer Interacting Pairs
R	Number of Replicas
R_w	Number of Replicas Per Worker
S	Number of Snapshots
S_I	Number of Iterations Between Snapshots = $\frac{I}{S}$
T	Number of Threads
V	Number of Doubles in a Vector (i.e. 1-No Vectorization, 2-SSE, 4-AVX, 8-MIC)
W	Number of MPI Workers
W_g	Number of MPI Workers in a group
x_{max}	Maximum log of squared distance held in the texture lookup table
x_{min}	Minimum log of squared distance held in the texture lookup table

Chapter 1

Introduction

1.1 Motivation

The National Science Foundation describes computational science as a field of study that is "well recognized by the scientific community as a critical enabling discipline underpinning modern research and development in all fields of science, engineering, and medicine." [1] In addition, the NSF has shown major interest in developing new hardware and cyberinfrastructure to help scientists perform computational work.

The increasing amount of computational resources fuels the growth of potential research possibilities in computational science. The development of computational resources and the benefits they bring spur an increased interest in the type of questions that can be answered with computational sciences. These resources enable research to examine areas of science that were previously too difficult or even impossible to explore experimentally. From microprocessors to graphics processing units (GPUs) to Intel Many Integrated Cores (MIC), it is important to decide which architecture is best for the application for high performance.

Historically, high performance computing systems use tightly connected microprocessors through ultra high speed networks (e.g. Jaguar [2] at Oak Ridge National Laboratory). However, within the past decade, heterogeneous computing has become

an ever-growing trend in computing. From cell processor workstations and clusters [3, 4] to massively parallel co-processors [5] to GPU enabled supercomputers [6, 7, 8], it is safe to say that heterogeneous computing is more than just a trend. It is a revolution in the computing research field. This is due to the potential for extremely high performance (more than 20 petaflops with the upcoming Titan supercomputer at Oak Ridge National Laboratory [7]) and excellent performance to watt ratios.

1.2 Problem

This dissertation explores the use of emerging architectures such as graphics processing units (GPUs) and supercomputers for accelerating computational science applications that use Quantum Monte Carlo methods. Using the power of analytical performance models, this work explores the viability of different architectures for use on this type of problem as well as the scalability of this problem on large computing systems.

In particular, the Quantum Monte Carlo method [9] leads to computational and memory intensive applications depending on various input parameters. However, these applications can be better managed with more advanced computational resources; in fact, it becomes a necessity to exploit these resources. Without these emerging architectures, performing simulations beyond simple toy problems becomes impossible. This is due to the larger number of atoms used, replicas needed, iterations needed, and other physical characteristics of the problem increasing the total amount of work needed to be performed. This work develops analytical performance models to describe the runtime and scalability of an implementation of a variant of a Quantum Monte Carlo application, Variational Path Integral Monte Carlo [10, 11].

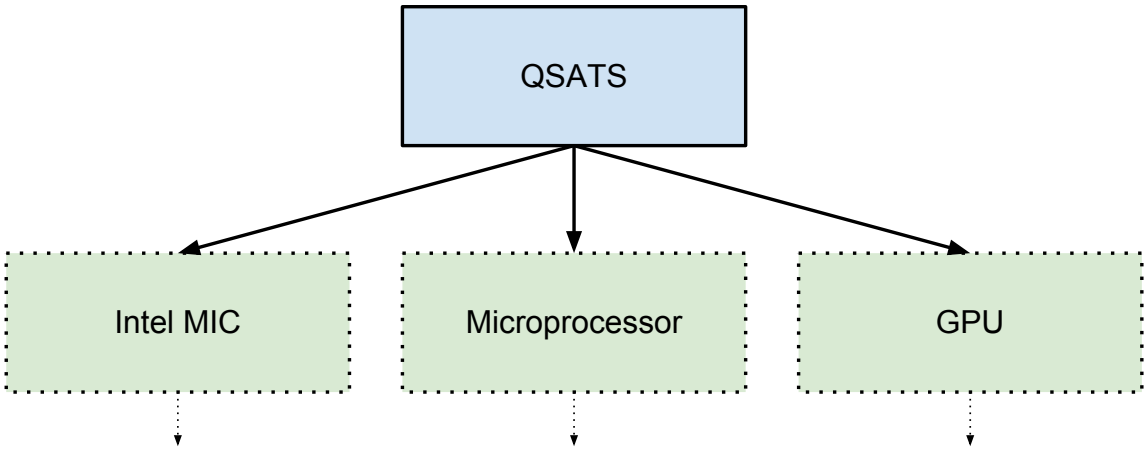


Figure 1.1: Architectures that are explored in this dissertation

1.3 Approach

This dissertation uses the Quantum Monte Carlo simulation method, Variational Path Integral Monte Carlo (named QSATS: Quantum Simulations of Atomic Solids), as a tool to explore the use of the emerging architectures shown in figure 1.1 for computational chemistry applications. This application is based on [12] in that it provides a new MPI implementation to produce a new optimized code. In order to perform this task, this work analyzed and profiled the code to determine bottlenecks and various ways to improve the program. Profiling the code highlighted a number of hot spots in the code that needed to be altered to optimize communications and computations thus increasing performance and scalability of the application. In addition to the MPI implementation, this work also produced a GPU implementation that exploits the highly parallel architecture with fast floating point arithmetic units to produce a code that has significantly higher performance than the microprocessor implementation. The last implementation is for the currently unreleased Intel MIC (many integrated core) using various parallelization techniques (e.g. MPI, threads, vector units).

In addition to the actual implementations, this work also develops analytical performance models that are used to explore different architectures and predict

runtime and scalability of the program. This allows for predicting the performance on future architectures that include different processing and networking possibilities. This also allows for the ability to compare architectures and tradeoffs for using each.

1.4 Contribution

This dissertation has the following contributions to the computer engineering and computational science fields:

- This work develops a highly scalable MPI implementation of a Quantum Monte Carlo method, Variational Path Integral Monte Carlo (VPI), that scales to hundreds of cores with greater than 90% efficiency.
- This work develops a high-performance graphics processing unit implementation of VPI, providing a significant improvement over other architectures and previous implementations.
- A packaged code that is ready for release and easy deployment by other users is developed for this VPI implementation.
- This work is the first to develop performance models for VPI in regards to scalability and runtime on supercomputers and graphics processing units.

1.5 Outline

The rest of this dissertation is as follows:
Chapter 2 provides an overview of the VPI method. It also supplies a survey of work implementing applications onto emerging architectures in the computational science realm. Finally, it provides a survey of performance modeling techniques and research.
Chapter 3 describes the microprocessor and Intel MIC implementations.

Chapter 4 details the graphics processing unit implementation and the optimization steps that this work has taken to produce a near optimal design.

Chapter 5 presents an analysis and comparison of the microprocessor, Intel MIC, and GPU implementations.

Chapter 6 presents the performance models developed for the implementations. This chapter will describe in depth how these models were obtained and their significance to this work.

Chapter 7 concludes this dissertation describing remarks about this work and future work.

Chapter 2

Background and Related Work

This chapter gives an overview of a type of simulation that is commonly used in computational chemistry: Quantum Monte Carlo. The Quantum Monte Carlo simulation method is used extensively in this work as a tool to explore the use of various emerging architectures in the realm of computational sciences. This chapter will also delve into the emerging architectures themselves and their programming and development environments. Also, it will survey related previous work in the areas of computational sciences, computer architectures, and performance modeling.

2.1 Monte Carlo Simulations

In order to find statistical solutions for problems that were intractable to solve analytically or experimentally, Metropolis and Ulam [13] created the Monte Carlo method in 1949. These methods typically generate a Markov chain of configurations (or states) of a physical system, and estimate observable properties by averaging the value of the property over the configurations visited along the Markov chain. As the length of the Markov chain approaches infinity, the estimate will converge to an actual value thus giving a solution to the problem at hand. The transitions from one configuration to the next are calculated with random numbers, causing the need to have high-quality random number generators such as the ones in [14, 15].

There are a number of different types of Monte Carlo methods, but all of them follow this general description. Monte Carlo methods are considered to be one of the 13 "dwarfs" of parallel computing [16, 17] thus showing their importance in the high performance computing realm. Monte Carlo methods are inherently parallel in a number of ways. For instance, when attempting to solve a Monte Carlo problem, it is best to perform many simulations (sometimes called walkers) with independent random number streams. Each of these simulations is independent from the others, thus they are embarrassingly parallel.

2.2 Variational Path Integral Monte Carlo Method

In systems that are composed of small, light atoms, such as helium and neon, quantum mechanical effects are more pronounced due to the atoms' low masses. Sets of atoms form a crystal structure whose quantum-mechanical movements and behaviors are very difficult to predict. The quantum mechanical properties of a many-atom system can, in principle, be determined by solving the Schrödinger equation. The wave function, Ψ, is used in this equation to describe the quantum state of the particles. The wave function is used to give the probability distribution of the location of the particles by equation 2.1

$$P(x) = \Psi^2 \tag{2.1}$$

where x represents the instantaneous configuration of the many-atom system. This wave function is a function of a large number of variables (3A where A is the number of atoms) making it very difficult to solve the Schrödinger equation analytically for Ψ. This leads to the need for statistical analysis such as the variational path integral (VPI) method.

The VPI method (also known as path integral ground state, PIGS, Monte Carlo method [11]) is a quantum Monte Carlo simulation method that allows for the computation of the evolution of replicas (individual "copies" of the structure that

have different configurations of atoms) in a polymer chain that move around in a 3A dimensional space [11, 10, 12]. The VPI method projects the system's ground state wave function out of a suitably chosen trial wave function [12]. The VPI method is an alternative method from the diffusion quantum Monte Carlo (DQMC) method. VPI is preferable because DQMC suffers from complications with fluctuating numbers of walkers and walkers with different levels of importance [18]. VPI uses traditional Metropolis [13, 19] Monte Carlo methods. However, unlike typical Monte Carlo applications, VPI does not use walkers but instead uses a large number of replicas. This is to avoid the complications that are present in the DQMC method [18].

The QSATS VPI implementation follows algorithm 2.1. The physical system's configuration is defined by a set of 3-vectors, one for each atom, that specify the displacement of that atom from its crystal lattice site. These vectors are called displacement vectors below. The VPI system's configuration consists of a set of A displacement vectors for each of R different replicas of the physical system. For each iteration, we loop through all the odd replicas then even replicas. It does not matter whether we do evens or odds first so long as we keep alternating them. These are done due to the independence of the two sets of replicas where the odd-numbered replicas are independent of each other but dependent on the even-numbered replicas and vice-versa. For each replica, we have a series of steps.

First, we calculate the displacements for the new, proposed step. For all replicas except the two "end" replicas, this is done by first averaging the $j-1$ and the $j+1$ replicas as shown in equation 2.2 where $s_{k;j}$ is the displacement vector for the k^{th} atom in the j^{th} replica. This equation does not hold for the end replicas (replicas with $j=1$ and $j=R$). In this case, we set the new displacement vector to $(j \pm 1)^{th}$ for the first and last replicas respectively. Once we have these average displacements, we then add a Gaussian displacement to each component of the vector to randomize the step. The Gaussian displacement is related to a finite-time approximation to the quantum mechanical kinetic energy operator for free particles and follows with the central limit theorem. We use the Box-Muller transformation [20] to generate these

Algorithm 2.1 QSATS algorithm

```
Store the initial configuration, C
I := number of iterations
R := set of replicas
A := number of atoms
for i ∈ I do
  for j ∈ odd R then j ∈ even R do
    for k ∈ A do
      Compute the displacement for the new state
      Calculate the potential energy, dv, for moving to the new state
      if dv < 0 or e^{-dv} > URN then
        Accept move
      else
        Reject move
      end if
    end for
  end for
end for
```

Gaussian random numbers.

$$s^*_{k;j} = \frac{1}{2} * (s_{k;j-1} + s_{k;j+1});\ 1 < j < R \tag{2.2}$$

$$s^*_{k;1} = s_{k;2} \tag{2.3}$$

$$s^*_{k;R} = s_{k;R-1} \tag{2.4}$$

Second, we calculate the potential energy arising from the interactions between the k-th atom and all of the atoms that are nearby. (The number of nearby atoms is specified by the experiment and model in question, i.e., 56 in the case of this work.) This calculation is performed twice, once for the old position of the k-th atom, and once for the new position of the k-th atom. For the potential energy function, QSATS uses the HFDB-He potential [21] as shown in equation 2.5. The parameters for this calculation are given in table 2.1.

Table 2.1: Parameters for the HFDB-He calculations in equation 2.5.

Parameter	Value
A	1.8443101e5
α	10.43329537
β	-2.27965105
D	1.4826
c_6	1.36745214
c_8	0.42123807
c_{10}	0.17473318
r_m	5.59926
ϵ	10.948

$$V_{HFDB} = \epsilon \left(A * e^{-\alpha x + \beta x^2} - \left(\frac{c_6}{x^6} + \frac{c_8}{x^8} + \frac{c_{10}}{x^{10}} \right) * \left(\begin{cases} e^{-\left(\frac{D}{x}-1\right)^2}, & \text{if } x < D \\ 1, & \text{if } x \geq D \end{cases} \right) \right) \tag{2.5}$$

where

$$x = \frac{r}{r_m} \tag{2.6}$$

Once the potential energies associated with the old and new positions are calculated, we subtract the two to get the difference in potential energies, dv.

Third, we have to decide to accept or reject this move. If the dv is less than or equal to zero, we will accept the move. If dv is greater than zero, we will generate a uniform random number. If this random number is less than the exponential of $-dv$, we accept the move. More precisely, the move is accepted with a probability of

$$exp(\frac{-\Delta t \Delta V}{\hbar}) \tag{2.7}$$

Otherwise, the algorithm rejects the move and the atom stays in the same place until the next iteration. This whole process is repeated until the specified number of iterations is complete.

Throughout the simulation, the QSATS program takes snapshots of some of the replicas in order to calculate the crystal's ground state energy later with another program.

From a science perspective, being able to execute VPI on a large crystal structure is important for a few reasons. The VPI simulations tell us the total ground state (zero-temperature) energy of the crystal as a function of its density. This information can be used to derive the zero-temperature equation of state for the crystal, which tells us the pressure needed to compress the crystal to a particular density. Both the ground state energy and the equation of state can be compared to experimental results. This will show that the simulations are correct and coincide with the experimental results. If the simulations are deemed worthy enough to be equivalent, they can be used to explore experiments that cannot physically be performed, such as ones that involve extremely high pressures.

For this work, the algorithm only focuses on two-body interactions. However, using three-body interactions is an interesting extension to this code, as it would provide a more detailed and accurate description of the actual interactions among the atoms. In addition, this work only focuses on simulating the ^{4}He atom structures. However, the VPI method is also applicable to other types of crystals such as solid para-H_2 and solid Ne.

A number of papers have been published on the use of VPI such as [22, 23]; however, none of these look at accelerating the process with emerging architectures. Hinde [12] previously wrote QSATS using Fortran and MPI to work across many compute nodes. Mudhasani [24] and Kakani [25] developed the VPI simulation for the GPU; however, this dissertation provides a different, higher performance approach, which will be explained later.

2.3 Survey of Architectures

This subsection gives an overview of the different types of architectures available for exploration for scientific applications. There are a number of benefits to each architecture as well as downfalls. Each will be described here.

2.3.1 Microprocessors

Traditionally, microprocessors have been the "go-to" architecture for any application, whether it is everyday applications or high performance computing. Due to its history, the microprocessor has a long line of support when it comes to languages, compilers, and users. Microprocessors not only give a high level of support for users, but they also have the potential to provide architectures for high performance computing applications. With vector units such as SSE and AVX that allow for 128 and 256 bit wide units, it is possible to achieve a 4x [26] or 8x [27] speedup in single precision over non-vectorized codes. All of this speedup comes with a minimal power gain [27]. For multicore, single node codes, applications that exploit the shared memory model typically use threads such as pthreads [28] or OpenMP [29]. It is even possible to use portable languages such as OpenCL [30] that can work many different types of microprocessors and GPUs. Libraries such as MPI [31] are available to provide communication capabilities between processes across many nodes. MPI is the most common and accepted way to perform high performance computing on clusters and large supercomputers.

Microprocessors have been a major contender for computational science applications. Chemistry applications such as GAMESS [32] and NWChem [33] have been parallelized using threaded and MPI techniques. Biology applications such as BLAST [34], HMMER [35], and NAMD [36] have been parallelized to thousands of cores. These are just a few examples, but they give a glimpse into the way computational science can take advantage of highly parallel microprocessor systems.

2.3.2 GPUs

Developers traditionally used graphics processing units (GPUs) for 3D graphics rendering for video games. As video gamers began to demand better graphics and high resolutions, GPU developers had to fight to meet demand. This led to the development of many architectural design improvements such as high core counts at lower clock frequencies. These highly parallel devices were soon adopted by developers outside of the gaming industry as they saw the potential to accelerate other applications, specifically scientific computing [37]. Currently, these GPUs support IEEE-compliant single- and double-precision arithmetic, making them acceptable for a variety of computational science codes. The main GPU companies at the time of writing are NVIDIA and AMD/ATI, both of which have their advantages in terms of performance and programmability. NVIDIA developed their proprietary programming language, Compute Unified Device Architecture (CUDA) [38] as an extension to C/C++ that allows for threaded programming on a large set of NVIDIA GPUs. AMD/ATI also had their own language called Brook+ [39]; however, they later decided to put full effort into assisting Khronos in developing an open language to work across many different devices called OpenCL [30].

These languages are centered around a Single Program, Multiple Data (SPMD) programming model. In particular, CUDA has a model that consists of a computation grid that has a set of blocks (maximum of 65535 at the time of writing), each containing a set of threads (maximum of 512 or 1024 depending on the device)[38]. These blocks get assigned to the stream multiprocessors where each thread works in a Single Instruction, Multiple Thread (SIMT) fashion where multiple threads execute the same instruction at the same time. The traditional CUDA model requires explicit data transfers from the host to the device using the runtime library. However, with newer architectures, CUDA allows for a unified address space so that data does not need to be explicitly copied to the device[38].

Developers have ported a large number of applications to the GPU ranging from computational biology [40, 41, 42, 43], chemistry [33, 44, 45], and mathematics [46, 47, 48] to even everyday applications such as Matlab and Mathematica [49]. In addition, researchers have built large machines that depend on GPUs to run these types of applications [6, 50].

2.3.3 Intel MIC

In the near future, Intel will release their many integrated core (MIC) architecture, a new form of general-purpose processor. Built off the 22nm manufacturing process, the MIC will scale to more than 50 cores [5]. This next generation of co-processors is able to achieve more than one teraflop of performance while computing a DGEMM (double precision general matrix multiplication) as demonstrated at Supercomputing 2011 in Seattle, WA[51]. Intel also announced that the MIC architecture will have a 512 bit wide vector unit that will help exploit parallelism even more [52]. The lack of information in this section is due to the nondisclosure agreement that surrounds my interaction with the product. All information is freely available by doing a simple search online [53, 52].

There have been a few applications that have been developed for the Intel MIC such as a protein folding application, molecular dynamics, SGEMM, and LU factorization [54]. Also, at the Supercomputing 2011 Oak Ridge National Lab booth, other MIC applications were presented such as NWChem, Enzo, and some computational fluid dynamics codes.

2.4 Performance Modeling

Researchers use performance modeling for a number of reasons. One reason is to determine how to best map the application to a set of resources. Hu and Gordon [55] describe three different modeling techniques: measurement, simulation, and analytical

modeling. Measurement is the obtaining of the performance on the machine in question by doing many measurements. The machine must be present in order to perform this type of modeling; however, it can be simple to do. Hu and Gordon also note that this cannot be used for performance prediction or applied to other systems. Due to background noise and other possible hindrances, the output parameters must be chosen carefully and gathered using statistical methods in order to draw meaningful conclusions [56]. Measurements can be expensive and have large workloads.

Simulation, on the other hand, allows one to have a computerized model of the desired system that mimics the behavior and functionality. Because it is a computerized model, it does not depend on the system to be available for testing thus simulations methods are capable of simulating any system possible, giving generality and flexibility [56]. However, for large complex designs, simulating an application may not be reasonable as simulations can take a long time to complete. Simulations are common when designing an FPGA circuit. Making simulation models more difficult is the need for validation that the simulation is correct. The simulation only includes the factors and effects that are included in the simulator. As with measurement modeling, there also needs to be a set of representative tests that show the full behavior of the model.

Finally, analytical models allow for a mathematical model to describe the performance of the application without simulations or access to the desired machine. The power of analytical models comes at the price of being difficult to develop or solve when trying to incorporate a large amount of detail. Smith [57] details a few advantages to analytical models:

- Insight into the inner workings of the system
- Accurate results even for basic models
- Better predictive value from results
- Ability to use differentiation to optimize various aspects of the application

There have been many examples that have used performance models to describe their application performance. Peterson [58] used performance modeling for implementations of synchronous-iterative algorithms, a type of fork-join algorithm, which includes application and background load imbalance. Smith and Peterson [59] extended this work to use this model to determine a way to best distribute a parallel application to a set of resources using FPGAs in a near optimal way. Hong and Kim [60] use analytical models to describe the deep inner workings of a GPU all the way down to the register level. This work shed light on the underlying architecture of the GPU so that developers can better understand how to tune their codes; however, this does not account for newer GPU architectures that contain global memory caching. Sim et al. [61] does consider the cache parameters and discuss their tools to extract various parameters of a CUDA program. However, at the time of writing, their static analysis and instruction analyzer tools are not available to the public so this modeling technique is not used in this dissertation. Volkov and Demmel [46] performed measurement based modeling to tune a high performance DGEMM for GPUs. Gothandaraman [62] used performance models to describe her implementations of her Quantum Monte Carlo application on FPGAs, microprocessors, and GPUs. Ipek et al. [63] explored the use of neural networks on a set of input program and platform input parameters to develop a model that described the performance of their application. Baghsorkhi et al. [64] discussed the development of memory hierarchy models using stochastic Monte Carlo techniques.

2.5 Architectures for Monte Carlo

Due to a wide variety of methods and applications, Monte Carlo (MC) methods have been implemented in numerous ways on various architectures. Researchers have extensively implemented MC methods onto microprocessor machines, clusters, and supercomputers. Abdikamalov et al. [65] explores modeling neutrino transport in core-collapse supernova in an astrophysics MC application on a computer cluster using

thousands of cores using OpenMP and MPI. Moral et al. [66] uses microprocessors to perform MC simulations for approximate Bayesian computations. Jenkins and Peterson [41] use tens of microprocessors using MPI to implement Gillespie's algorithm, a Monte Carlo method to simulate the time evolution of chemically reacting species [67]. CASINO is a Quantum Monte Carlo simulation software package that uses MPI to obtain parallelism on parallel machines [68]. Most related to this dissertation, Hinde [12] used tens of processors to implement the Variational Path Integral Quantum MC method to simulate the atom interactions in a helium crystal structure.

Monte Carlo methods are not restricted to microprocessor implementations. Many researchers have explored the use of graphics processing units due to their highly parallel capabilities and the embarrassingly parallel nature of Monte Carlo methods. Li and Petzold [69] and Jenkins and Peterson [41, 70, 71] implemented Gillespie's method on CUDA-enabled GPUs achieving significant speedups. Heymann and Siebenmorgen [72] use GPUs to perform MC simulations of dust radiative transfer applied to active galactic nuclei in a computational astrophysics application. Anderson et al. [73] provide a GPU implementation of QMcBeaver to perform Quantum Monte Carlo implementations. Gothandaraman [62] uses FPGAs and CUDA-enabled NVIDIA GPUs for Quantum MC simulations. Related to this dissertation, Mudhasani [24] and Kakani [25] developed the VPI simulation for the GPU with moderate performance gains. This dissertation uses a different, high performance approach that will be discussed later.

Implementations of Monte Carlo applications are also available for reconfigurable computing architectures (not explored in this dissertation). Reconfigurable architectures (such as field programmable gate arrays [FPGAs]) have been used for radiative heat transfer [74], pricing derivatives of interest rate models [75], mixed precision studies of MC methods [76], and Quantum MC simulations [45, 62].

This list is by no means exhaustive but is representative of the different architectures that have been used for MC simulations.

2.5.1 Random Number Generation for MC

Since Monte Carlo simulations require a large number of random numbers, statistically high quality random number generators are needed to produce uncorrelated results. There are many random number generators available that provide for high quality numbers. The most notable software package that supplies a variety of random number generators is SPRNG [15]. Random number generators that are provided by the package include a Multiplicative Lagged Fibonacci Generator, Combined Multiple Recursive Generator, Modified Lagged Fibonacci Generator, and 48-bit and 64-bit Linear Congruential Generator. Each of these have varying periods and statistics that are good for different applications making it ideal for MC methods. Previous research provides FPGA and GPU implementations (called HASPRNG [77] and GASPRNG [78]) that are helpful for developing MC applications on accelerators. There are a couple of random number generation algorithms developed for the GPU such as GASPRNG [78] and CURAND [79]. For this work, a custom Combined Multiple Recursive Generator [14] is used due to its customizability and ease of use.

Chapter 3

Microprocessors and Intel MIC

This chapter provides an overview of the microprocessor and Intel MIC implementations of the Variational Path Integral Quantum Monte Carlo method, including the serial, threaded, and MPI implementations for both architectures. This work uses this algorithm to simulate the movement and interaction of helium atoms in a crystal structure as described in section 2.2.

3.1 Basic Algorithm overview

This implementation follows the algorithm listed in 2.1 for the QSATS VPI implementation. Essentially, there are 6 main steps:

- Initialization (1)
- For each iteration, replica, and atom
  - Calculate a Gaussian random displacement (2)
  - Calculate potential energy for moving to new state (3)
  - Accept or reject the move (4)
- Occasionally, take a snapshot of the current state (5)
- Finalization (6)

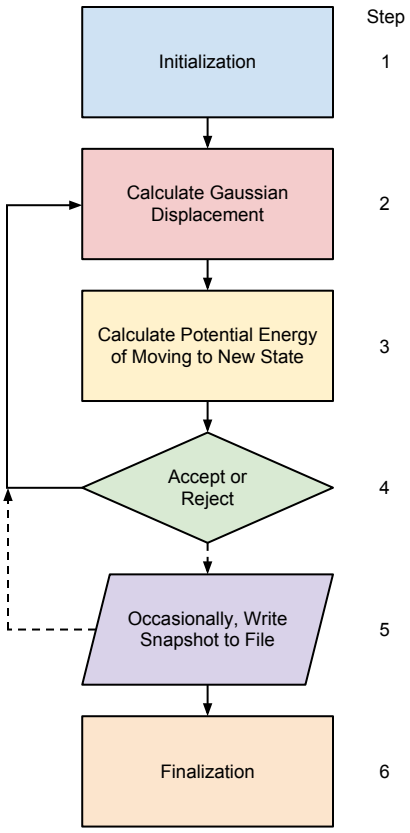


Figure 3.1: General flowchart of the VPI method

These steps can be visualized in figure 3.1 and each step will be referred to throughout this chapter by the color, name, and step number of the box.

As mentioned in 2.2, this implementation seeks to expand and optimize the previous QSATS implementation developed by Hinde [12]. First, a serialized CPU implementation is written to test various algorithmic and memory optimizations. Second, threaded versions using OpenMP and Pthreads is written to compare the performance on a shared memory system. Third, this work developed several MPI implementations to compare the scalability on various systems. For each of these implementations, this work also provides vectorized SSE and AVX codes developed using intrinsics.

Throughout the rest of this dissertation, any references to the implementations will use the term QSATS since this work is a new version of the original. For any

notation not described here, refer to the Nomenclature section on page xiv at the beginning of this dissertation.

3.2 Code Overview

This section provides an overview of the code. For more discussions about the background of the algorithm, refer to section 2.2.

3.2.1 Step 1: Initialization

The initialization step (the blue box in figure 3.1) first reads in the crystal lattice structure file. This file consists of a total of A+2 lines. The first line specifies the number of atoms in the simulation box, A. This number is here to ensure that the rest of the file is not malformed. The second line gives the lengths of the edges, in Bohr radius units, a_0, of the simulation box in the x, y, and z directions. QSATS uses this number to scale the coordinates of the lattice sites and the edges of the box to create the crystal in the correct atomic density given at run time. The remaining N lines are the (x, y, z) coordinates of the lattice site positions.

Next, QSATS allocates the the storage space for the displacement vector (variable: path), random number generator vector (variable: rstatv), and the potential energy look-up table (variable: v). The double precision path variable is used to hold the displacements of each atom in each replica from their "ideal" position (i.e., a displacement of 0 means the atom is in its desired position) and is allocated to 3*A*R double precision elements (3 for x,y,z coordinates and R to hold the displacements for each replica). At this step, if a save file is specified to continue a previous run, it is read and stored into this variable. Otherwise, path variables are initialized to their ideal position, 0.

The rstatv variable holds the random number generator (RNG) state values for all of the replicas so it is allocated as 8*R. Each replica has 8 RNG state variables: 6

for the uniform RNG and 2 for the Gaussian RNG. We have to first seed the RNG by inputting values into each of the 6 uniform RNG state variables. For the first replica, all 6 values are set to 12345. For the remaining replicas, the seeds need to be chosen in such a way to avoid overlapping random number streams. To do this, QSATS uses the "jump-ahead" matrices given in [80]. The other two values in the state vector are used for the Gaussian RNG. Since the Box-Muller transformation generates two Gaussian random numbers at a time, one is returned from the function while one is stored in one of the extra spaces in the state vector. The last free spot in the state vector is a flag stating whether to use the stored number or not.

Next, the v vector has 20,000*2 elements allocated. This vector holds a look-up table for the potential energy function in equation 2.5 for a given $distance^2$ value. There are a total of 20,000 bins, each containing two values that are used for the linear interpolation of the function. The first value of which is the y-intercept and the second is the slope. Figure 3.2 shows the interpolated table overlaid with the actual computed function. The lookup tables are discussed in more detail in section 3.2.3.

Finally, QSATS generates a list of interacting atoms and their separation vectors for each atom. The list is sorted in order of their ideal separation distances in ascending order. The reason for this will be described later as part of the optimizations in section 3.3.1.

Once the initialization phase is complete, the simulation can begin. This includes the red, yellow, green, and purple boxes (steps 2-5) in figure 3.1.

3.2.2 Step 2: Gaussian Displacements

The first step in the simulation is to calculate the displacement that each atom will potentially move (the red box in figure 3.1). For each atom, a, in the replica, r,

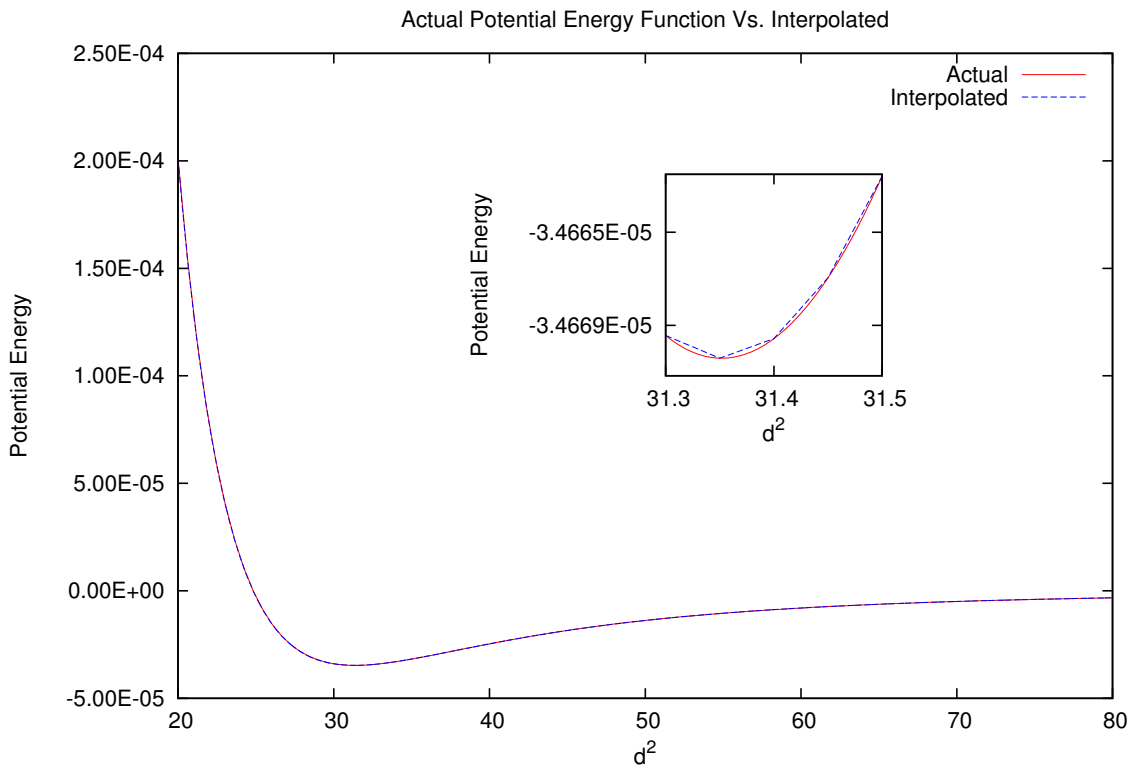


Figure 3.2: Actual potential energy function versus interpolated values

QSATS will calculate the average displacement of the neighboring replicas. That is,

$$new\_path(r, a_{\{x,y,z\}}) = \frac{1}{2} * \big(path(r-1, a_{\{x,y,z\}}) + path(r+1, a_{\{x,y,z\}})\big) \tag{3.1}$$

For the edge cases where r=0 or r=R-1, the displacement of the atom in the neighboring replica is used (i.e., path(1, $a_{\{x,y,z\}}$) and path(R-2, $a_{\{x,y,z\}}$), respectively).

Next, QSATS adds the Gaussian displacement to each component of averaged move vector to randomize the step. The Gaussian displacement is related to a finite-time approximation to the quantum mechanical kinetic energy operator for free particles and follows with the central limit theorem. To generate the Gaussian random numbers, QSATS uses the Box-Muller transformation [20].

Calculating the average and Gaussian displacements is an $\mathcal{O}(2 * 3 * A) = \mathcal{O}(A)$ operation. It is possible to do both steps in one pass, however, the compiler is able to better auto-vectorize the average and Gaussian generation calculations when the loops are separated thus increasing performance. The 3 factor comes from having to perform each operation three times, one for each of the vector components in the x, y, and z directions.

3.2.3 Step 3: Potential Energy Calculations

Calculating the difference in potential energy to move the atom is the most time consuming part of the entire program. The potential energy that QSATS is calculating is the difference in potential energy, dv, of the old, v_{old}, and new, v_{new}, location, as shown in equation 3.2.

$$dv = v_{old} - v_{new} \tag{3.2}$$

To calculate each of the potential energies, QSATS first calculates the difference between the two locations and all of the atom's closest neighbors. The number of neighbors, P, is a configurable option in the program that is dependent on the experiment at hand. In the case of this dissertation, this number is 56. Equation 3.3 shows the calculation of these squared distance, $d^2_{\{old|new\}}$, values, where r is the current replica, $a_{\{old|new\},\{x,y,z\}}$ is the current atom's old/new x, y, z displacement, and $a_{i,\{x,y,z\}}$ is the i^{th} neighbor's displacement.

$$\begin{aligned} d^2_{\{old|new\},i} &= (a_{\{old|new\},x} - a_{i,x})^2 \\ &+ (a_{\{old|new\},y} - a_{i,y})^2 \\ &+ (a_{\{old|new\},z} - a_{i,z})^2 \\ &i \in Neighbors \end{aligned} \tag{3.3}$$

Once all the squared distances are calculated, QSATS uses them to compute the potential energy by doing a table look-up on each one as shown in equations 3.4 and 3.5 and summing them together as in equation 3.6. The bin width is set to .05.

$$lookup(d2) = v[bin * 2] + v[bin * 2 + 1] * d2 \tag{3.4}$$

$$bin = \frac{d2}{bin\_width} \tag{3.5}$$

$$v_{old|new} = \sum_{i=1}^{P} lookup(d^2_{\{old|new\}i}) \tag{3.6}$$

The difference of potential energies, dv, is now calculated using 3.2. This is where sorting the neighbors in idealized distance order comes into play. Because the calculated squared distances will be roughly sorted in increasing order, accesses to the potential energy table will be within a similar range thus increasing cache reuse.

Calculating the difference in potential energy is $\mathcal{O}(2P) = \mathcal{O}(P)$. The 2P represents two passes over the data: one to calculate the squared distances and one to calculate the potential energies. This is to facilitate optimizations that will be discussed later.

3.2.4 Step 4: Accept or Reject a Move

Now that the difference in potential energy of the old and new positions is calculated, it is time to decide whether to accept or reject the move. This is done based on the graph shown in figure 3.3.

The first check QSATS does is to see if the new location is an "uphill" move. This means if dv is less than or equal to zero, we will accept the move. If dv is greater than zero, we sample a uniform random number. If the exponential of -dv is greater than the uniform random number, then we again accept the move. In other words, the move is accepted with a probability of

$$exp(\frac{-\Delta t \Delta V}{\hbar}) \tag{3.7}$$

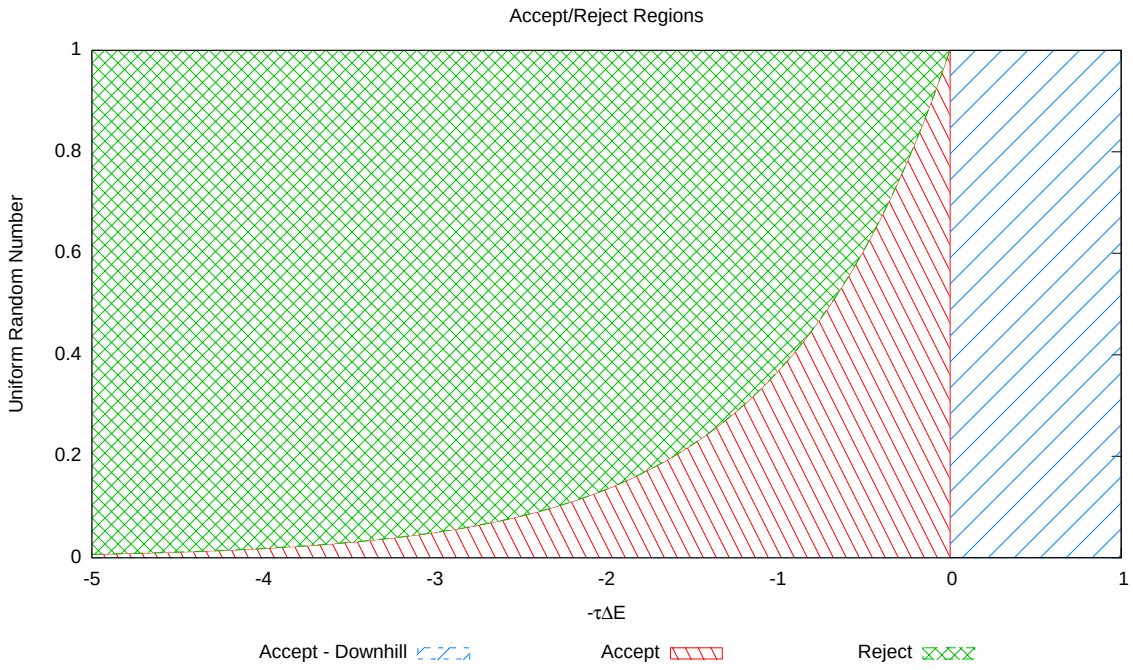


Figure 3.3: Accept/reject regions

Otherwise, the move is rejected and the atom remains in the the same location.

Sections 3.2.3 through 3.2.4 are repeated A times and sections 3.2.2 through 3.2.4 are repeated R*I times. That is, each atom in each replica attempts to move for the entirety of the simulation (i.e., I iterations).

3.2.5 Step 5: Snapshotting

Every S_I iterations, QSATS will take a snapshot of the states of every 11^{th} replica and write all the atom coordinates to a file. These snapshots are used later by an auxiliary program to analyze the crystal's ground state energy.

3.2.6 Step 6: Finalization

Lastly, QSATS will write a file that saves the last displacement state and RNG state information of all of the replicas in the simulation. This gives the user the ability to continue a run from the current state instead of starting from the idealized positions.

Once the save file is written, all allocated memory is freed including the displacement, RNG states, and potential energy lookup table.

3.3 Microprocessor Implementation

3.3.1 Serial

Before attacking the parallel implementations, the first step is to implement the algorithm serially to ensure correctness of the algorithm and explore different core level optimizations. The original QSATS code was written in Fortran, so it was converted to C for ease of implementing SSE and AVX versions (discussed in section 3.3.2).

The serial implementation goes as follows. For each of the odd numbered replicas, potential new atom locations are calculated. All of the odd replicas are independent of each other, but rely on the even replicas to aid in the calculation of the displacement. This repeats over all the even replicas. As with the odd replicas, the even replicas are also independent of each other but are dependent on the odd replicas. If it is the S_I^{th} iteration, a snapshot is written to a file. This is repeated for I iterations serially making this an $\mathcal{O}(I * R * (2 * A + A * 2 * P)) = \mathcal{O}(I * R * A * (1 + P))$ algorithm. Before moving on to parallelizing the program, the serial version must be optimized.

Figure 3.4 shows the breakdown of function contributions to the overall runtime of the unoptimized C code for a variety of parameters as obtained from gprof. From here, we can see that the most expensive portions of the code come from looking up the potential energy and calculating the squared distances. These are prime areas for performance improvements since the majority of the runtime is consumed in these portions.

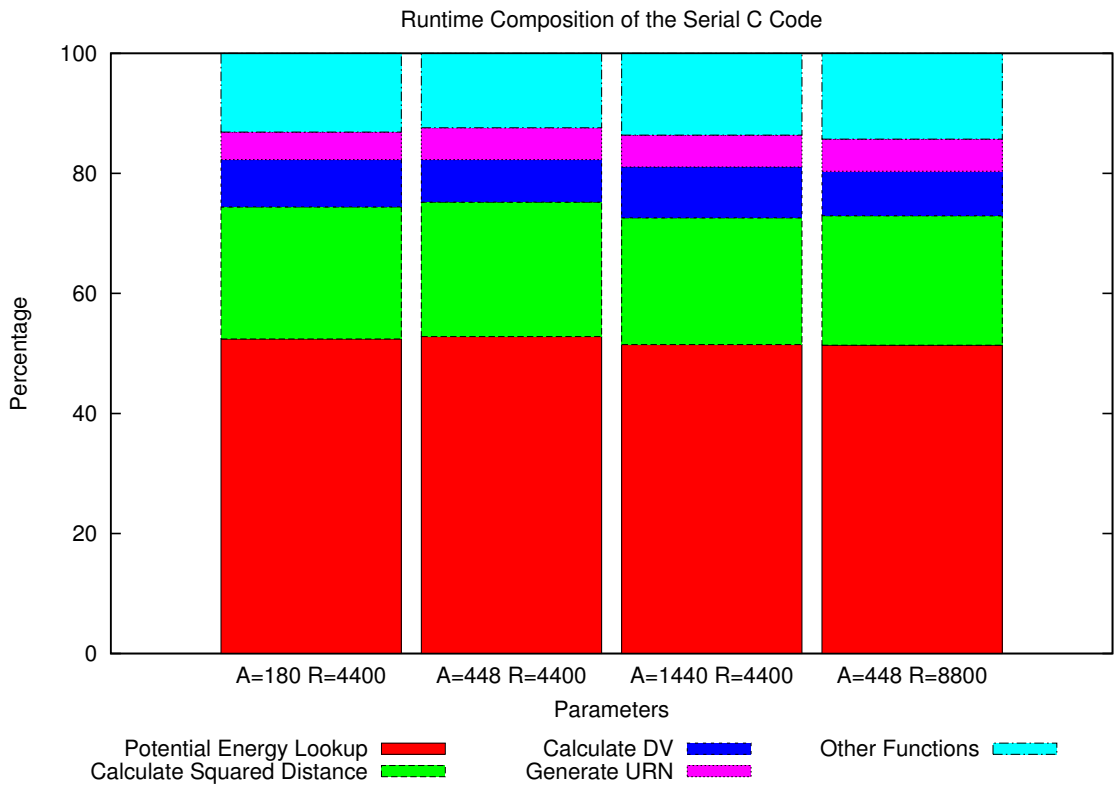


Figure 3.4: Runtime composition of serial C code

Optimization: Potential Energy Lookup

From the profiling, it is obvious that the most time consuming portion of the code is looking up the potential energy. For each squared distance, there are a total of seven double precision operations and 2 memory accesses, as shown in equations 3.8 and 3.9.

$$lookup_{original}(d2) = v[bin * 2] + v[bin * 2 + 1] * (d2 - d2min - .05 * bin) \tag{3.8}$$

$$bin = (d2 - d2min) * 20 \tag{3.9}$$

The potential energy lookup table, v, is set up using the code snippet in 3.5a. We can simplify the table lookup calculations using the changes in the code snippet

```
/* Hartree to kelvin conversion factor */
const double hart = 315774.65;
d2min  = 9.0;
bin    = .05;
binvrs = 1.0/bin;

for(uint32_t i=0; i<NVBINS; i++)
  {
    double d2 = (i) * (bin) + (d2min);
    v[ i * 2 ] = hfdbhe(d2);
  }

for(uint32_t i=0; i<NVBINS-1; i++)
  {
    v[i * 2 + 1] = (v[(i + 1) * 2] - v[i * 2]) / (bin);
  }
```

(a) Original lookup table

```
/* Hartree to kelvin conversion factor */
const double hart = 315774.65;
d2min  = 0.0;
bin    = .05;
binvrs = 1.0/bin;

for(uint32_t i=0; i<NVBINS; i++)
  {
    double d2 = (i) * (bin) + (d2min);
    v[ i * 2 ] = hfdbhe(d2);
  }

for(uint32_t i=0; i<NVBINS-1; i++)
  {
    v[i * 2 + 1] = (v[(i + 1) * 2] - v[i * 2]) / (bin);
  }

for(uint32_t i=0; i<NVBINS; i++)
  {
    double d2 = d2min + (double)(i)*bin;
    v[ i * 2 ] -= (v[ i*2 + 1] * d2);
  }
```

(b) Improved lookup table

Figure 3.5: Code snippets of the potential energy lookup table initialization

in figure 3.5b. With this reordering, the table can be accessed using equation 3.4, reducing the overall number of floating point operations to 3 per lookup (more than 50%).

Optimization: Neighbor Localization

One optimization is to maintain an array of vector distances of the ideal crystal sorted in order of increasing distance. This allows for better caching for the potential energy lookup table because if the atoms stay within the relative same area, they should have similar distances with their neighbors. Since the R2 values are calculated in a roughly sorted order, access to the lookup table should be within the same proximity of memory locations, thus allowing for potentially better performance by decreasing the number of L1 cache misses.

Optimization: Potential Energy Lookup Reduction

Profiling the code indicates that the potential energy lookups and squared distance calculations are the most time consuming portions of the code: requiring approximately 70% of the runtime. It is logical to examine that code section to see how the performance can be improved. First, the most computationally expensive portion of this step is the calculation of the squared distances in equation 3.3. All of these distances need to be calculated for further steps such as the potential energy calculations. The neighboring atoms are not guaranteed to be in consecutive memory (e.g., atom 1 may interact with atoms 1, 3, 4, 12, ... and not 1, 2, 3, 4, ...) so the loop is not vectorizable by the compiler nor is it advantageous to do vectorization by hand due to the large number of serial loads from memory. This is evidence of the inability for complete vectorization.

At first glance, it appears that we are already optimizing the lookup the best way possible despite having sparse lookups. The lookup table is $20000*2*8/1024 \approx 312$KB meaning most can fit into a 256KB L2 cache. However, the squared distances mostly within a limited range (e.g., not many lookups are within a distance of 120 a_0^2 for the density of interest) so the most commonly used distances are stored in L1 so we are getting decent cache behaviors. As alluded to previously, if the density is increased or decreased, the squared distances vary more, causing caching performance to be worse. We are also providing similar lookups to the table by keeping the atoms in sorted order by their ideal position distances so we that we can increase cache efficiency as much as possible. In addition, the lookups cannot be vectorized either due to the lack of guarantee that consecutively computed squared distances result in consecutive memory lookups.

However, can the number of lookups be reduced? If the number of lookups can be reduced, we can save clock cycles since the table is larger than L1 and L2 caches on some systems. Looking at the behavior of the system, we can see that for each atom, there is a subset of interacting atoms that are, in the ideal case, the closest to the

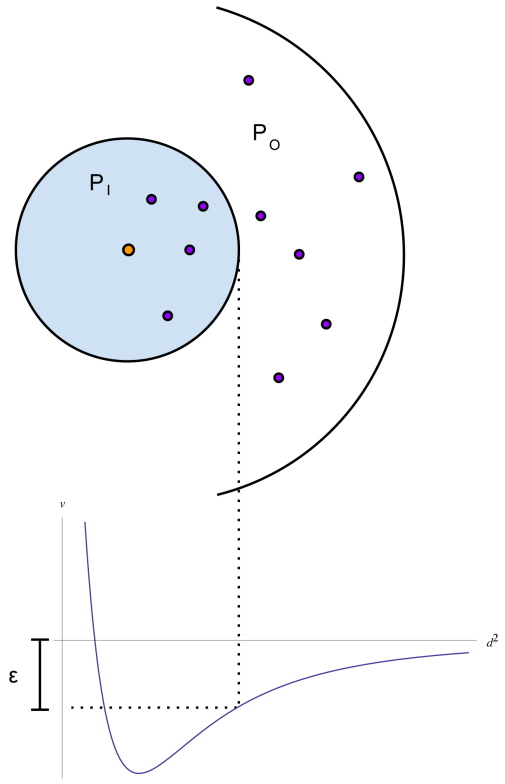


Figure 3.6: Diagram of the inner and outer regions of neighbors

atom at hand, P_I. Likewise, there is a subset that are farther away from the atom at hand, $\mathrm{P}_O = \mathrm{P}$ - P_I. This is shown in figure 3.6.

The P_I closest atoms have the highest contribution to the potential energy of the interacting pairs while P_O has the lowest. The potential energy can be split up into the sum of the potential energies as shown in equation 3.10

$$\begin{aligned} dv &= \sum_{i=1}^{P} (v(d^2_{old,i}) - v(d^2_{new,i})) \\ &= \sum_{i=1}^{P_I} (v(d^2_{old,i}) - v(d^2_{new,i})) \\ &+ \sum_{i=P_I+1}^{P} (v(d^2_{old,i}) - v(d^2_{new,i})) \\ &= dv_{P_I} + dv_{P_O} \end{aligned} \tag{3.10}$$

Typically, a move is accepted as described section 3.2.4 and shown in equation 3.11 where URN is a uniform random number. Plugging 3.10 into 3.11 and moving terms around, the acceptance can be put in terms of dv_{P_O} as shown in 3.12.

$$URN < e^{(dv*\Delta\tau)} \tag{3.11}$$

$$ln(URN) - dv_{P_I} * \Delta\tau < dv_{P_O} * \Delta\tau \tag{3.12}$$

Since the closest neighbor in the set of P_O outer neighbors, $d_0^2 = min(d^2 \in P_O)$, will have an energy of $v(d_0^2) = \epsilon$ (as seen in figure 3.6), all of the remaining neighbors, if in the monotonically increasing section of the potential energy curve, must be within the range shown in equation 3.13

$$\epsilon <= v(r_i) < 0 \qquad \forall i \in P_0 \tag{3.13}$$

Each difference in potential energy for the new and old positions for the outer neighbors will be within this range. This means the summation of these energies is also within this range. That is,

$$dv_{P_O} = \sum_{i \in P_O} dv_{P_{O,i}} <= P_O * \epsilon \tag{3.14}$$

Now that we have this upper bound, how can we use this to reduce the computation time? If the closest outer interacting pair has a potential energy that is within the monotonically increasing portion of the potential energy curve, it is possible to decide if the remaining outer interacting pairs could have enough contribution to sway the acceptance or rejection decision. This can be done using equation 3.15 as visualized in figure 3.7a. The purple, red, and blue regions are ones that can be decided without having to calculate all of the energies based on the upper bound on the contribution from these energies. The green region indicates that the outer interacting pairs could have enough contribution to affect the acceptance or rejection

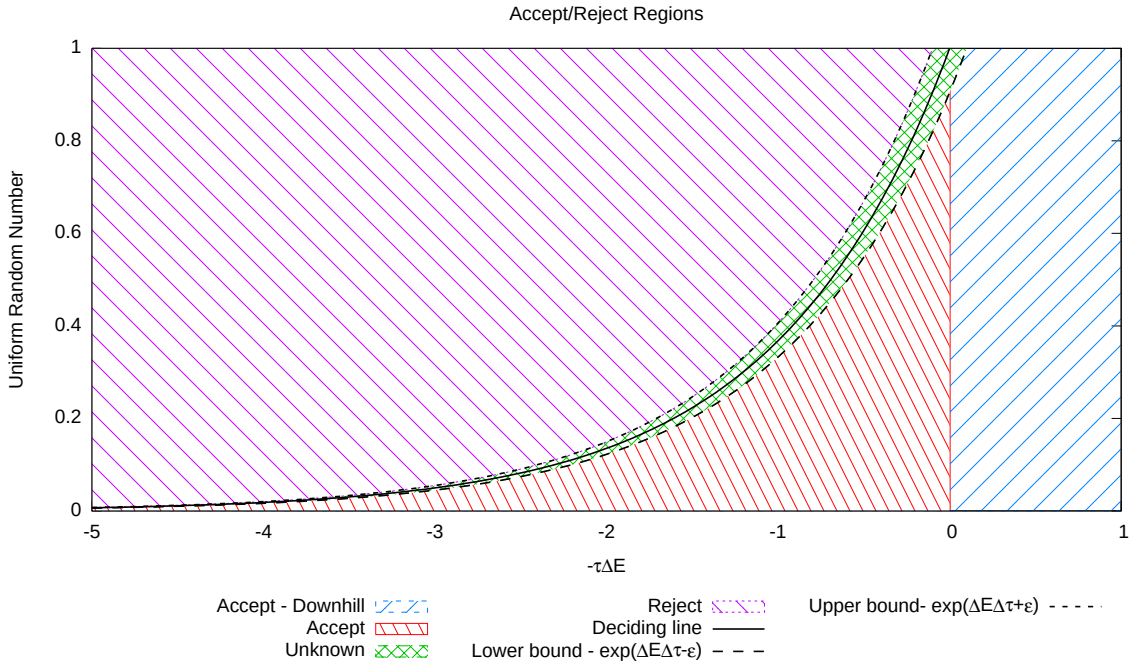
decision.

$$\begin{aligned} &\text{accept if: } ln(URN) - dv_{N_O} * \Delta\tau < N_I * \Delta\tau * |\epsilon| \\ &\text{reject if: } ln(URN) + dv_{N_O} * \Delta\tau > N_I * \Delta\tau * |\epsilon| \\ &\text{else, calculate rest of energies and accept/reject normally} \end{aligned} \tag{3.15}$$

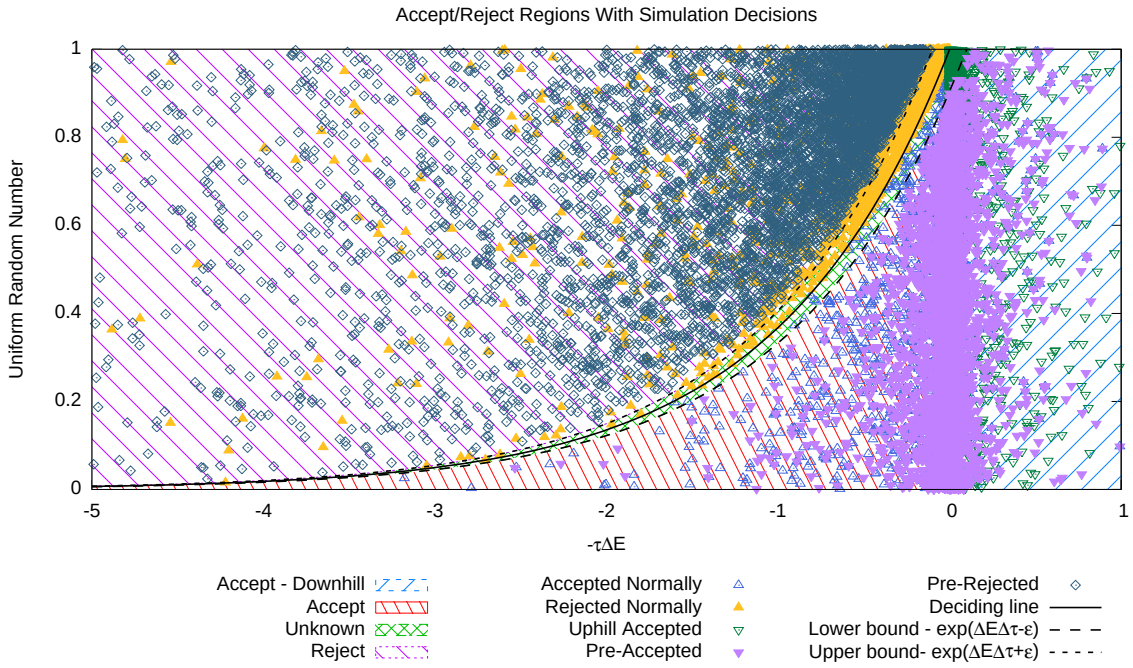
How does this affect performance? This is dependent on the density of the crystal, the number of inner pairs, and the $\Delta\tau$ time step that are used. If the crystal's density is high, meaning the atoms are closer together, there will be less moves that can be predetermined using this method due to the P_O neighbors having more potential energy contribution. If the density is smaller, meaning the atoms are farther apart, more moves can be predetermined. In a test case with a density of ρ =0.004614 and $\Delta\tau$=500.0, roughly 97%-98% of the moves can be predetermined resulting in a 15% decrease in overall runtime. While this seems small, this is still a significant speedup when considering that the runtime of a full run is on the order of hours or days for big crystals and large numbers of replicas. This will be discussed in section 5.1.1. Figure 3.7b shows the accepted and rejected points that were both predetermined and calculated normally for a subset of a run on 1440 atoms with ρ =0.004614, P =56, and P_I =18.

3.3.2 Parallel - Vectorization

The computations done by QSATS on a single replica cannot easily be vectorized beyond what the compiler already takes care of because of sparse accesses to memory throughout the program. However, it is possible to vectorize across replicas, that is, execute multiple replica movements simultaneously. This is possible because all of the odd replicas are independent from each other and all the even replicas are independent. In this way, we can pack replica information into memory for easy recovery to reduce the amount of loads from memory. Refer to figure 3.8 to see how the memory is packed for an SSE vector. For SSE, two replicas are packed together



(a) Regions only



(b) Regions with simulation decisions

Figure 3.7: Predetermination regions for accepting or rejecting a move

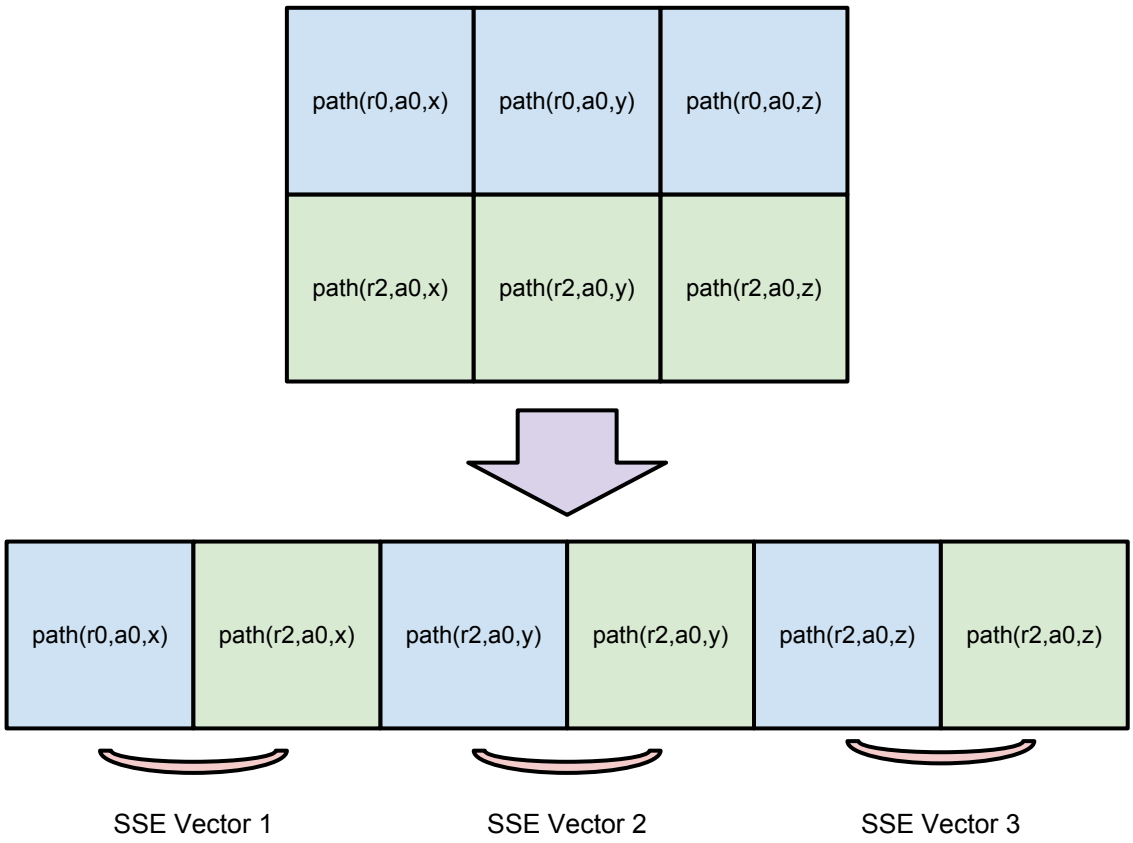


Figure 3.8: Memory packing for SSE vectorization

since SSE vectors can hold 128 bits (i.e., 2 doubles). For AVX, four replicas are packed together since AVX vectors can hold 256 bits (i.e., 4 doubles).

The information for two replicas are interleaved and packed together so that any operation on, for example, the x coordinate of atom 0 in replica 0 will also be on the same atom in replica 2. This allows for random number generation (both uniform and Gaussian), squared distance calculation, and accept reject determinations to be performed in parallel. As mentioned in section 3.3.1, the neighbor information lookup cannot be vectorized due to the fact that atoms in different replicas might be looking at interaction neighbors that are consecutive in memory so this information must be serialized. Once this information is loaded from memory, the rest of the calculations can be vectorized. Also, since the replicas most likely have different

squared distance values, it is not possible to vectorize potential energy table lookups. However, once these are serialized and stored into vectors, the difference in potential energy calculations can be performed using vectors.

3.3.3 Parallel - Threads

As mentioned in the previous section, it is possible to parallelize QSATS across even and odd replicas. This is an ideal situation for threaded implementations as there is little synchronization. The only synchronization that is needed is when switching from odd to even or even to odd numbered replicas. The cost of this synchronization is minimal because each thread is working on the same amount of data as the others. Therefore, each thread should complete their work at approximately the same time. Each thread works on R/T replicas thus performing $R * A/T$ iterations.

There are two libraries that can be used for the the threaded implementations: pthreads and OpenMP. First, pthreads were used to implement the threaded version. In the initialization stage, QSATS forks off T threads and sticks them into a thread pool to wait for work to be issued to them. The main thread sends a signal to the threads informing them to wake up and start on their replicas. Each thread starts to work on the (thread id)th replica in the set. For example, thread 2 starts trying to move the atoms in replica 3 if the master thread tells the threads to work on the odd replicas and 2 if it is working on the even replicas. Once all of the threads finish moving all the atoms in their set of replicas, they signal to the master thread and go to sleep to wait for more work. At this stage, the master thread waits until it gets a signal from all the threads (synchronization). This process is repeated I times until the simulation is complete. In the case of snapshots, the worker threads continue to sleep while the master thread writes the snapshot file in order to prevent overwriting data needed for the snapshot.

The OpenMP implementation follows along with the same scheme. There is a pool of threads that waits until work is available. These threads are signaled to start

working through the use of a "parallel for". Once all the threads are completed with their work, they synchronize and wait for more work. The benefit of using OpenMP is that there are minimal code changes to achieve the same result.

To extend this threaded implementation one step further, an OpenCL [81] implementation has also been developed. Using OpenCL allows the exploration of code on multiple different architectures such as GPUs and microprocessors. This implementation is described in the GPU chapter so the reader is referred to chapter 4 for in depth details. However, as an overview, the OpenCL implementation for the CPU has a single kernel that performs the QSATS method where each thread executes the algorithm on a single replica. After each kernel call, the threads are synchronized before moving onto the next set of replicas. The benefit of using OpenCL is that the kernel can be compiled with automatic vectorization support without having to change any code. Because microprocessors do not have the texture units that GPUs have, the kernel is slightly altered from the GPU version to use global memory and software linear interpolation. Other than removing texture accesses, the kernel is the same as the GPU kernel code.

3.3.4 Parallel - MPI

At this point, it is time to explore multi-node/process implementations. To do this, this work uses the Message Passing Interface (MPI) [31] to perform inter-node communications. As a first pass, this work uses the parallelization scheme used in [12]. In the original QSATS Fortran code, parallelization is achieved by using a master-worker model where a central process delegates work to a set of "worker" processes. This approach is a flexible parallel programming paradigm in that it is applicable to a wide range of applications and provides acceptable load balancing. In the QSATS application, parallelization is performed using the following approach.

Note: For all of the following implementations, vectorized versions are available using the approach described in section 3.3.2.

Master-Independent Workers

Out of all the N MPI processes that are forked at the start of the program, there is one master. The master takes care of all the initialization, I/O, and distribution of work. While the master is doing the initialization described in section 3.2.1, the workers are allocating enough space for a single work packet. For this implementation, a work packet consist of a single replica's RNG state information and two replicas' displacement information (i.e., one for the replica at hand and one for the average of the neighboring replicas as described in equations 2.2-2.4). That is a total of $(8 + 2 * 3 * A) * V$ doubles that are allocated to hold the incoming work packet from the master. Once the workers are initialized, they each send a small message to the master indicating they are ready for work. Once the master is initialized, it will start distributing the work among the workers. Upon receiving a "ready" message, the master will copy the next replica atom displacements in the replica work queue to a new buffer and calculate the averages as discussed in section 3.2.2. The master will then send the replica information and RNG state information to the worker and wait for another "ready" message. The master starts with the odd replicas, synchronizes after all the odd replicas are completed, and repeats with the even replicas. When a snapshot is taken, the master ensures that all the workers have returned the work that they are working on. Once all the workers are idle, the master writes the snapshot to file then starts distributing work again. This process continues until all of the iterations are complete.

While this is an easy to implement parallelization scheme, there are some downfalls. First, there is significant amount of data shuffling and communications. For each iteration, there are a total of $4 * R$ communications (two for replica displacement and two for RNG state to and from the master). On the assignment of work, the package size for the replica displacement and RNG state information are $48 * A$ bytes and 64 bytes, respectively. For the return of work results, they are $24 * A$ bytes and 64 bytes, respectively. While these messages are relatively small for

small problem sizes, as discussed in the later sections, the performance for real science problems (e.g., large crystal sizes or large numbers of replicas) would be limited by the large number and size of communications. For a single iteration, the amount of communications is $4 * I * R$ for a total of $I * (72 * A + 128)$ transferred bytes. Second, the master will get bogged down with work as the number of workers increases. The master will struggle to keep up with the incoming results from the workers, causing the workers to have longer idle times. Finally, all of the workers must wait for the master to write the $24 * A * R/11$ byte snapshot to file. This is time where the workers could be doing work, yet they have to sit idle until the writing is complete.

Master-Dependent Workers

The high number of communications leads to the next implementation. To address the problem of too many communications, the master now assigns a large chunk of work to each worker, $R_w = R/W$ replicas per worker, at the initialization stage. Each worker will attempt to move all the atoms in all of the replicas that were assigned to it. When all the work is complete, the workers will exchange "end" replicas with each other as shown in figure 3.9. For instance, when the third worker is done with all of its work, it will send its first replica to the second worker and last replica to the fourth worker. This communication is needed so that the neighboring workers can calculate the average of replicas as described previously. In order to make the programming easier, the workers do the following. First, they all work on the odd replicas. When they have completed all of the odd replicas, they send their first replica to the left (i.e., $rank - 1$). Then they all attempt to move the atoms in the even replicas. At the completion of the even replicas, they send their last replica to the right (i.e., $rank+1$). This ensures that all replicas are doing the same instructions and sending in the same direction to avoid deadlock. Each of the replica exchanges are $24 * V * A$ byte packages. For each iteration, there are two exchanges per worker (except for ranks 1 and W) meaning there are total of $W - 2$ communications each of size $24 * V * A$ bytes. Since QSATS has a particular order in which communications

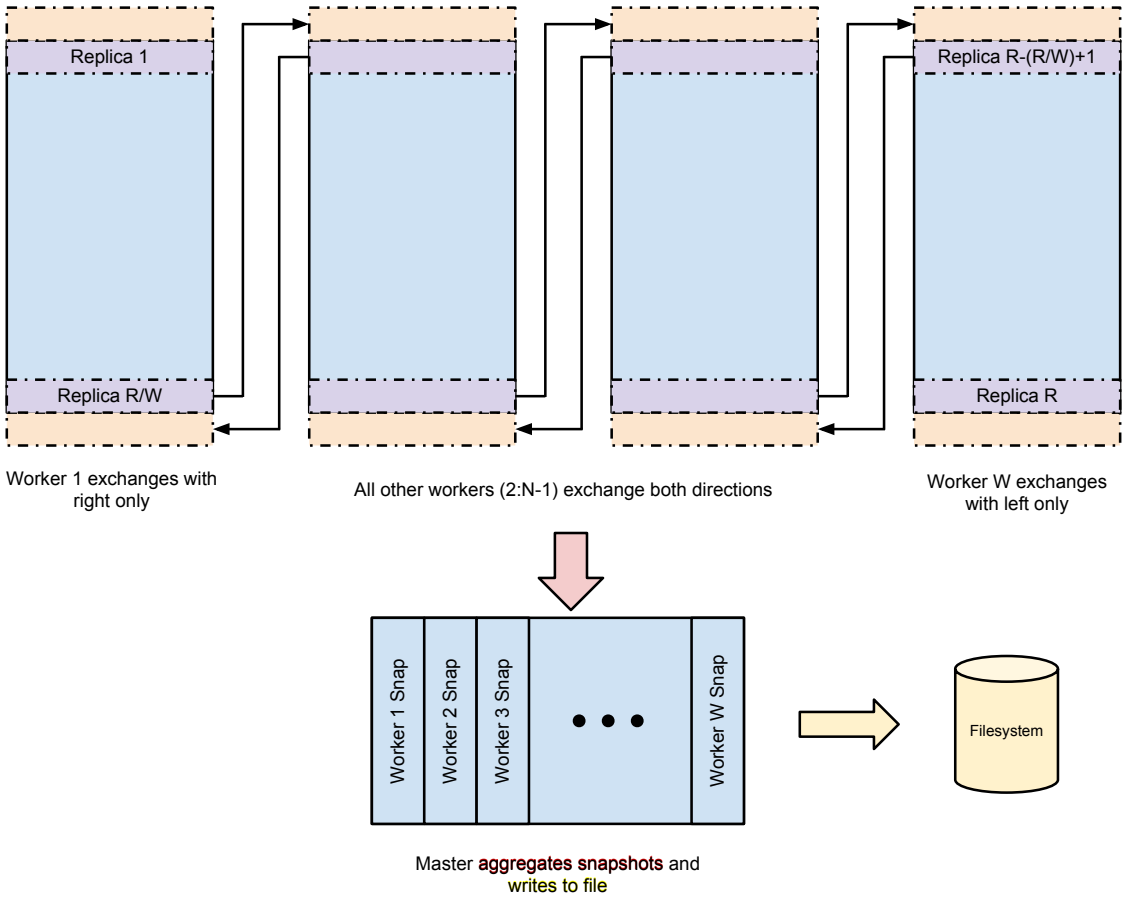


Figure 3.9: Exchange of work among workers

are performed, this puts a requirement on the number of replicas that can be used. There must be enough replicas to evenly split across all of the workers and each worker must have an even number of replicas. This ensures that each worker does the same about of work as the others (load balancing) and that the first replica for each worker is an odd number replica and the last is an even number. Also, in order to collect statistics from the snapshots about the end replicas (replicas 0 and R-1), the number of replicas + 1 must be divisible by 11. That is,

$$\begin{aligned} R \bmod W &= 0 \\ R \bmod 2 * V &= 0 \\ (R+1) \bmod 11 &= 0 \end{aligned} \tag{3.16}$$

The master sits idle until there is a snapshot, at which point the master will aggregate all the snapshot information from all of the workers. Once the workers complete the snapshots, they continue doing work while the master asynchronously writes the snapshot information to a file.

This approach takes care of the many small package communications between the workers in the master; however, it still has a limitation. This limitation is that as the number of workers grow, so does the communication overhead from the master perspective for the snapshots as it has to collect from all of the workers. So, this implementation only temporarily hides the communication bottleneck by only performing many large data transfers occasionally. As the frequency of snapshots increases, the more of an impact it will have on the runtime. With a relatively small number of workers and a large snapshot frequency, this implementation will have no problems. However, the fewer snapshots we have, the less information we can gather about the simulation. Also, the fewer workers we have, the longer the program will take to run.

Multiple Master-Dependent Workers

How can we address the problem of the master being swamped with snapshot information? As mentioned, an undesirable solution would be to reduce the amount of snapshots that are taken and the number of workers that are used while still using the single master approach. However, it is possible to reduce both the number of snapshots and the number of workers that send snapshot information to a master by using multiple masters, as shown in figure 3.10. Each worker is assigned a master and each master has a subset of the workers. Each master collects snapshot information only from its workers subset and writes to its personal file. For instance, assume there are a total of 960 total processes, 4 of which are masters. In this example each master has $W_g = (N - M)/M = (960 - 4)/4 = 239$ workers. Workers [1-239] belong to master 1, [240-478] to master 2, etc. Each subset of workers communicates with the neighbor workers (i.e., rank$\pm$1) and their corresponding master. Having the

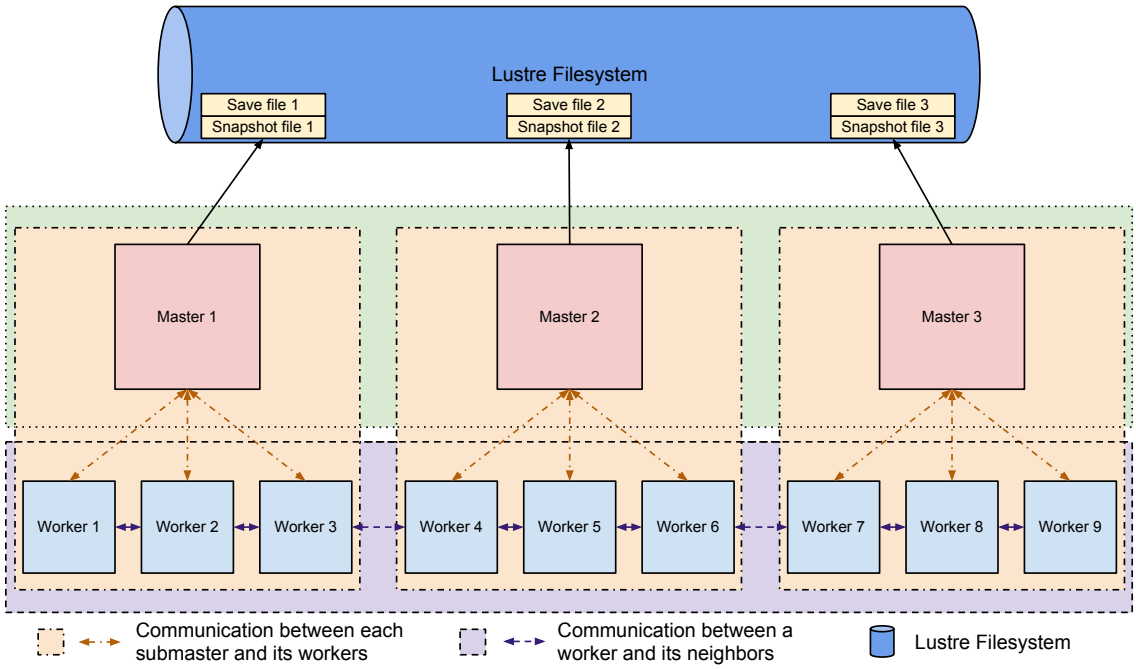


Figure 3.10: Multiple master MPI approach

communications split in such a way allows for the number of communications to be reduced from $N-1$ to only a fraction of W_g for each master.

To help ease the programming and orchestration of the sends and receives, the main communicator, MPI_COMM_WORLD, is split into three different types of communicators: workers, masters, and groups. All of the workers share a communicator so that their ranks range from $[0, W)$. This is helpful for exchanging the end replicas so that each worker can simply use $rank \pm 1$ instead of trying to accommodate skipping over the master ranks. Likewise, the masters have their own group such that their their ranks range from $[0, M)$. Since the masters are independent of each other, this communicator has no real use, instead it is just a side effect of creating the workers communicator. Finally, there is the communicator that describes either group. Each group has N/M processes where rank 0 relative to the communicator is the master of the group. To perform the snap shot, all of the workers send the snapshot information to rank 0 relative to their group communicator. As

mentioned, this was solely to ease the orchestration of the communications among processes.

Another advantage of having multiple masters on large systems such as supercomputers is that the program is able to take advantage of parallel I/O file systems such as Lustre [82]. As shown later in this dissertation, in order to get peak I/O speeds from parallel file systems, it is necessary to have multiple processes read/write files on the file systems. This means that a snapshot can effectively be written to file much faster if multiple processes are writing rather than a single process (can also be seen [83]). Not only does this new multiple master implementation increase the I/O read and write speeds, it also reduces the amount of I/O each master has to do. In the single master case, each snapshot is $24 * A * R/11$ bytes; however, the multi-master approach splits this evenly amount all the masters. Each master only has to write $24 * A * R/11/M$ bytes for each snapshot.

3.4 Intel MIC Implementation

Intel is currently developing a new architecture called Many Integrated Cores (MIC) [52]. The goal of this architecture is to have many simple CPU cores stamped onto a single die that can provide high performance with little code porting. Intel prides itself on the MIC's ability to run existing codes by only recompiling the source with new flags to its icc compiler. This means rapid development of programs onto its new architecture without having to learn a new language or re-code existing programs.

Saying this, the microprocessor serial, threaded, and MPI implementations were all "ported" to the Intel MIC to test Intel's claim of performance without rewriting code. As promised, the code was able to be compiled and run successfully; however, it is still possible to provide improvements by modifying some of the code.

One of the main architectural features of the Intel MIC is the 512-bit wide vector units. Using a similar approach to the SSE and AVX vectorization, a MIC vectorized implementation was written to squeeze a little more performance out of the code.

Due to some generic programming and the use of C macros, this implementation was quickly vectorized and running since the intrinsics (while not the same) are similar to the SSE and AVX.

Chapter 4

Graphics Processing Units

Computer video game developers have pushed the limit of graphics processing units (GPUs) to a point where billions of calculations per second are needed in order to accurately render realistic graphics of modern games. Computational scientists have been able to exploit the massive amount of cores and memory bandwidth to obtain high performance in their scientific applications, opening a new world of possible problems that can be explored. Supercomputer vendors are rapidly adopting the use of GPU accelerators to obtain high performance rates and performance per watt ratios [7, 8, 6]. While there are a number of vendor and programming options, NVIDIA is by far the leader in high performance computing due in part to the successful adoption of their compute unified device architecture (CUDA)[38]. However, OpenCL is a competitor to CUDA that allows for the programming of many architectures, including GPUs, using a similar syntax. This chapter provides an overview of the graphics processing unit implementation using CUDA and OpenCL on NVIDIA and ATI GPUs.

4.1 NVIDIA GPU Architectures

This section provides a number of details about the GPU architectures used in this work.

4.1.1 Tesla

NVIDIA's breakthrough CUDA architecture, Tesla, was first introduced with the GeForce 8800 GPU [84]. The Tesla architecture unifies the vertex and fragment processors that were present in traditional GPUs. Containing an array of multithreaded streaming multiprocessors (SMs), each consisting of eight streaming processor (SP) cores and two special function units, the Tesla architecture paved the way for general purpose computation on graphics programming units (GPGPU). There are various types of memory (i.e., global, registers, shared, texture, and constant) that each have their advantages and disadvantages. Global memory is the main memory that has read and write abilities and up to 4GB storage space; however, access is slow at 400-800 clock cycles. The memory clock rate is 800MHz and has a 512-bit interface [85]. Registers are the fastest memory units available in the GPU at just a couple of cycles per access but are limited to 16k 32-bit registers. The shared memory allows for storage of 16KB per SM and can be used to store common data that has a high reuse rate. For memory that is read-only and has a regular access pattern, 64KB of cacheable constant memory is available. For read-only memory with irregular access patterns, texture memory allows caching and hardware linear interpolation in 1-D, 2-D, and 3-D data. Each of the SMs have access to these memories. Three SMs are clustered into a texture processing cluster (TPC) that each share a memory controller. All of the threads of a blocks are assigned to an SM and run concurrently. Each SP core within an SM has a scalar multiply-add unit with 8 such units per SM in all. Each special function units contains four floating-point multipliers. Each SM is able to manage 768 concurrent threads in hardware, allowing for the high parallelism of this architecture. On the Tesla c1060, there are 30 SMs, each with eight SPs and one double-precision core. This amounts to 240 single precision cores or 30 double-precision cores. All of the above information comes from [84].

4.1.2 Fermi

The next generation of GPU architectures is the 3 billion transistor Fermi. This architecture features up to 512 CUDA cores where each executes a floating point or integer instruction per clock for a thread. 32 cores are grouped into SMs for up to a total of 16 SMs per GPU. Each core contains a pipelined integer ALU and FPU. Each SM contains 16 load/store units and four special function units. The Fermi architecture contains two caches: L1 and L2. The L1 cache is shared with the shared memory space (totaled at 64KB) and is configurable to 16KB or 48KB. The L1 cache is a hardware managed cache, meaning less programmer interaction to use this high speed memory. The L2 cache is 768KB of memory that is shared across all SMs, but the L2 cache can only reduce over-fetch in the case of scattered memory accesses, for example. Like the Tesla architecture, there are also constant (64KB) and texture (up to 8KB) memories. The texture memory has the ability to perform hardware linear interpolation as well. Each SM has their own texture cache, unlike the Tesla where three SMs shared a single texture cache. The Fermi architecture doubles the number of registers from 16k to 32k. The memory clock rate has been increased from 800MHz on the Tesla architecture to 1848MHz [86]. Each SM contains a dual warp scheduler that is able to select two warps and issue one instruction from each simultaneously to a group of 16 cores, 16 load/store units, or for special function units. For the GeForce GTX 480 consumer card, there are 15 SMs for a total of 480 cores, but it has crippled double precision units making double precision an eighth the speed of single-precision instead of the expected half as noticed with other consumer cards [46]. Professional grade GPUs such as the Tesla M2070/2075 and M2090 bring this ratio back to the expected half. In particular, the M2090 provides 512 cores, 6GB of global memory, and 1.33 TFLOPS of single precision peak performance. The Keeneland GPU cluster [6] contains 360 Tesla M2090 GPUs. Unless otherwise stated, all the Fermi information comes from [38, 87, 88].

4.2 GPU Programming Synopsis

Modern GPUs are able to be programmed in a number of ways. NVIDIA GPUs are often programmed using their custom CUDA language [38]. However, it is also possible to program them with a framework that is designed for programming parallel architectures called OpenCL [30]. This framework provides the ability to run code on various architectures without the need for reprogramming. This section details the various aspects of CUDA and OpenCL.

4.2.1 CUDA

To draw developers to program on their devices, NVIDIA developed the compute unified device architecture (CUDA). CUDA is an extension to the C/C++ languages, giving the user direct access to device memory management, thread level parallelism, and a large number of cores for general purpose applications [89]. CUDA can be used on a large number of NVIDIA GPUs ranging from average desktop and laptop models, such as their NVS line, to gaming devices, such as the GeForce line, to high performance computing using the Tesla line [90]. In comparison to traditional microprocessors, these GPUs have more transistors devoted to data processing and high memory bandwidth allowing for many instructions to be executed at the same time (see figure 4.1). Single instruction, multiple data (SIMD) applications thrive on this parallelism as there is little to no data sharing, allowing computations to dominate. Essentially, there are three abstractions that CUDA provides: a hierarchy of threads, shared memories, and barrier synchronization. These abstractions allow the programmer to break their problem into a set of chunks that are independent from each other (block level) where each chunk has a set of threads that work together. Of course, this is only one programming approach possible with CUDA. It is possible to also split all work into completely independent threads. This is the approach that this dissertation takes and will be discussed in more detail in section 4.3.

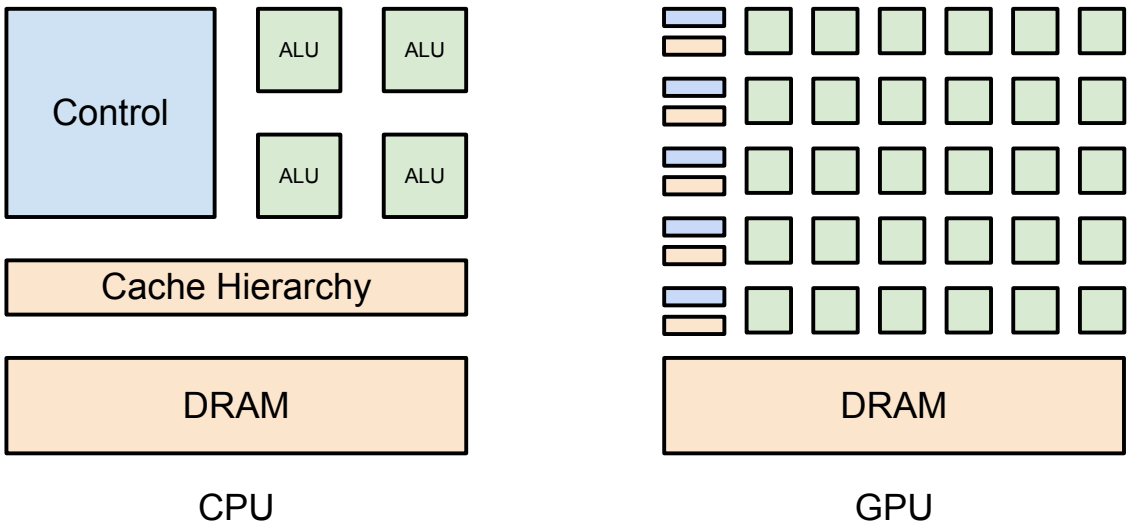


Figure 4.1: CPU Architecture versus the GPU [38]

4.2.2 OpenCL

OpenCL is a framework that was developed to ease the programming efforts to write code on various heterogeneous architectures ranging from microprocessors to GPUs. OpenCL also reduces the work it takes to program devices across vendors such as ATI and NVIDIA GPUs. This is accomplished by a standard specification developed by the Khronos Group that is accepted by various vendors. Using a kernel C99 syntax that is similar to CUDA kernels, OpenCL provides programmers with the ability to develop parallel computing applications using data-based or task-based parallelism [30].

4.3 VPI Implementation

The key to implementing any GPU-enabled application is determining what portions of the algorithm can benefit the most from hardware acceleration. In the case of VPI, there is no one portion/function that would map well onto the GPU due to data dependencies. Therefore, instead of accelerating one portion of the program, this work accelerates the entire QSATS program by exploiting parallelism across replicas,

similar to the MPI and threaded microprocessor implementations. That is, perform all the odd replicas in parallel, synchronize, and repeat with the even replicas. This allows for memory to stay on the GPU throughout the entirety of the program while only needing to copy data to the host in order to periodically take snapshots.

In the remainder of this section, a number of implementations are discussed ranging from a naïve implementation with poor memory access patterns to highly optimized implementations that take advantage of many architectural features in CUDA. Also, this section describes the optimized OpenCL implementation.

4.3.1 CUDA

Naïve Implementation

As a first pass, the initial step was to get an implementation that produces the same results as the "gold standard" microprocessor implementation. To do this, the microprocessor implementation is essentially just copied into the CUDA kernel code. While this is the easiest implementation, it is not the smartest as it does not take into account any of the programming optimization techniques described in [38]. Most importantly, this includes no coalescing, which murders performance. The purpose of this step is to merely ensure correctness, not to produce optimized code.

Out of necessity, this code also ports the uniform and Gaussian random number generators onto the GPU. It is possible to use GASPRNG [78] or CURAND [79]; however, to obtain the same results as the microprocessor implementation, it is needed to use the same random number generator to ensure exact results.

Coalesced Implementation

The next step was to coalesce memory accesses to/from global memory. This is one of the main ways that GPUs obtain performance by ensuring all accesses to global memory are coalesced in order to fill the data path. This involved rearranging data in such a way as to ensure that data accesses are consecutive in memory and are always

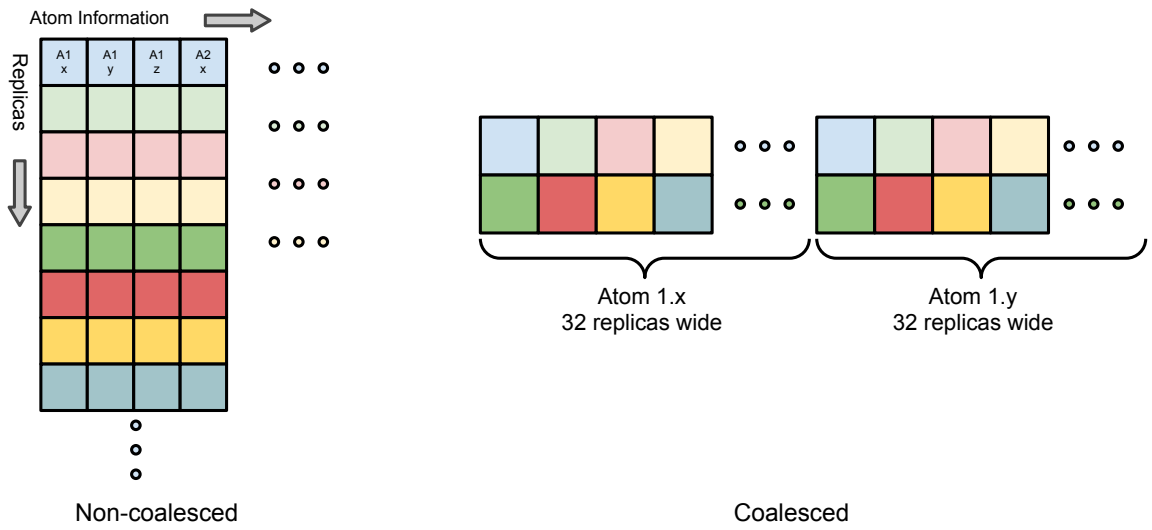


Figure 4.2: GPU data layout allowing for coalesced accesses

128 byte aligned. In order to access memory that is consecutive in memory, it was necessary to flip the ordering of the data from the easy to index layout of replicas by atoms to a mixed data layout as shown in figure 4.2. This layout allows for all of the threads of a warp (32 threads) to access sequential elements in memory that are 128 byte aligned. It is similar to the vectorized layout described in 3.3.2 where the width is 2048 bits wide to hold 32 doubles, enough data for an entire warp's access.

Due to the rearranging of the data to produce coalesced memory accesses, this implementation allows for 16 double precision fetches from memory to occur simultaneously. Before, each access to global memory had to be serialized as it was unable fill the complete 128 byte wide data path. However, the data path can now be fully utilized as for each warp, only two accesses have to be serialized, one for each half warp. This is essential for high performance due to the high latency incurred when accessing global memory.

According to the NVIDIA Visual Profiler, this implementation achieves a 98% coalescing rate for reads and writes. The 2% that is not coalesced is due to some of the conditionals in the uniform random number generator that ensure only positive values are generated.

Hardware Texture Linear Interpolation

Related to the coalescing above, programs with random, nonlinear access patterns perform poorly when accessing global memory. Unfortunately, this implementation uses a lookup table to calculate the potential energy as the calculation of the exact function is far too expensive to be performed in a reasonable amount of time on any of the architectures explored in this work. Originally, the lookup table is $20000 * 2 * 8 =$ 320KB in size and each thread could not guarantee linear accesses to the table since atom positions are calculated from Gaussian random numbers. Besides this, it is not very cacheable as the atom information for all the replicas will dominate the caches. This type of access pattern, however, is ideal for texture memory [38]. Texture memory also has its own separate cache from the global memory so the other memory accesses will not interfere with the lookup table.

As mentioned in section 3.2.1, the lookup table for the microprocessor implementation used the squared distance for the lookup; however, this results in the large table size of 320KB. This table can be reduced to $1,536$ elements by storing the potential energy of the log of a squared distance. This comes at the cost of calculating the logarithm which is an expensive operation. Reducing the table in this way presents a tradeoff: use the large table that can push valuable data out of cache or reduce the table size and perform a logarithm. In the case of the microprocessor implementation, the cost of the logarithm is significant enough to not warrant the use of the smaller table. However, on the GPU, the smaller table allows for greater caching in the texture units where it will not affect other caching data.

Another argument for using the smaller table on the GPU is that the GPU provides hardware units to perform a linear interpolation. This has great advantages in that the calculations can be accelerated and the random memory accesses get the benefit of fast accesses through the texture cache. The texture memory is set up to have a clamping address mode, linear filter mode, and element type read mode (as opposed

to using normalized floats). Once the squared distance has been put onto the log scale, only one function needs to be called, `tex1D`.

According to [38], the hardware texture linear interpolation units perform the interpolation as shown in equation 4.1.

$$tex(x) = (1 - \alpha)T[i] + \alpha T[i + 1] \tag{4.1}$$

$$\alpha = frac(x_B) \tag{4.2}$$

$$i = \lfloor x_B \rfloor \tag{4.3}$$

$$x_B = x - 0.5 \tag{4.4}$$

The table has to be rearranged in order to fit into the equation. First, the function is split into n_tex_pts evenly spaced segments across a representative range of the log of squared distance values, $[log_2(d^2_{min}), log_2(d^2_{max})] = [x_{min}, x_{max}]$. The lookup table is capped with a minimum x value in order to help reduce the memory footprint of the table while still accurately describing the potential energy function. Since

$$\lim_{x \to 0} V(x) = \infty, \tag{4.5}$$

an x_{min} value can be chosen such that the resulting energy is very positive in regards to the rest of the function such that an interpolation with any smaller distance will also have a very positive energy. Likewise, it is not necessary to cover very large distances either as

$$\lim_{x \to \infty} V(x) = 0. \tag{4.6}$$

Therefore, it is possible to select an upper bound on the x values such that for any larger distances the function can be estimated to be approximately a near zero value. The texture memory is set up in such away to clamp any supplied addresses that fall outside of this range.

To fill the table, equations 4.7-4.9 are used where the $hfdbhe$ function 4.9 represents the HFDB-He function [21].

$$xx = \frac{x_{max} - x_{min}}{n\_tex\_pts - 1} * n + x_{min} \tag{4.7}$$

$$d^2 = 2^{xx} \tag{4.8}$$

$$T[n] = hfdbhe(d^2) \quad i \in [0..n\_tex\_pts) \tag{4.9}$$

The minus 1 in n_tex_pts-1 is so that the last texture point where n = n_tex_pts is actually located at the x_{max}. The 2^{xx} is used so that when calculating the lookup index on the GPU, the fast $\log_2(d^2)$ function is used.

For each of the calculated squared distance, the index to lookup in the table is calculated by equation 4.10.

$$x = \big(log(d^2) - x_{min}\big) * \frac{n\_tex\_pts - 1}{xmax - xmin} + 0.5 \tag{4.10}$$

This equation is essentially inverting equations 4.7 and 4.8 with an additional 0.5 added to the result to compensate for the subtraction of 0.5 in equation 4.4. Equation 4.10 is used as the lookup table index for equation 4.1 to perform the linear interpolation in hardware.

It is necessary to pick values of x_{min}, x_{max}, and n_tex_pts to reduce the amount of error when interpolating the potential energy function. x_{min} is picked to be something small enough to accentuate the positiveness of the potential energy function at such small distances. For this value, it was determined that $\log_2(d^2_{min}) = x_{min} = 1$ is sufficient enough to represent the high potential energy at small distances as $V(x_{min})$=.3964. Likewise $\log_2(d^2_{max}) = x_{max} = 10$ is chosen as it has a potential energy of $V(x_{max})$=-1.373e-9. Any distance greater than this will only get closer to zero and have less contribution. n_tex_pts=1,536 is chosen in order to fill the entire texture cache of the Tesla c1060, 6KB, with single precision lookup values. To recap, the parameters used for the hardware interpolation are shown in 4.1.

Table 4.1: Hardware interpolation parameters

Parameter Name	Parameter	Value
Number of texture points	n_tex_pts	1,536
Minimum x value in table	x_{min}	1
Maximum x value in table	x_{max}	10

Due to hardware limitations, the hardware linear interpolation only supports single precision so the lookups lose precision with this optimization. Figure 4.3 shows that using the single precision, log scaled, interpolated table does not exactly reproduce the values of the function at the texture points; however, the differences are minuscule as can be seen at the zoomed inset graph. Since the table is now stored in single precision, it only occupies $4 * 1,536 = 6$KB of space, which completely fits into texture cache. Performance-wise, the cost of performing the logarithm is negligible in relation to the rest of the code due to the program being memory bound on the GPU.

Due to this reduction in precision and the use of the hardware units to perform the interpolation, the values of the simulation are no longer equivalent to the microprocessor code. To test the statistical equivalence, a run is performed using the microprocessor and GPU implementations for 10,000 iterations of 46,080 replicas taking snapshots every 100 cycles. The ELOC tool described in [12] is used to calculate the sum of the potential energy and local energy for the configurations. Doing this shows that even with the loss of potential energy precision and reduced lookup table size, the GPU still produces results that are statistically equivalent to the "gold standard" microprocessor code as they share the same mean and the same standard deviation. This can be visualized in figure 4.4.

Since using the texture hardware interpolation is not a very commonly used optimization technique used in HPC or computational sciences, example code is given in [91] that sets up, calculates a set of energies, and prints out the table [91]. This serves as an example code to demonstrate how to use these units.

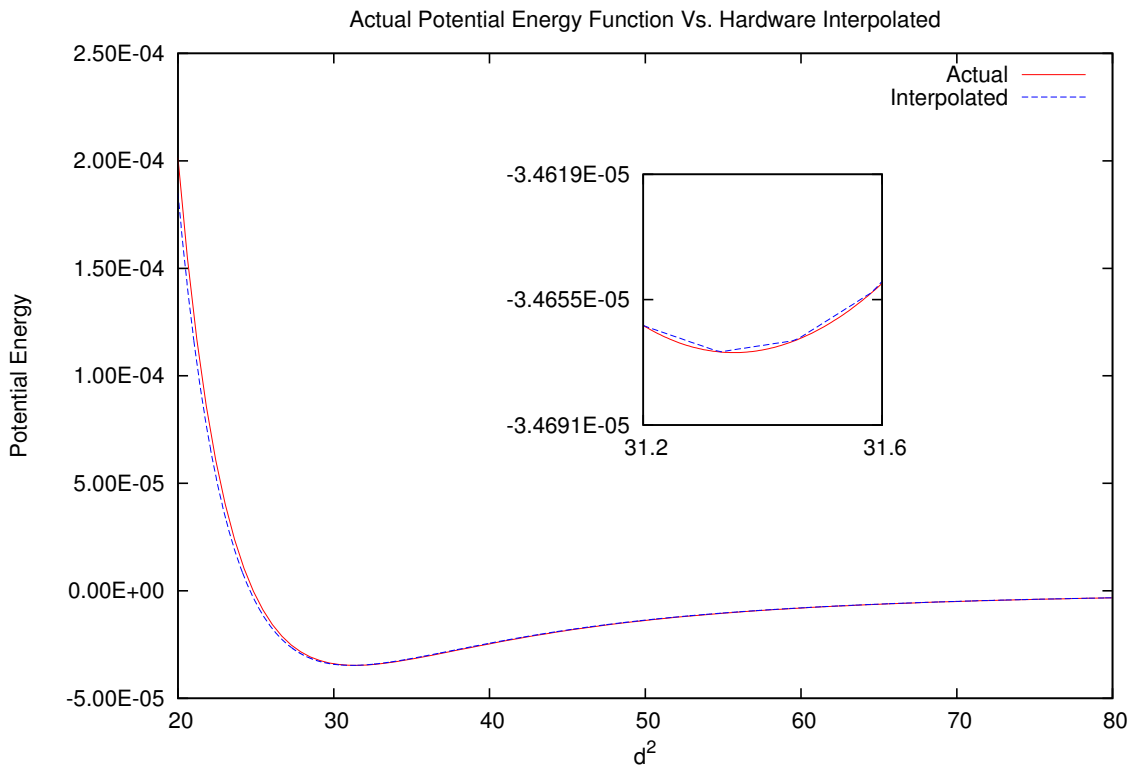
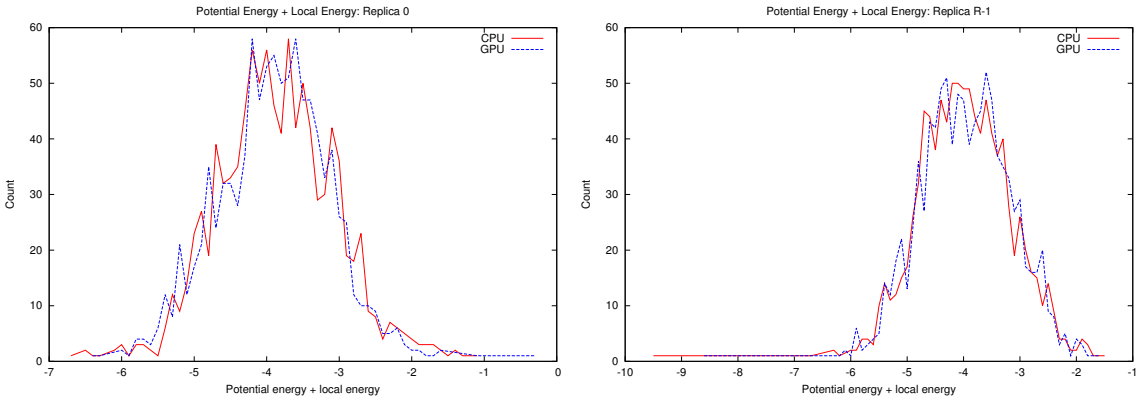


Figure 4.3: GPU log-scale potential energy lookup table



(a) First Replica **(b)** Last Replica

Figure 4.4: Statistical correctness for GPU implementation

Other code features

In order to draw relevant information from the simulations, QSATS must take snapshots of the crystal displacements for a set of the replicas. This involves copying

data from the device to the host at user specified intervals. To facilitate this, host memory for the displacement information is allocated using pinned memory and the cudaMallocHost function. Pinned memory allows for greater copy bandwidth from the device to the host. This is important as the amount of data that is copied is $\mathcal{O}(3 * A * R) = \mathcal{O}(A * R)$ elements for each snapshot. For instance, for a simulation of 65,536 replicas and 1,440 atoms, there is a total of $65,536 * 1,440 * 3 * 8 = 2.26$GB of data that must be copied from the GPU for each snapshot interval. Paged memory only allows for transfer with a bandwidth of approximately 1.7GB/s while pinned memory bandwidth is 5.5GB/s, a 3.2x difference [92]. Writing the snapshot files is performed asynchronously with the kernels in order to obtain as little computational delay as possible.

4.3.2 OpenCL

Since the kernel programming is very similar to the CUDA kernel syntax, only the final optimized version was written in OpenCL. This includes coalesced memory access optimization and texture memory interpolation. It is straightforward to convert CUDA kernel code to OpenCL as it is essentially a string replace. However, this isn't true for the hardware texture linear interpolation. Instead of textures, OpenCL calls these units images. OpenCL v1.1 (the version shipped from NVIDIA) does not support 1D images so the texture lookup table needed to be mapped to a 2D image. This had its own complexity as there were very few examples on how to use these units. The following is how the OpenCL hardware image interpolation units are set up.

The lookup table is first initialized on the host as described in section 4.3.1 and copied to the device. Then device image memory is allocated to have an image format of CL_A, CL_FLOAT as discussed in the specification [81] with a width of n_tex_pts elements and height of 1 element. CL_A specifies the number of channels and the channel layout in memory. CL_FLOAT specifies that the image will hold floats as

opposed to integers. Next, the code calls a kernel that fills up the image with the values using the write_imagef function which takes an int2 specifying the coordinates of the point and a float4 which holds the data to be stored. The coordinates to store the data has 0 for the y coordinate and the index number for the x coordinate. For the data that is stored, since the channel was chosen to be CL_A, only the w element is set to the value of the table while the x,y,z elements are set to 0. This kernel has n_tex_pts threads where each stores a single value in the image.

In order to read from the image, the index is first calculated from the squared distance value as described in section 4.3.1. This is stored into the x coordinate of a float2 and -1 into the y coordinate. The sampler is set to accept non-normalized numbers, clamping to the edge of the image, and a linear filter mode. The coordinate and sampler variables are fed to the image read function read_imagef which returns a float4. At this point, the potential energy value is stored in the w element and is returned to the QSATS program for processing.

Again, since there are very few examples, example code that illustrates how to use the texture units for hardware interpolation in OpenCL is available at [91].

This implementation gives the ability to run the code on NVIDIA and ATI GPUs, as well as AMD and Intel processors. This will be explored in the next chapter.

4.3.3 Multi-GPU Support

Modern supercomputers are increasingly adopting the use of GPUs to accelerate user applications while reducing the total power consumption. For this dissertation, the GPU implementation was created to take advantage of multiple GPUs through the use of MPI. This is accomplished by assigning each MPI process with a single GPU. Each GPU works independently from the others by performing a simulation with different random number generator seeds. Each MPI process will read/write its own save file and snapshot file. Exploiting parallelism in this way gives more statistical data on the end replicas which is helpful in some experiments. Since the GPUs

are independent from each other and each will have the same amount of work, it is presumed the speedup should be ideally linear with the number of GPUs used.

Chapter 5

Discussion of Results

This chapter provides a discussion of results from the the VPI implementations described in sections 3 and 4. Runs were performed on a variety of architectures ranging from multiprocessors, Intel MIC cards, and graphics processing units.

For the runs in this chapter, four different processors were used. Table 5.1 lists the processors used in this chapter to obtain results. Likewise, table 5.2 lists the GPUs used to explore the GPU implementation.

Unless otherwise stated, all code was compiled with the -O3 optimization flag and the corresponding architecture-specific flags.

Error bars on some of the graphs indicate the standard deviation of the runtimes of ten runs. Runtime is measured using the gettimeofday C function. To compare implementations, two metrics are used: speedup and efficiency. Speedup is calculated as the base time divided by the optimized time where the larger the speedup factor, the better. Efficiency is how well the program scales as the number of cores used are

Table 5.1: Processors and compilers tested

Processor	Referenced as	Compiler
Intel Core i7 920	Nehalem	gcc v4.4.5
Intel E5-2680	Sandy Bridge	icc v12.1.4
AMD Opteron Istanbul	Istanbul	icc v12.1.2
AMD Opteron 6272 Interlagos	Interlagos	gcc v4.6.1

Table 5.2: GPUs tested

GPU	CUDA/OpenCL
NVIDIA Tesla C1060	CUDA 4.1/OpenCL 1.1
NVIDIA GeForce GTX 480	CUDA 4.1/OpenCL 1.1
NVIDIA Tesla M2090	CUDA 4.1/OpenCL 1.1
ATI Radeon 7970	OpenCL 1.2

increased, where a value of one is ideal. Both of these metrics are shown in equations 5.1 and 5.2.

$$\text{Speedup} = \frac{T_{base}}{T_{opt}} \tag{5.1}$$

$$\text{Efficiency} = \frac{Speedup}{Cores} \tag{5.2}$$

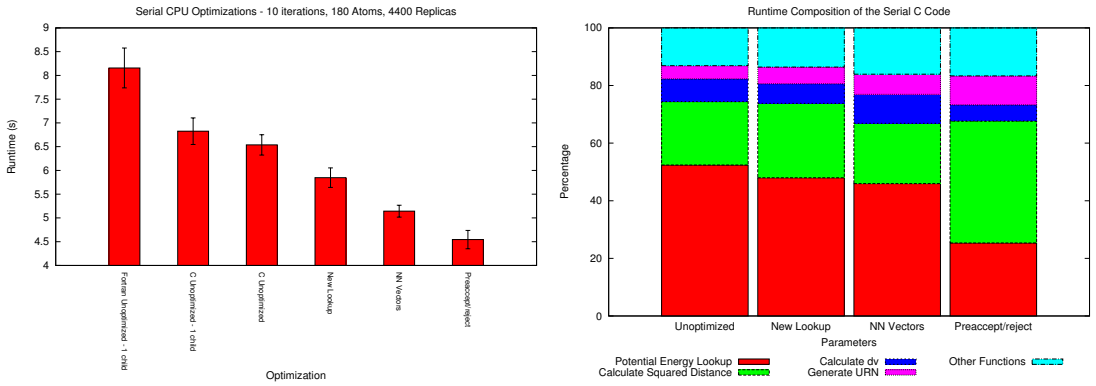
Many parameter variations are necessary to satisfy replica configuration requirements mentioned in chapters 3 and 4, performance is described as the rate that replicas, atoms, and iterations can be executed per second:

$$\text{Performance} = \frac{R * A * I}{time} \tag{5.3}$$

5.1 Microprocessor Results

5.1.1 Serial Optimizations

In order to test the microprocessor implementation, the effect of the optimizations described in section 3.3.1 is shown. The first step in the microprocessor code was to translate the original Fortran code to C. While this is not necessarily an optimization, it does provide a significant speedup without changing the algorithm. The main algorithmic optimization is the reduction of potential energy calculations from the lookup table as described in section 3.3.1. Since the Fortran code is written only with MPI support (i.e., no serial implementation), the code was run with two MPI processes: one parent and one child. To be fair, the MPI C code without these optimizations is compared to show the improvements over the Fortran code using one



(a) Runtimes **(b)** Runtime composition

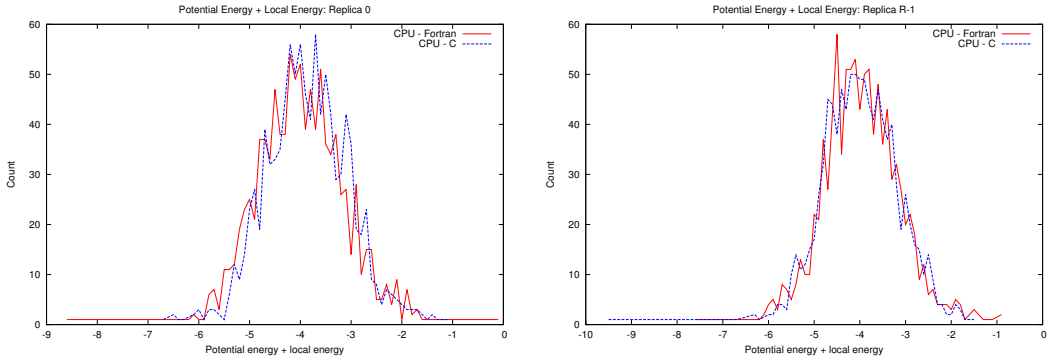
Figure 5.1: Serial optimizations

master and one worker as well just to show the impact of using the MPI version. Figure 5.1 shows the results of these optimizations running on the Intel Sandy Bridge processor for 10 iterations, 180 atoms, and 4400 replicas. It can be seen that simply changing from Fortran to C provides a decent performance decrease. The C implementation is 1.19x faster than the Fortran version without any optimizations. Nothing special is happening here besides flattening two dimensional Fortran arrays to one dimensional C arrays. This provides for better caching performance and better compiler optimizations. Using the new potential energy lookup table results in a 1.12x over the unoptimized C implementation and a 1.39x over the Fortran implementation by reducing the number of double precision floating point operations. Maintaining a list of the nearest neighbor vectors provides a 1.27x improvement over the unoptimized C code and 1.59x over the Fortran code by decreasing the number of L1 misses by 50% as reported by PAPI [93]. Decreasing the number of potential energy lookups provides for a 1.43x performance increase over the unoptimized C implementation and 1.79x over the Fortran implementation. The contribution of each function for each of the optimizations is given in figure 5.1b.

From this point forward, any speedups are given relative to the optimized C implementation with the reduced table lookups as this is the fastest serial implementation.

Table 5.3: Parameters used for correctness tests

Parameter	Value
Atoms	180
Replicas	46,080
$\Delta\tau$	500
ρ	0.00461421
Mass	7,294.3
α	0.15
B	5.4
P	56
P_I	18
I_{warmup}	10,000
I	100,000



(a) First replica **(b)** Last replica

Figure 5.2: Check for microprocessor C statistical correctness

Since there are some algorithmic changes in the code such as when random numbers are generated, the output of the C code is different than the Fortran code. Therefore, it is necessary to check the statistical correctness. Figure 5.2 shows a histogram of the sum of the potential energy and local energy for each of the snapshots of the first and last replica using the parameters in table 5.3. From the figure, it can be seen that the results are statistically equivalent to the Fortran code as they share the same mean and the same standard deviation.

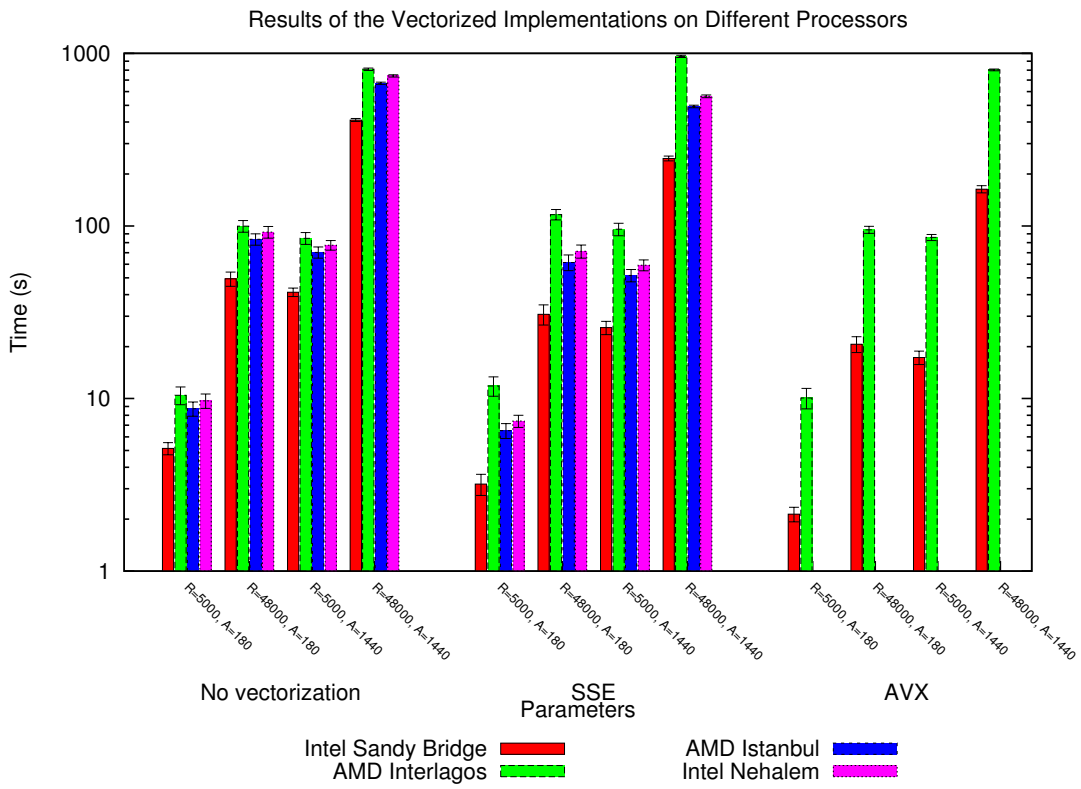


Figure 5.3: Vectorization runtime results

5.1.2 Vectorization

The next optimization that needs to be tested is the parallelization of the QSATS algorithm using SSE and AVX vectorization units. In order to test this, both of the vectorization implementations are run on various architectures to obtain runtimes. All of the processors tested have SSE capabilities; however, only two have AVX units: Intel E5-2680 and AMD Interlagos. Figure 5.3 shows the runtimes of the vectorization implementations on the processors listed in table 5.1.

The speedup factors can be seen in table 5.4. From the figure and table, it can be seen that using the vector units to accelerate the QSATS algorithm is a beneficial optimization for most processors. Speedups are not in line with the number of doubles the vector can hold (i.e., 2x speedup using SSE and 4x with AVX) due to the serialization of table lookups and neighbor information references because

Table 5.4: Speedups for the vectorized microprocessor implementation

Parameters	Sandy Bridge		Interlagos		Istanbul	Nehalem
	SSE	AVX	SSE	AVX	SSE	SSE
R=5,000, A=180	1.61	2.40	0.88	1.04	1.34	1.31
R=48,000, A=180	1.61	2.39	0.86	1.05	1.36	1.29
R=5,000, A=1,440	1.60	2.39	0.89	0.99	1.36	1.30
R=48,000, A=1,440	1.67	2.51	0.84	1.01	1.36	1.31

it is not guaranteed that every memory accesses are not to consecutive memory locations. Despite this serialization, all of the processors show an improvement with the exeception of the AMD Interlagos processor where the SSE implementation is actually slower than the serial and the AVX is approximately the same speed as the serial. Two Interlagos processor cores share a single vector unit potentially causing this slow down. Another possibility could be that the gcc compiler is not able to optimize the intrinsics well enough to obtain as much performance as the other processors.

In regards to programmability, converting this code to use the vectorization units was difficult for the SSE implementation; however, supporting the other types of vectorization was fairly straightforward especially if coded in a flexible manner. For this code, #define macro wrappers over the common functions are used so the code can be easily ported to a different set of intrinsics. For instance, the addition of two double precision vectors is performed by calling the _mm_add_pd(x,y) function in SSE and _mm256_add_pd(x,y) in AVX. Each of these can be called by a simple macro call to add(x,y) where this macro is set appropriately at compile time based off flags that are supplied. Using this approach for all of the functions used in the QSATS programs allows for the implementation to share much of the same code. It also comes with the added benefit of increasing code readability.

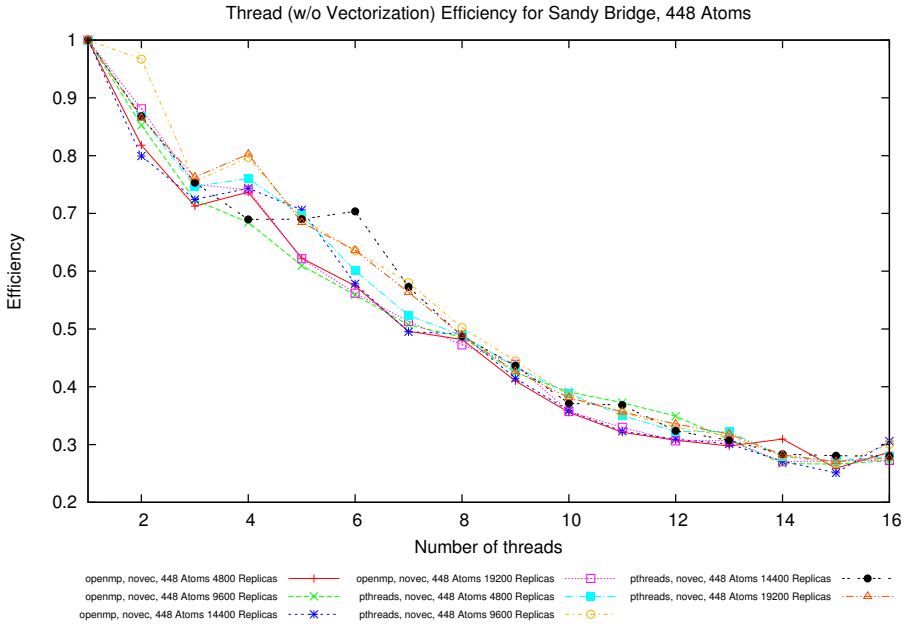
5.1.3 Threads

As mentioned in section 3.3.3, a number of threaded versions have been implemented: pthreads, OpenMP, and OpenCL. Implementation details for each of these are provided in section 3.3.3. Figure 5.4-5.7 give the runtimes of the OpenMP and pthreads implementations on some of the processors listed in table 5.1.

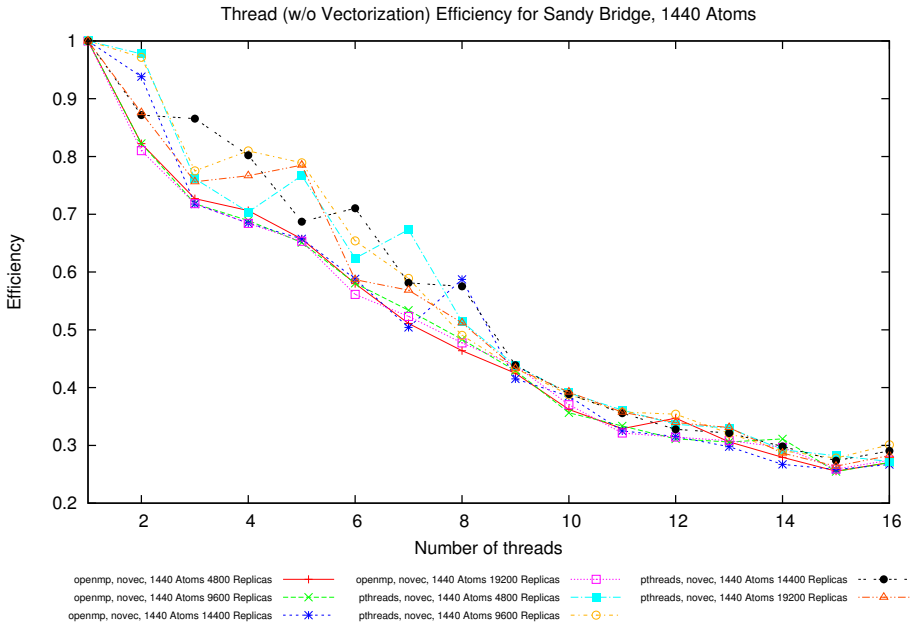
From the plots in the figures, it can be seen that the threaded implementation does not provide a high efficiency due to a large amount of data that each thread must handle. Each thread's data interferes with the other threads' data movement. For the Sandy Bridge machine, the efficiency drops to 60% when using half of the available cores (8) and 30% when using all of the cores (16). For the Interlagos processors, the efficiency drops to 60% when using half of the cores (16) and 45% when using all of the cores (32). For the Nehalem processors, the efficiency was around 90% when using half of the cores (4) and 70% when using all cores (8). For the Istanbul processors, the efficiency drops significantly to 20% when using only 2 cores, indicating poor scalability when using the threaded implementation.

When comparing OpenMP and pthreads, there does not appear to be much performance difference. The pthreads implementation is negligibly faster in most cases with the exception of using all cores. This is due to having a "master" thread that is in a spin-lock waiting for the "worker" threads to complete, thus consuming compute cycles and resources. When using enough worker threads to occupy all available cores, the master thread is scheduled to run on top of a core that a worker thread is using causing the core to have to context switch between the two threads slowing down the entire simulation. It would be possible to make the master thread double as a worker thread; however, this would involve more coding for correctly handling the orchestration of the threads. The OpenMP implementation does not have this pitfall as the library handles this already.

Although threading is not an effective parallelization technique for this application, vectorized threaded implementations were developed. Since the serial vectorized

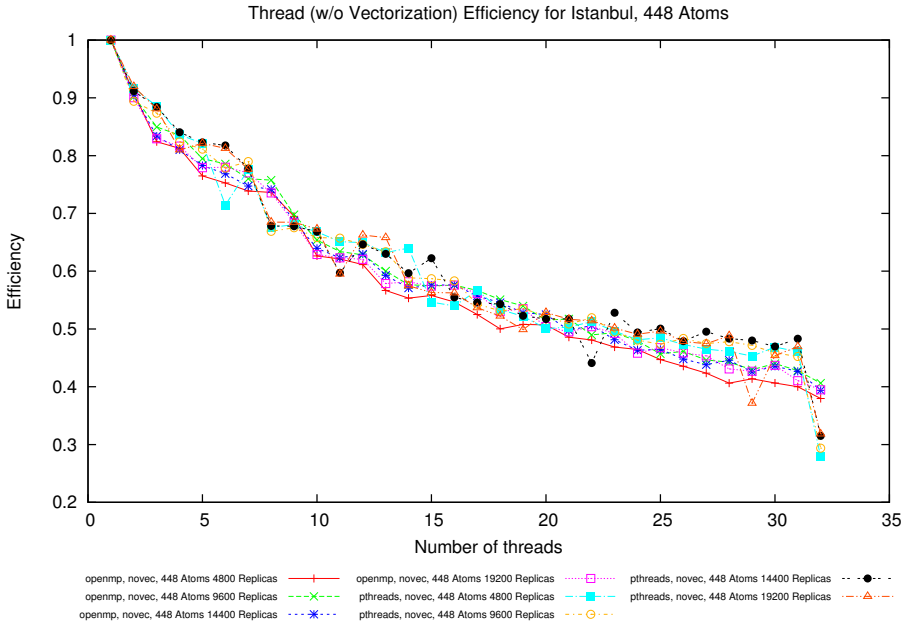


(a) 448 Atoms

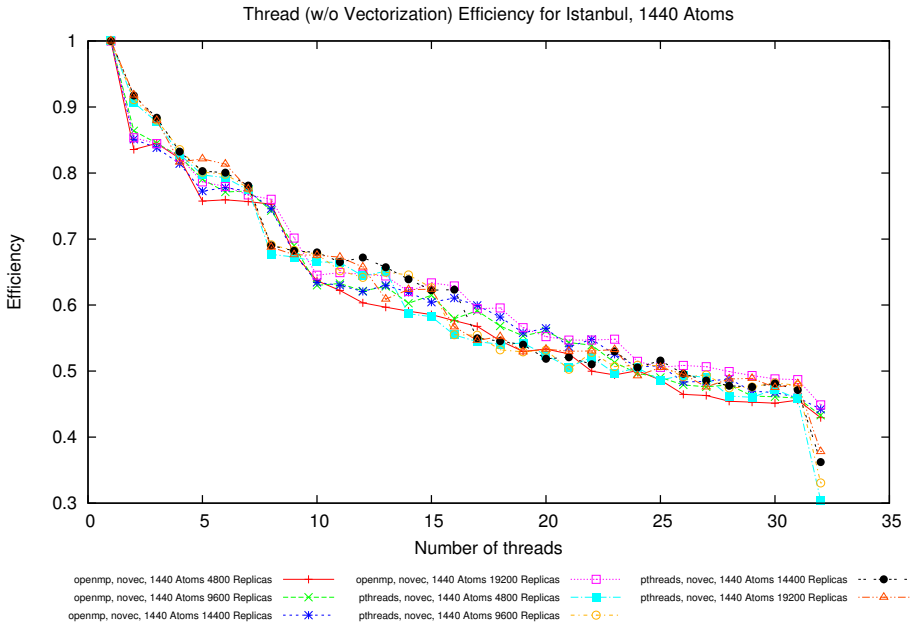


(b) 1440 Atoms

Figure 5.4: Threaded results for the Sandy Bridge processor

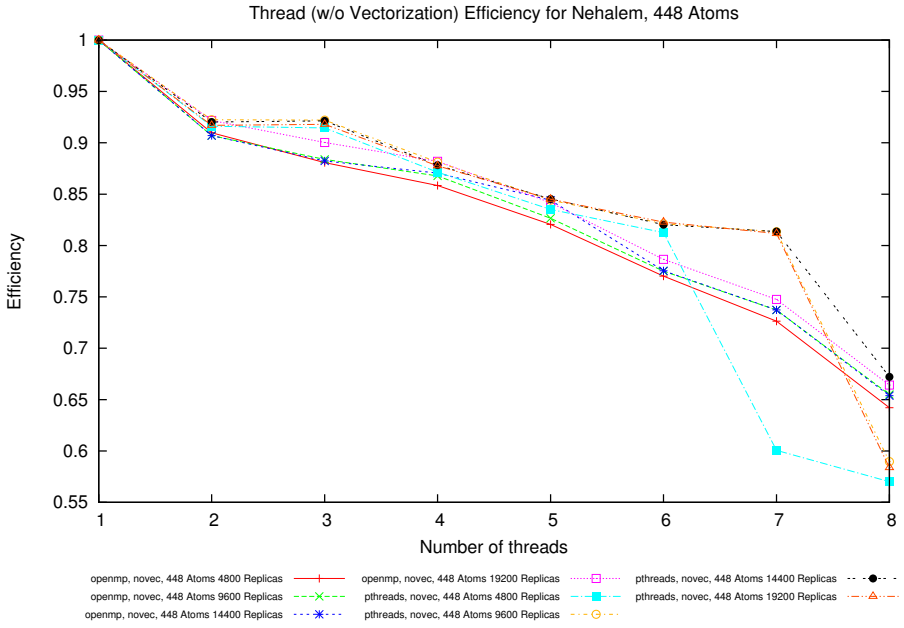


(a) 448 Atoms - Interlagos

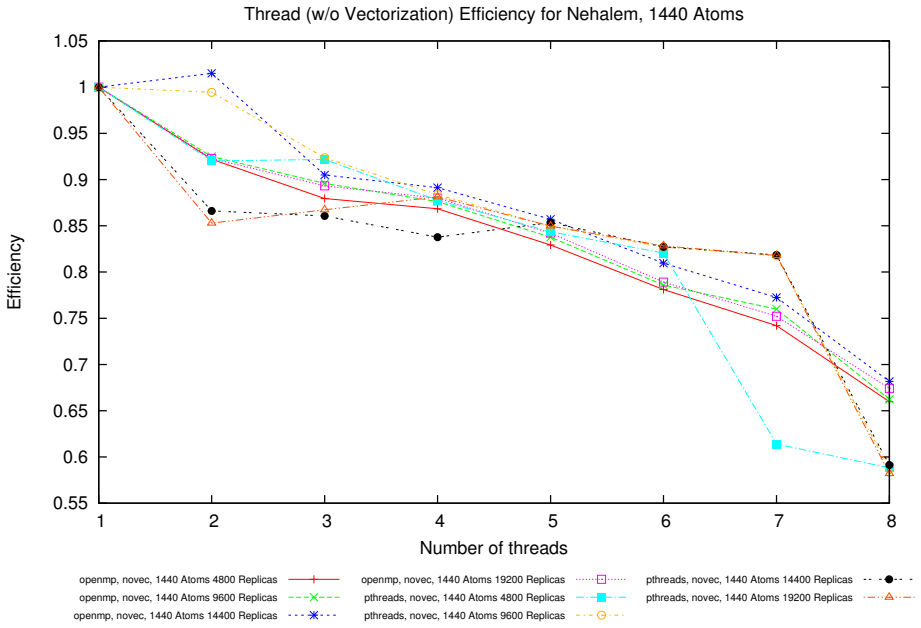


(b) 1440 Atoms - Interlagos

Figure 5.5: Threaded results for the Interlagos processor

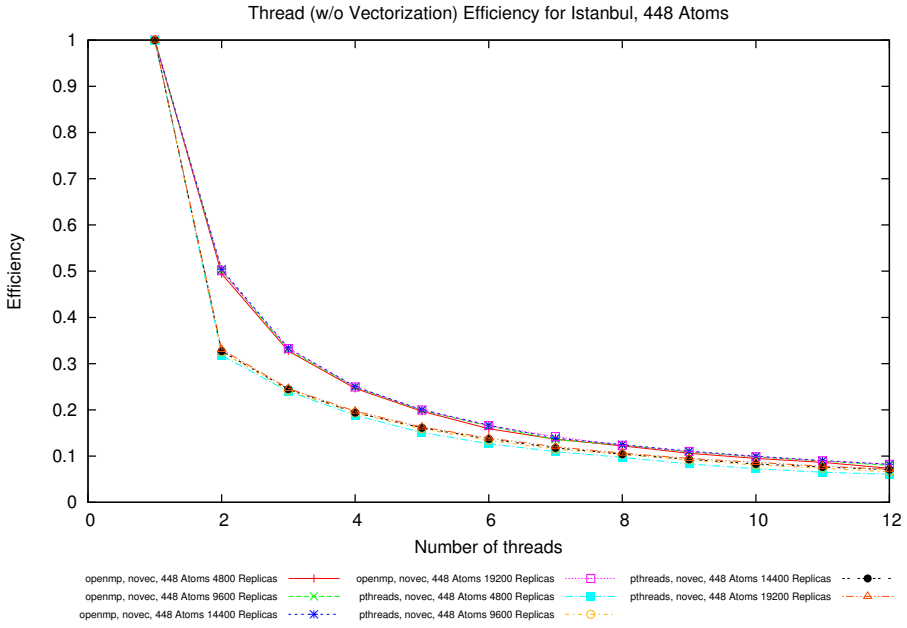


(a) 448 Atoms - Nehalem

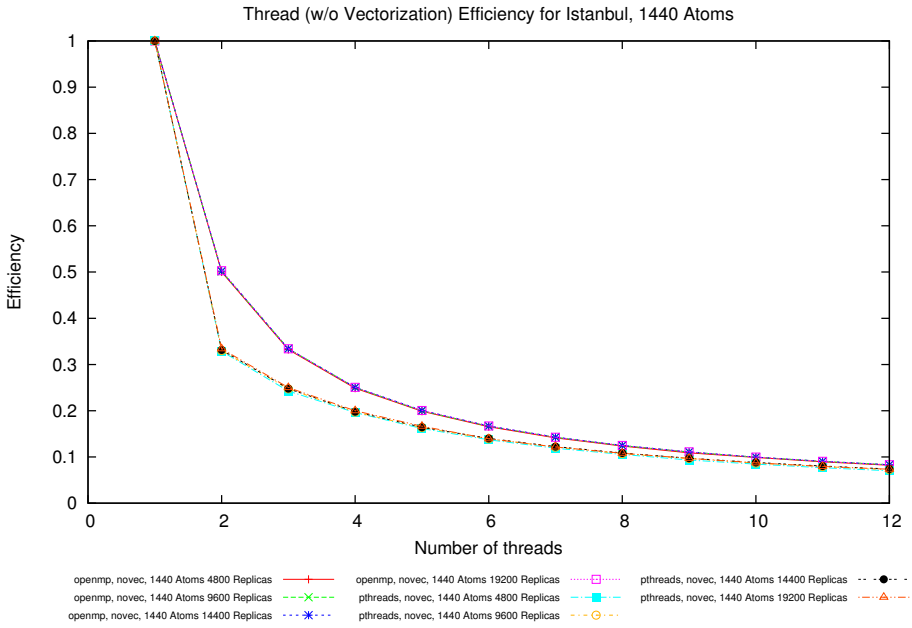


(b) 1440 Atoms - Nehalem

Figure 5.6: Threaded results for the Nehalem processor



(a) 448 Atoms - Istanbul



(b) 1440 Atoms - Istanbul

Figure 5.7: Threaded results for the Istanbul processor

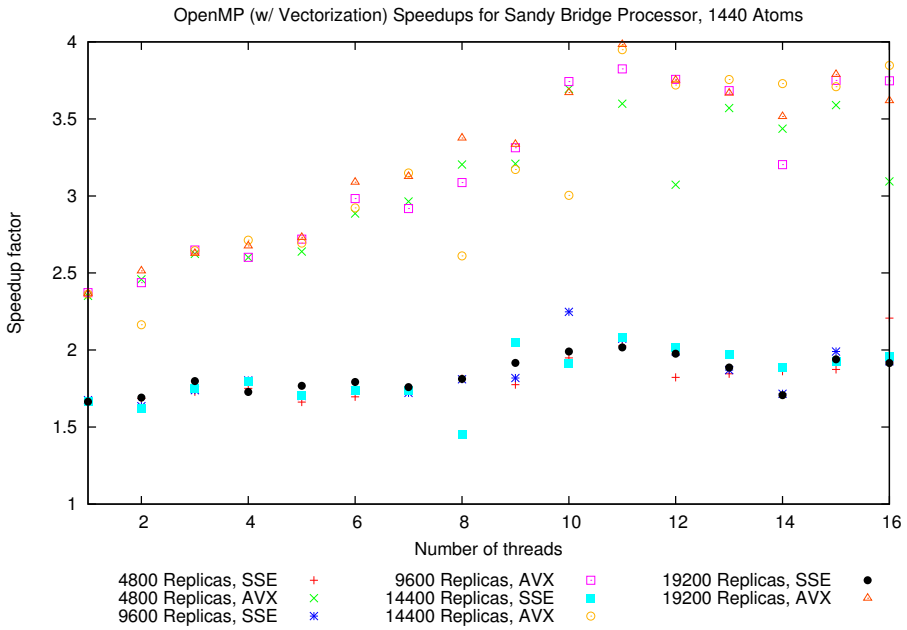


Figure 5.8: OpenMP with vectorization speedups for the Sandy Bridge processor

versions were already created, extending the threaded implementations to use the vector units was a straightforward addition. Figures 5.8-5.11 gives the speedups of some sample runs using the vectorized implementation using 1440 atoms and OpenMP.

From these plots, we can see that vectorization speeds up the threaded implementation dramatically as the number of threads increase for both Intel processors. However, due to the reasons specified before, the AMD Interlagos gets poor performance and the speedup of using the vector units varies significantly as the number of threads increases, not to mention the speedups are no better than around 1. Using SSE vectorization with the Istanbul processor provides a pretty consistent speedup of around 1.35x. To see what effect vectorization has on the efficiency of the threaded implementation relative to a single vectorized thread run, the plots in figures 5.12-5.15 shows the efficiency with a varying number of replicas and atoms on the Sandy Bridge, Interlagos, and Nehalem processors. From these plots it can be seen that adding vectorization increases the efficiency by increasing the

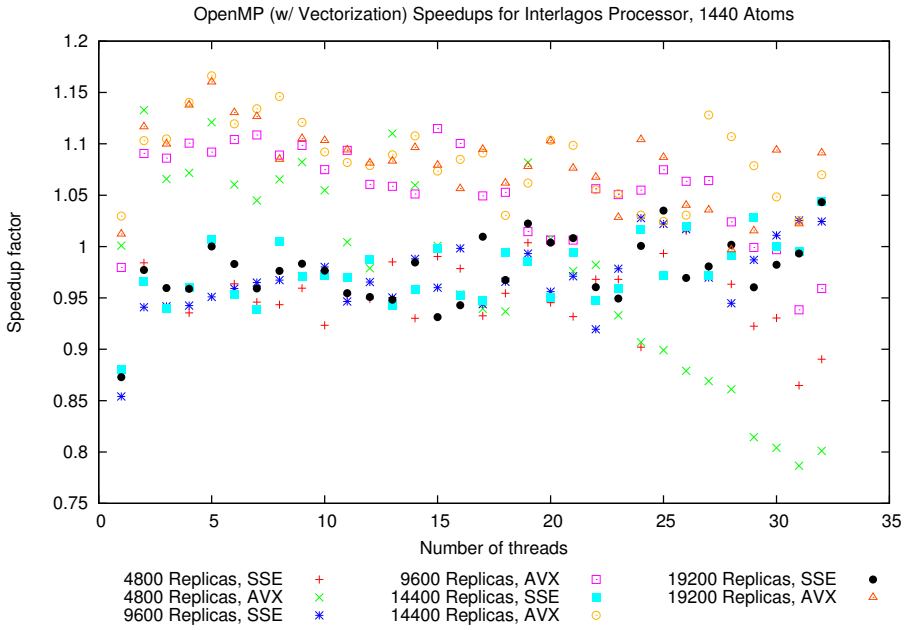


Figure 5.9: OpenMP with vectorization speedups for the Interlagos processor

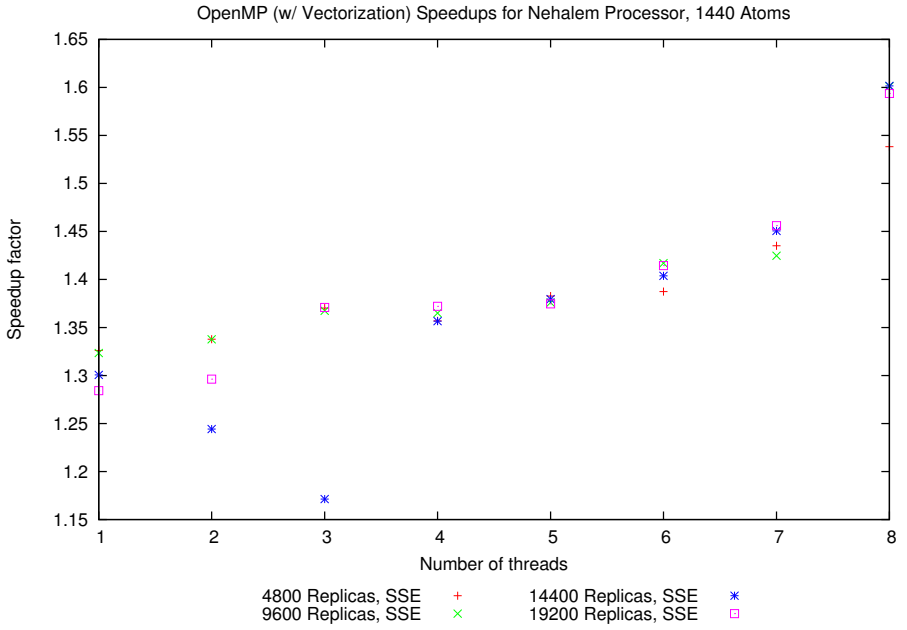


Figure 5.10: OpenMP with vectorization speedups for the Nehalem processor

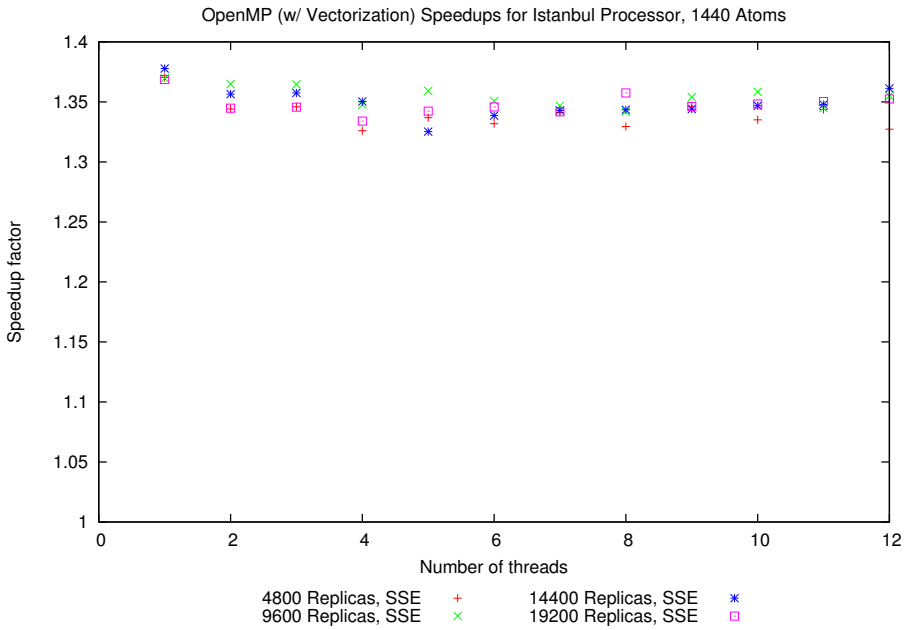
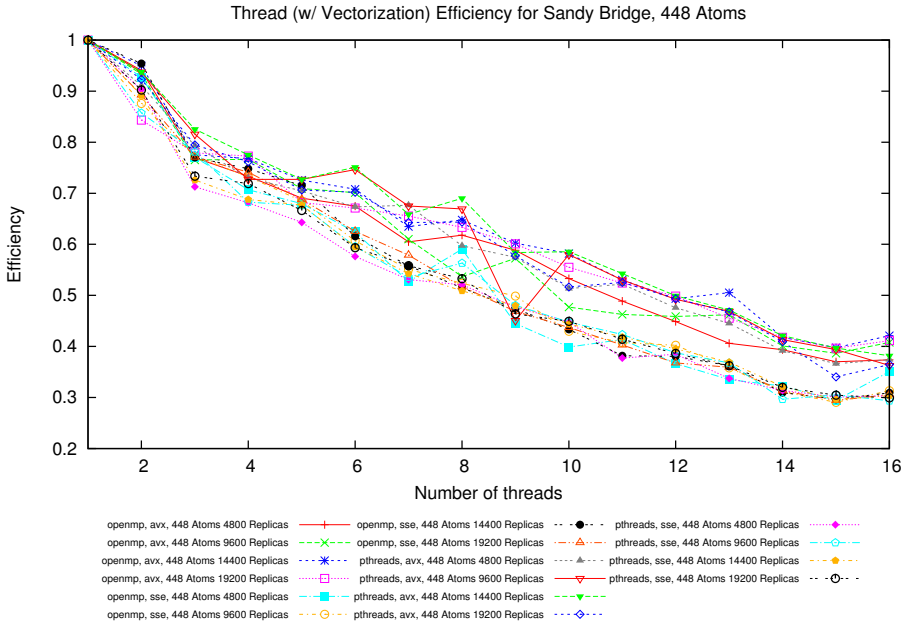


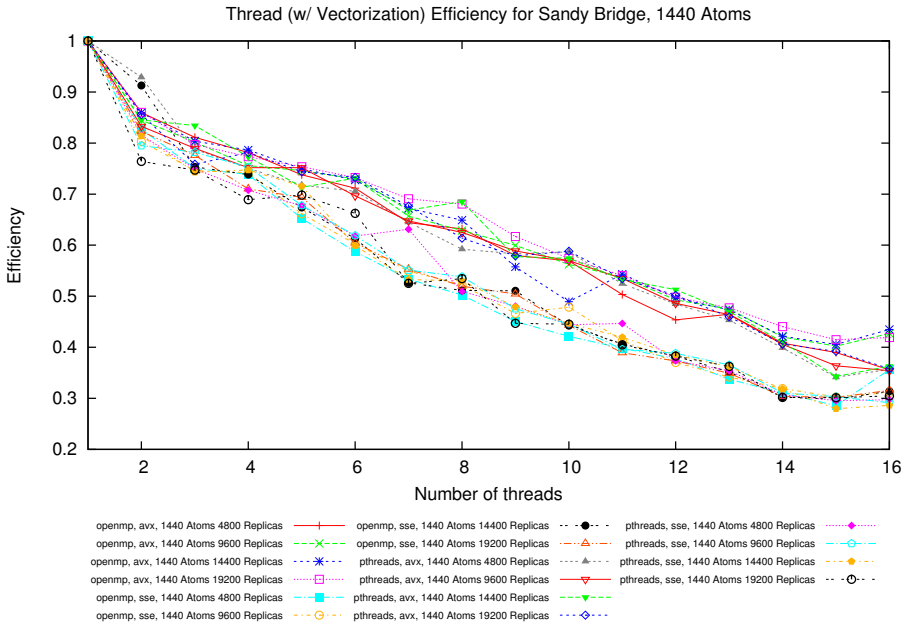
Figure 5.11: OpenMP with vectorization speedups for the Istanbul processor

number of operations that can be performed simultaneously and reducing the total number of instructions. For instance, with the Intel Sandy Bridge processor, the efficiency doubled from the non-vectorized implementation when using all available cores. However, even with this increase, obtaining 60% efficiency is not acceptable so another approach must be used.

In terms of programmability, the OpenMP implementation was trivial to write as it mainly involved adding the following pragma: #define parallel for on the odd and even for loops. At this point, the number of threads can be set by calling a function and the library will take care of the rest. Once a thread has been created, it is added to a pool of threads that OpenMP manages so that the application isn't creating and destroying threads through the runtime of the program. However, the pthreads implementation was much more involved as a thread pool implementation had to be developed. Within this thread pool, code is available to send a "wake up" signal to the worker threads and to send an "I'm finished" message to the master thread to allow for synchronization. In terms of programming time, the OpenMP

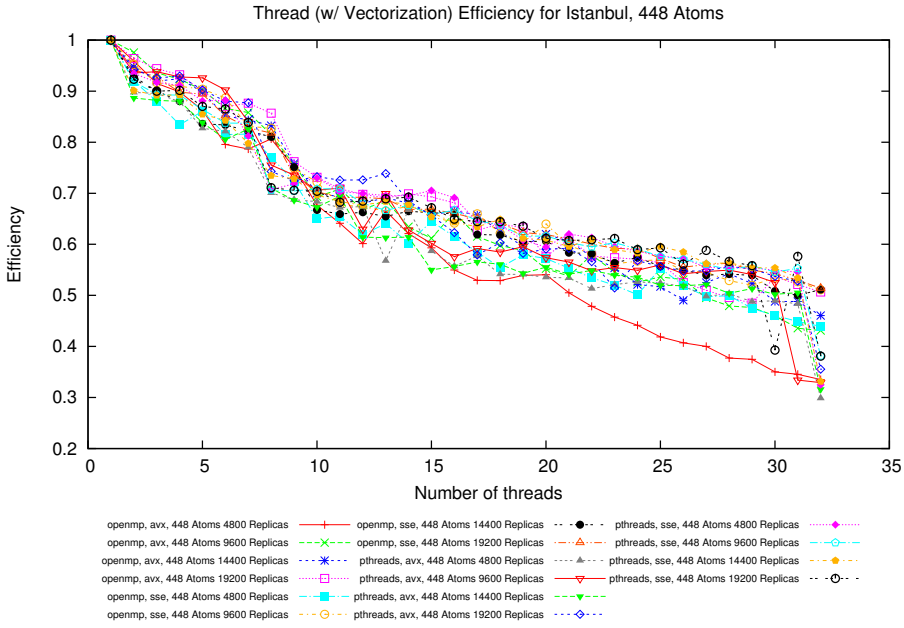


(a) 448 Atoms - Sandy Bridge

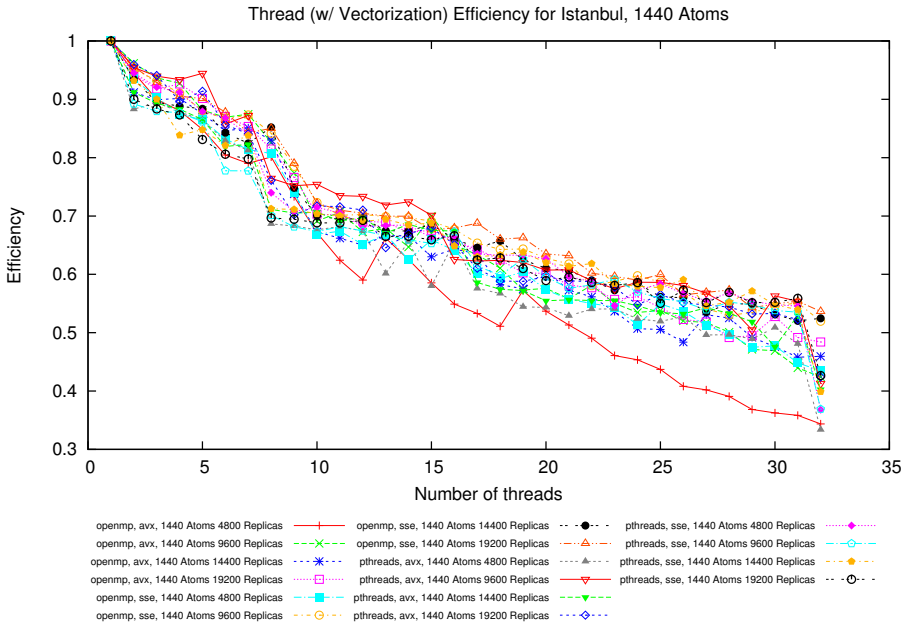


(b) 1440 Atoms - Sandy Bridge

Figure 5.12: Threaded vectorized results for the Sandy Bridge processors

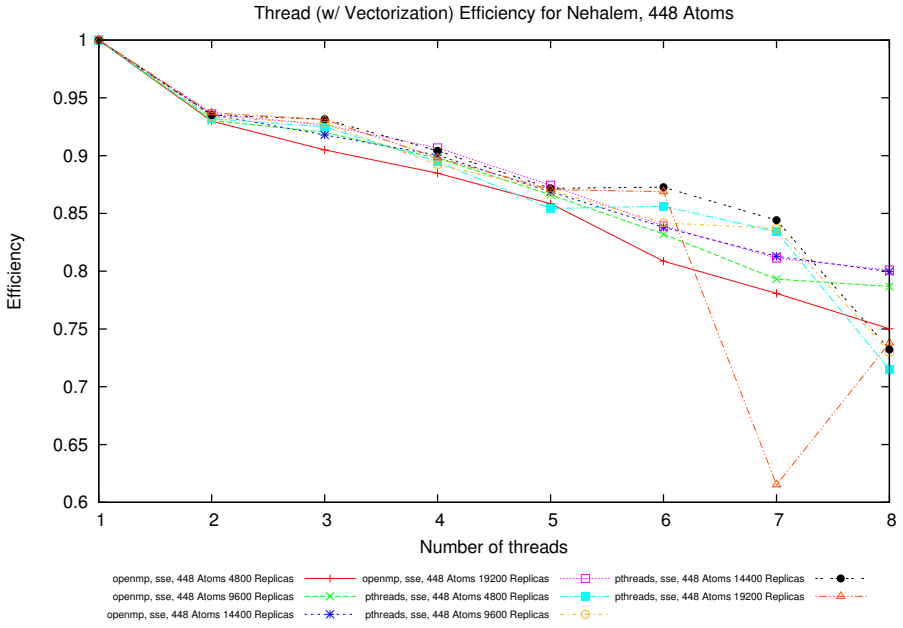


(a) 448 Atoms - Interlagos

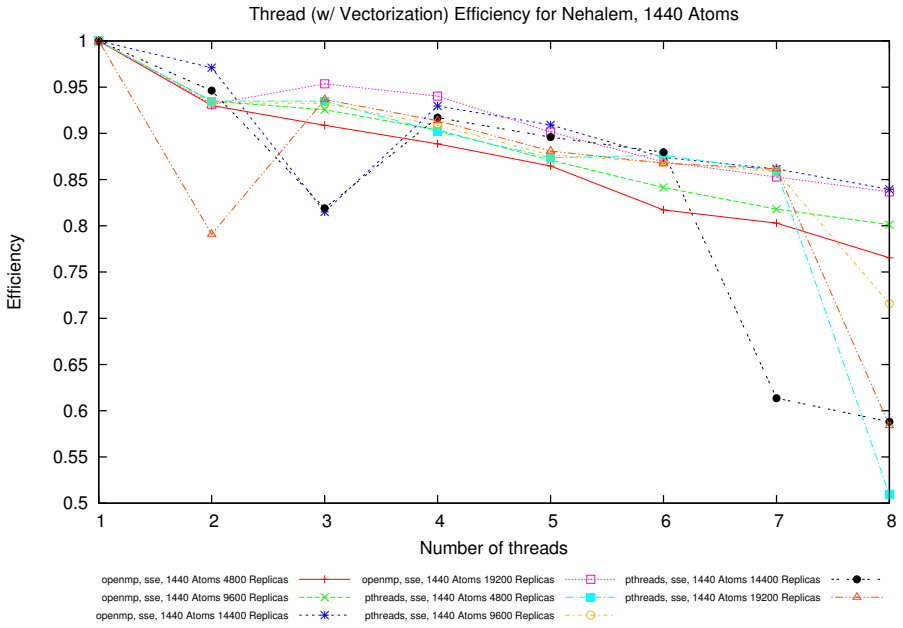


(b) 1440 Atoms - Interlagos

Figure 5.13: Threaded vectorized results for the Interlagos processors

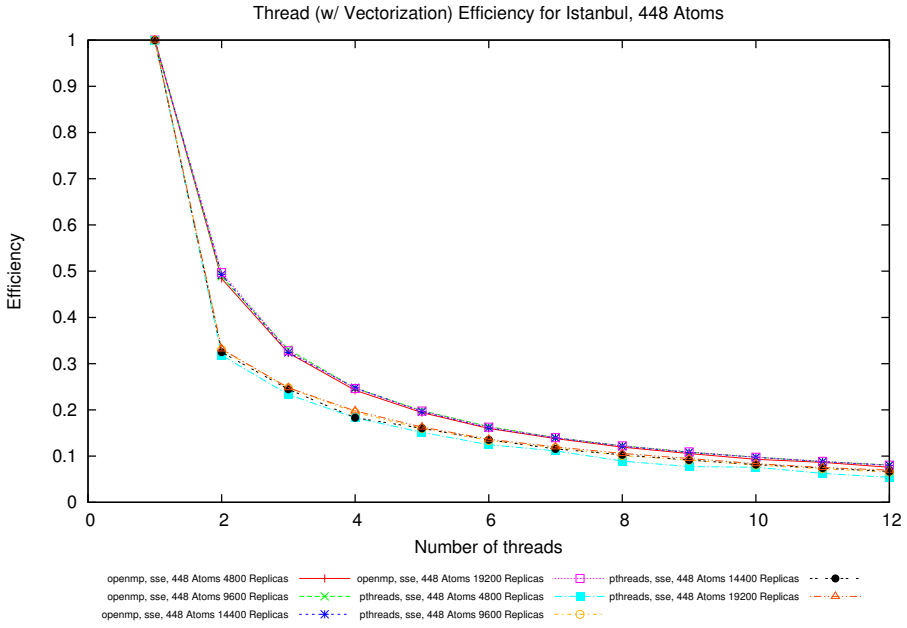


(a) 448 Atoms - Nehalem

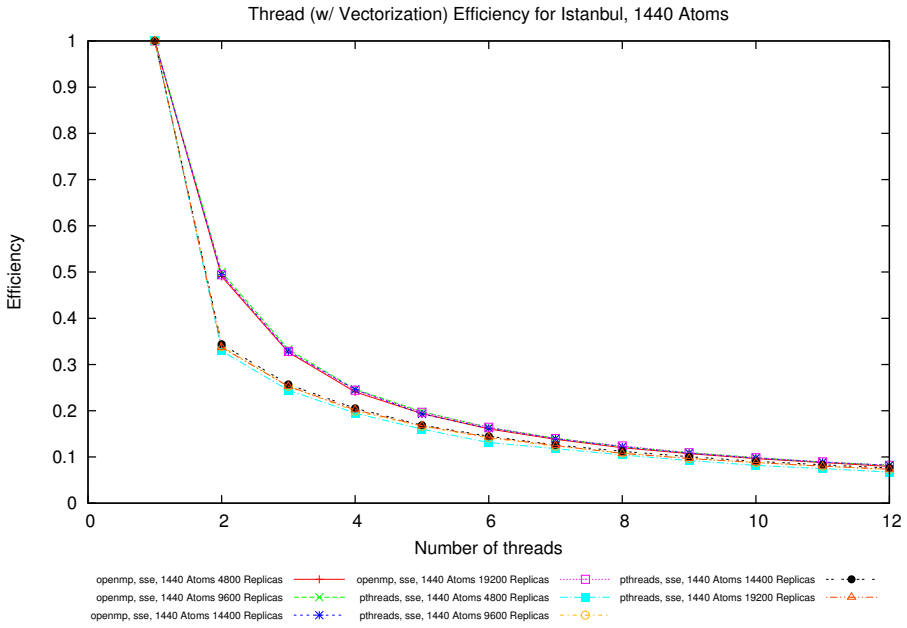


(b) 1440 Atoms - Nehalem

Figure 5.14: Threaded vectorized results for the Nehalem processors



(a) 448 Atoms - Istanbul



(b) 1440 Atoms - Istanbul

Figure 5.15: Threaded vectorized results for the Istanbul processors

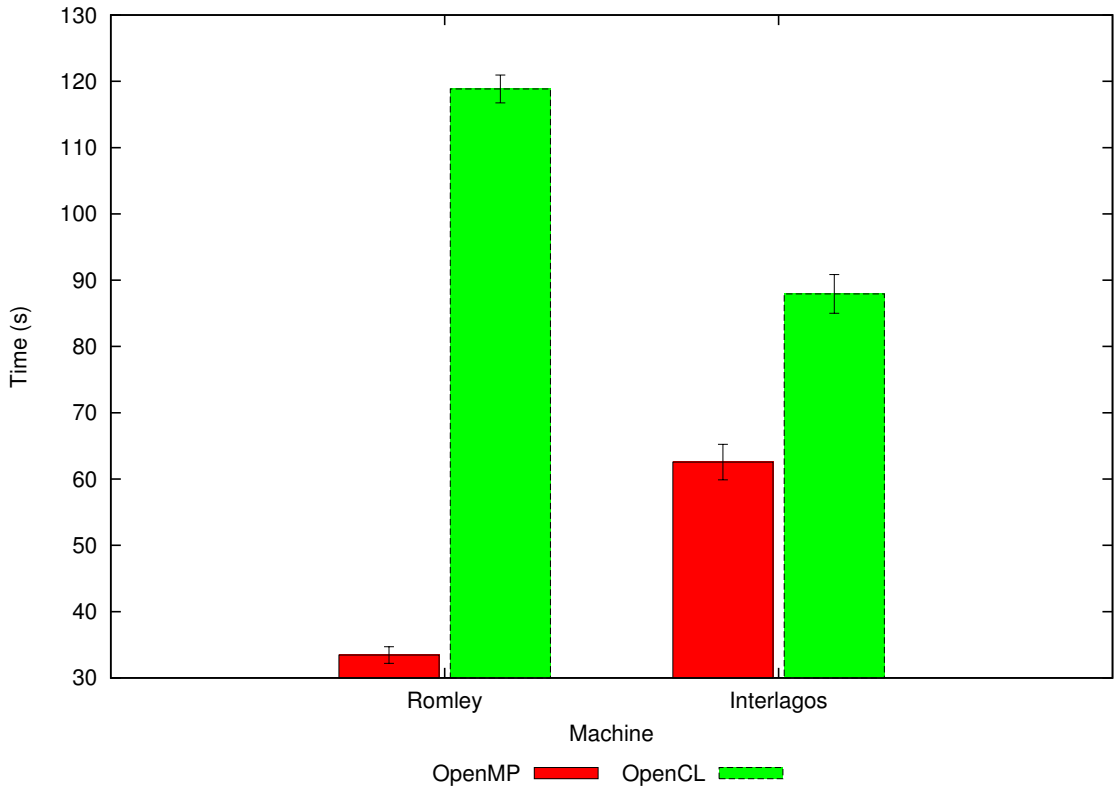


Figure 5.16: OpenCL comparison to OpenMP threaded multiprocessor code

implementation took less than half an hour to write and verify whereas the pthreads took a couple of days in order to develop and test the thread pool code.

In comparison to the OpenCL implementation, though, the OpenMP implementations perform better. Figure 5.16 compares the OpenCL implementation on two processors each using OpenCL version 1.2. These runs used 46,080 replicas, 180 atoms and 100 iterations. The OpenMP code used 16 threads. The OpenCL used a work-group size of one since a work-group is assigned to a core. One interesting thing to note here is that the AMD OpenCL implementation performed much better than the Intel implementation; however, the OpenMP code is faster on the Intel processor. This could be due to differences in OpenCL vendor implementation.

In terms of programmability, OpenCL was very straightforward as the GPU implementation was written first. However, to code the OpenCL implementation for the multiprocessor from scratch would be very time consuming unless the programmer

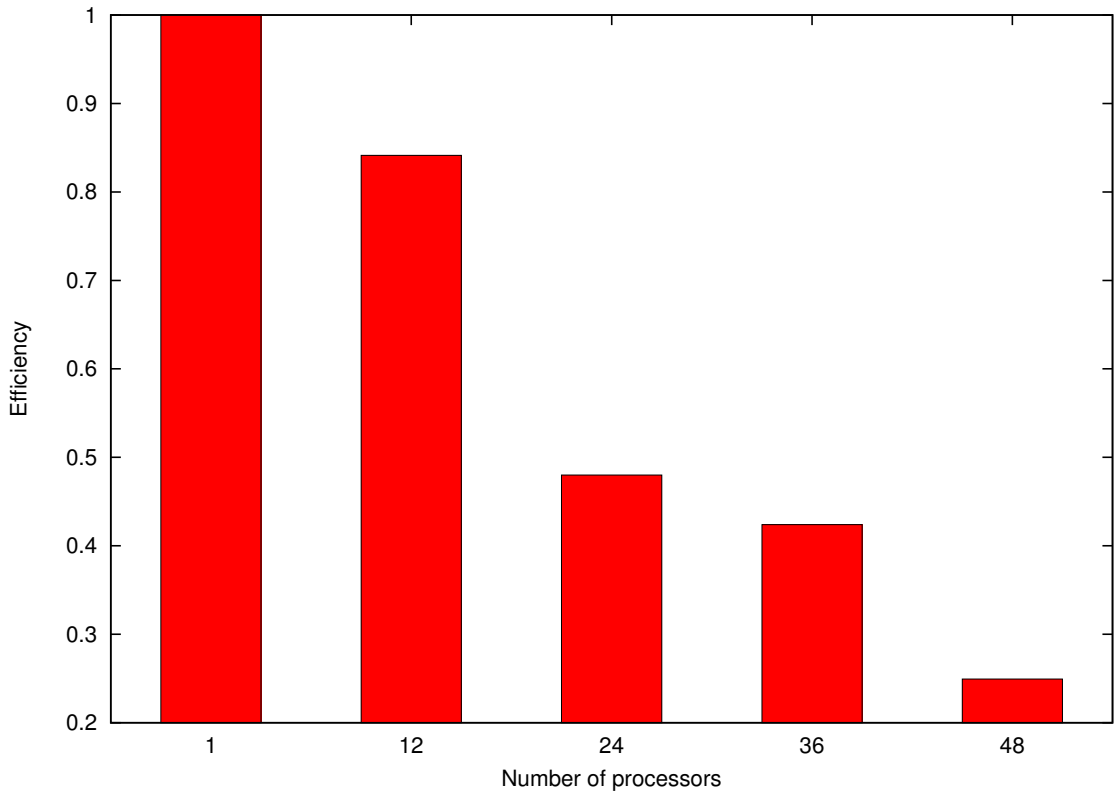


Figure 5.17: Efficiency of the single master - independent worker code

is familiar with the OpenCL API or uses a library that simplifies the process such as clUtil [94]. Once the code is written to initialize OpenCL and all the things associated with it, writing the kernel is very straightforward for a programmer who is familiar with CUDA as it is mostly just keyword changes.

5.1.4 MPI

Master-Independent Workers

As described in section 3.3.4, the original MPI implementation used a typical master-worker model to distribute work. While this is a very straightforward implementation and does take care of load balancing, it is not a very scalable approach. Figure 5.17 shows the efficiency of this implementation using 103202 replicas, 10 iterations, and 180 atoms running on the Istanbul processors in Kraken [95]. This implementation is

not able to scale beyond a small number of cores. The lack of scalability is attributed to the master's inability to successfully manage the requests from all of the children, causing the children to stay idle for long periods of time. Because of the extreme lack of scalability, this implementation is not suitable for large crystal simulations with many replicas.

Single Master-Dependent Workers

Next, the single master, dependent workers implementation seeks to increase the scalability beyond a few nodes. This is accomplished by pre-allocating work at the beginning of the simulation and allowing the workers to communicate with each other. Occasionally, the master will aggregate snapshot information from all of the children and writing it to a file while the children continue with their work. Figure 5.18 shows the runtime for a variety of parameters and numbers of cores.

Using a single master and dependent workers increased the scalability substantially. In comparison to the previous implementation, this approach reduces the total amount of communication significantly and reduces the load on the master since it is no longer a central process for the entirety of the simulation. From the plot, the efficiency starts to drop off at 960 cores as the time it takes to do a snapshot is the limiting factor. This is due to the master having to serialize snapshot communications from the workers thus increasing the time a child must wait before continuing with the simulation.

Multiple Masters-Dependent Workers

A third implementation adds the ability to have a variable number of masters where each master aggregates replica configurations from only a subset of the workers. For snapshotting or saving configurations, each master writes to its own file; therefore, there is no communication among the masters. With multiple masters writing to their own file, this implementation is able to fully take advantage of the parallel file systems

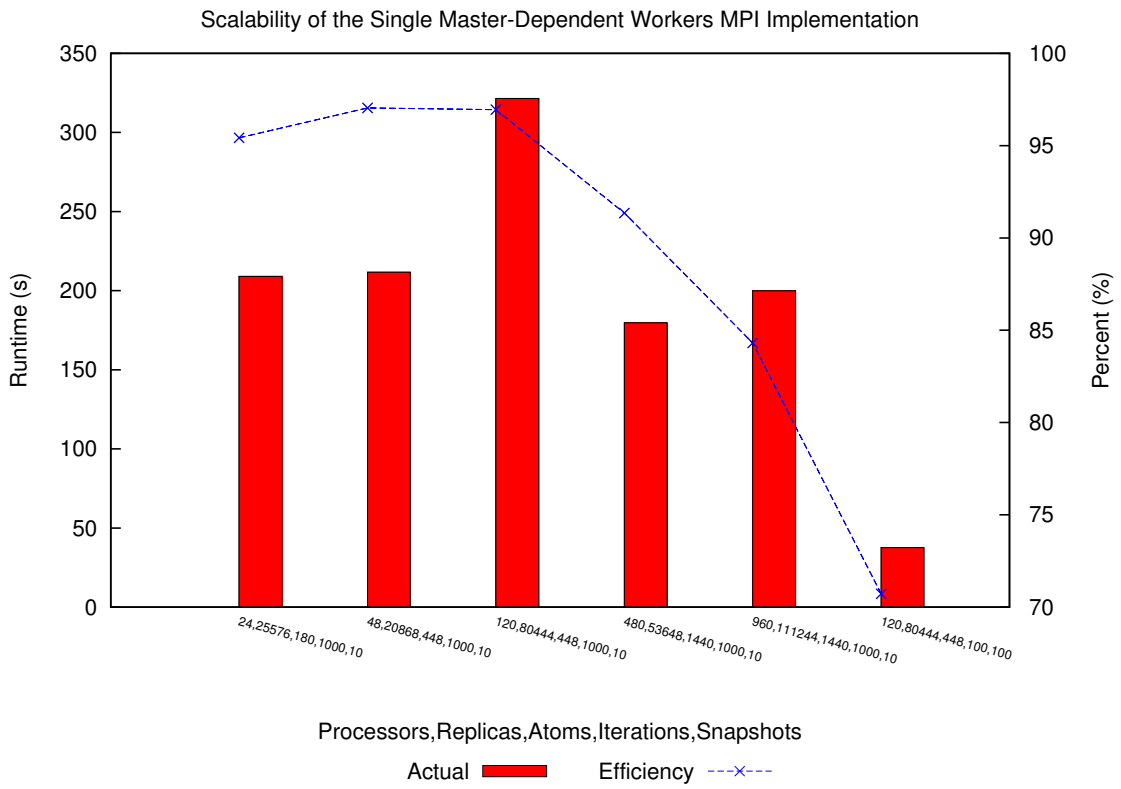


Figure 5.18: Single master performance rates and efficiency

(e.g., Lustre) on large machines. Section 3.3.4 gives a more in-depth explanation of this approach. Figure 5.19 shows the runtime for a variety of parameters and numbers of cores.

When comparing the multiple master approach to the single master, efficiency is significantly improved. This implementation achieves a 95% efficiency using 960 cores and 15 masters compared to 85% when using a single master. Efficiency stays around 90% for 3,840 cores. Efficiency drops to 85% using 7,668 cores and 71% using 15,300 cores. This drop is attributed to some issues with Kraken lowering the bandwidth between nodes significantly. To rule out algorithmic problems, a couple of jobs were run using 960 cores, 15 masters, and the same parameters as the run where 95% efficiency was obtained. Originally, the bandwidth from the workers to the master was around 1.5GB/s; however, the extra runs had a bandwidth ranging from 6MB/s to 30MB/s. By outputting the time it takes for one worker to send its packages to

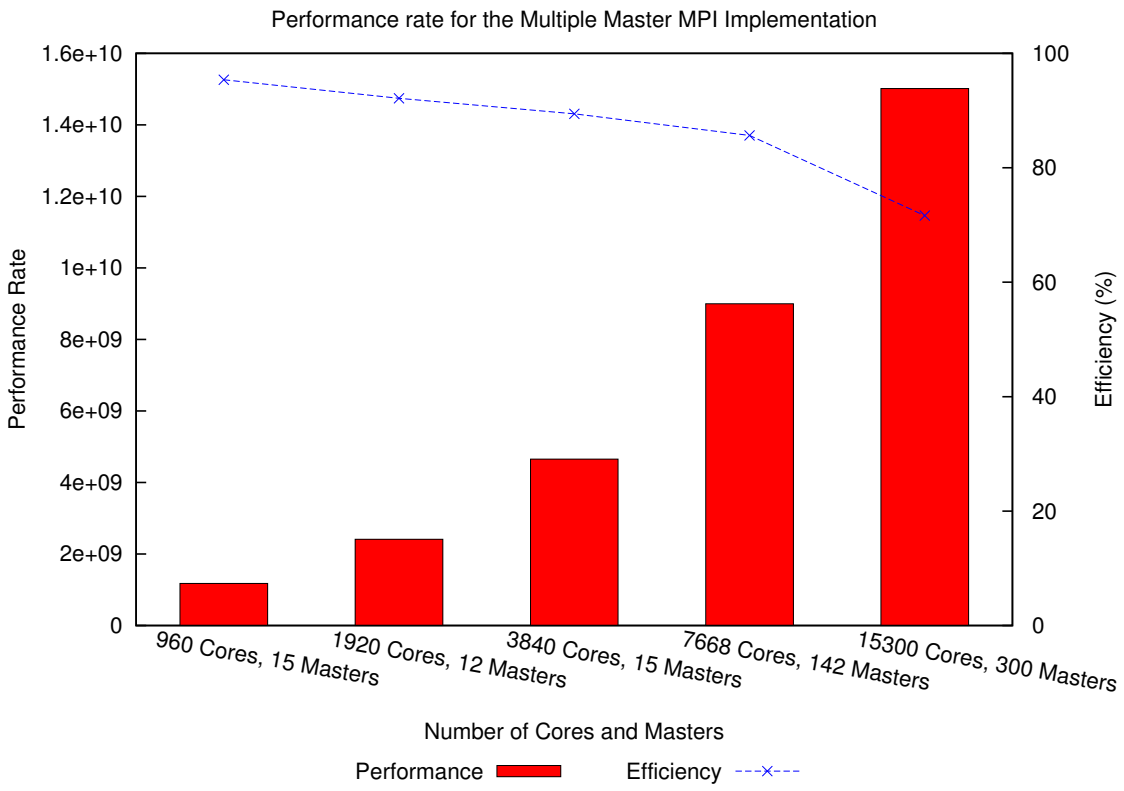


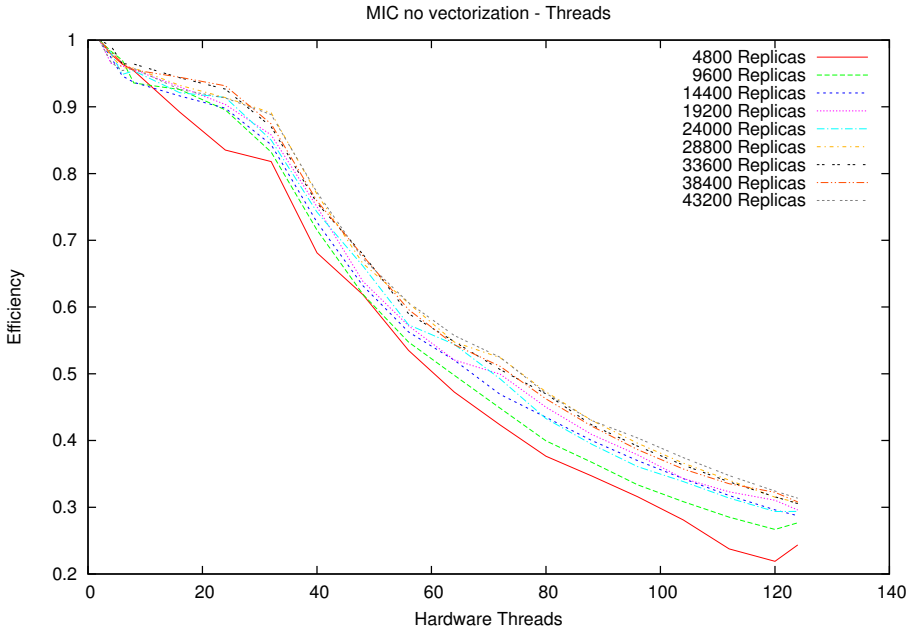
Figure 5.19: Results of the multiple masters implementation

the master, I was able to verify that bandwidths for the two larger runs fell within this range. Despite the results being skewed, this implementation still achieves an acceptable scalability to 7,668 cores.

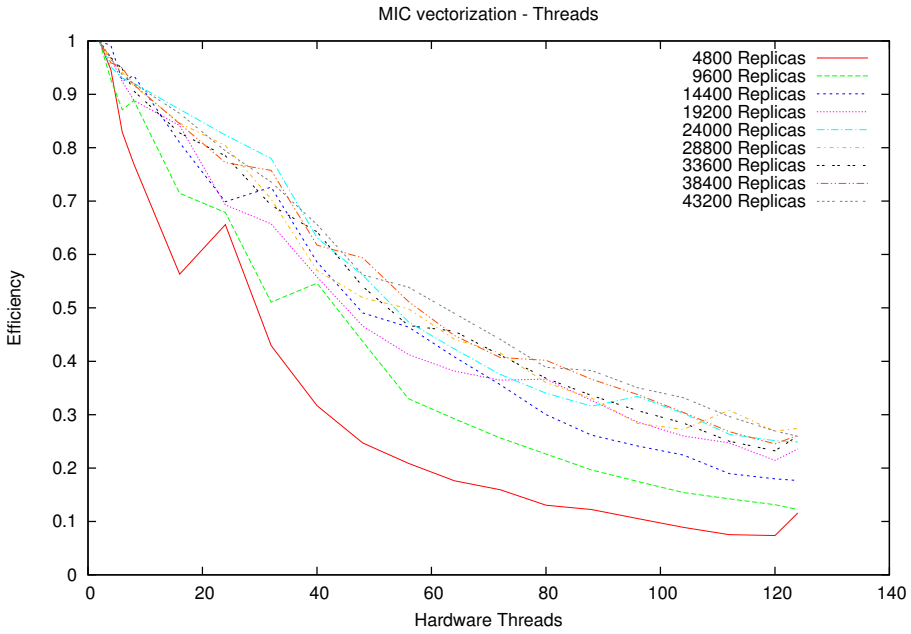
5.2 Intel MIC

Because this work with the Intel MIC is under a non-disclosure agreement, specific details, absolute numbers, or direct comparisons to other architectures are not given. However, the plots in figures 5.20 and 5.21 show the efficiency of using more cores on the Intel MIC for various numbers of replicas. Also, plots are given for the MPI and threaded implementations with and without vectorization.

The MPI implementation obtains a higher efficiency as the number of cores increases than the threaded implementation. However, the threaded implementation

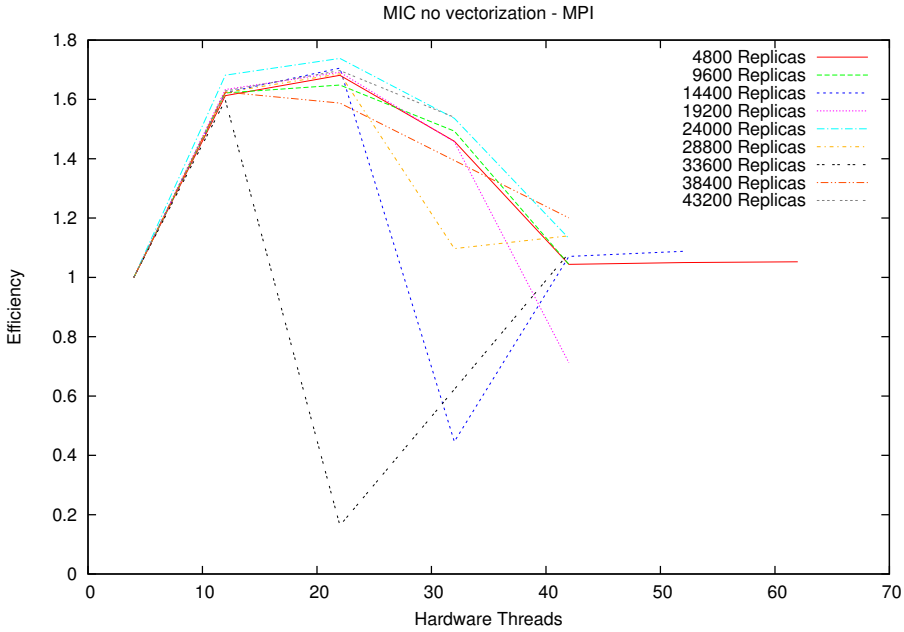


(a) Threaded without vectorization

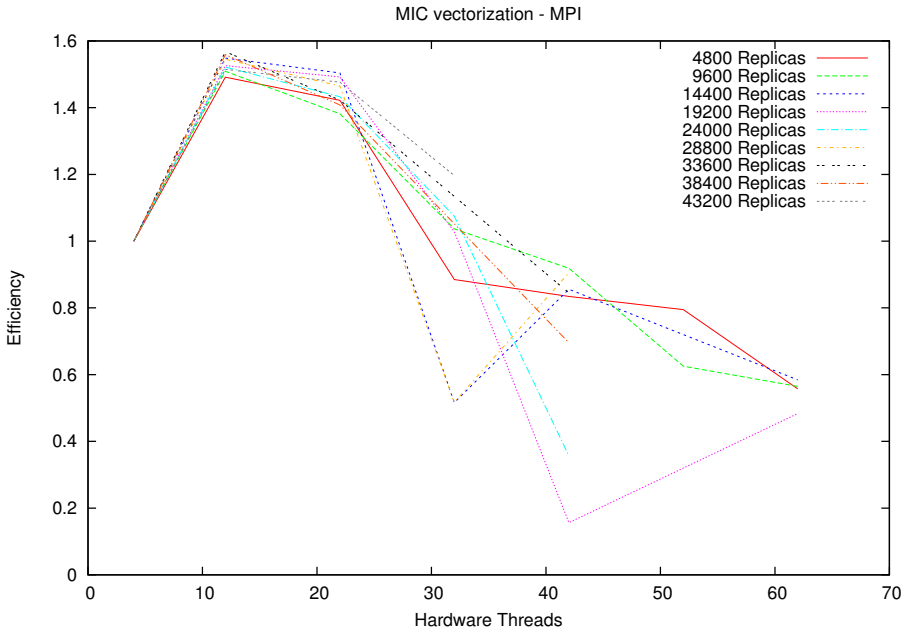


(b) Threaded with vectorization

Figure 5.20: Intel MIC threaded results



(a) MPI without vectorization



(b) MPI with vectorization

Figure 5.21: Intel MIC MPI results

is faster than the MPI implementation (not shown). Also, the non-vectorized implementation obtain a higher efficiency than vectorized implementations, but the vectorized implementations have a lower runtime. The lack of efficiency can be attributed to the large amounts of memory that have to be shuffled from main memory to the cores. Also, the memory accesses of one core significantly affect the memory accesses of another core. Other architectural features affect the performance as well.

5.3 GPU

As discussed in chapter 4, CUDA and OpenCL implementations were developed to run on graphics processing units. Before discussing any of the results, it is first necessary to determine the optimal number of threads per block. While groups of 32 threads (warps) are created, managed, scheduled, and executed by the streaming multiprocessor, it is logical to assume that setting 32 threads per block would be a satisfactory configuration. However, this is not always the case so it is necessary to explore what configuration works best for this implementation. Figure 5.22 shows a series of runs on the GeForce GTX 480 GPU varying the number of threads per block by multiples of 32. Although 64, 128, and 256 threads produce similar runtimes, 64 is chosen to allow for more active thread blocks per multiprocessor. For the remainder of this dissertation, it will be understood that the number of threads per block used is 64.

Chapter 4 lists a number of optimizations that were developed on the GPU to enhance the performance. Figure 5.23 shows the runtime (5.23a) and speedup (5.23b) of each optimization over the naive implementation for CUDA and OpenCL on each device using 46080 replicas and 180 atoms. From the figure, it can be seen that swapping the row and column data gave an average of 2.5x improvement for the C1060 and ATI 7970 cards and almost a 3x improvement in the Fermi cards. The greater performance increase in the Fermi cards is due to the global memory L1 cache. The CUDA implementations outperformed all of the OpenCL implementations. For the

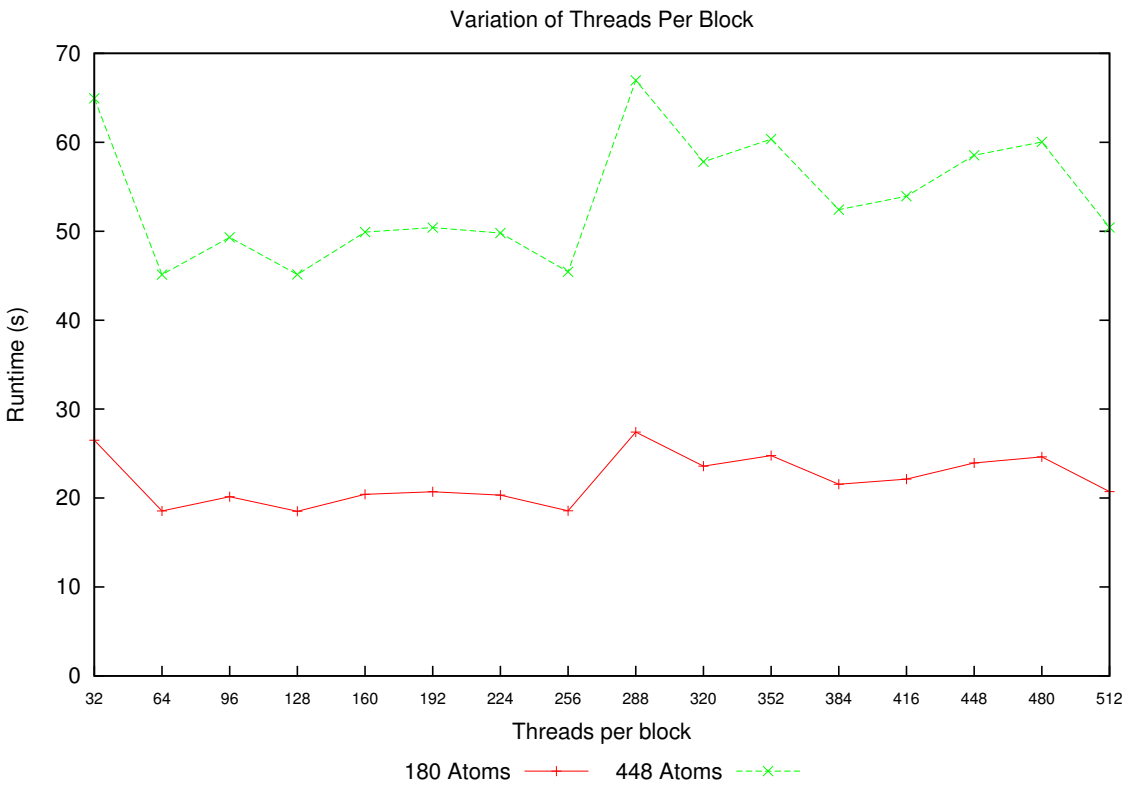
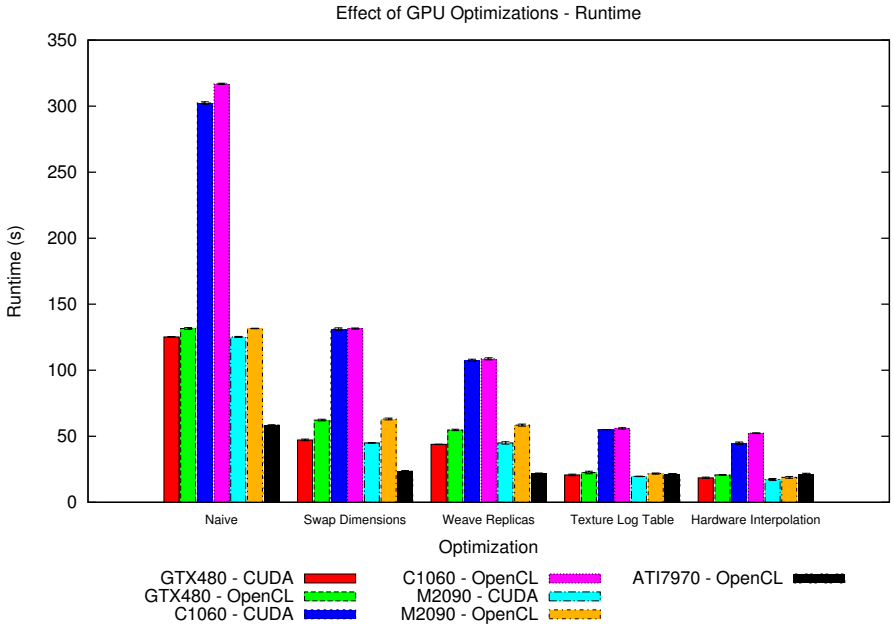
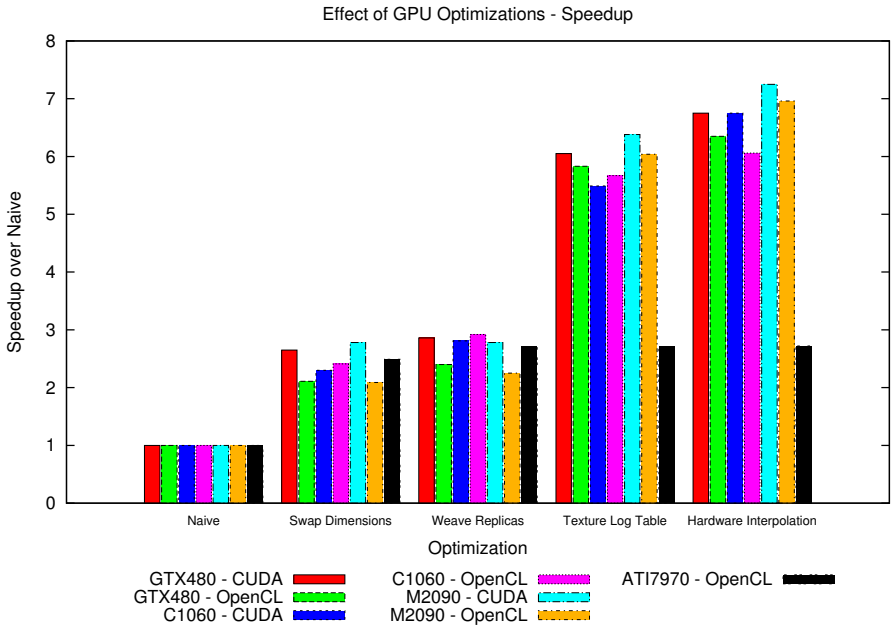


Figure 5.22: Varying number of threads per block

NVIDIA GPUs, the greatest optimization was reducing the table size by transforming it on the log scale and accessing it through texture memory. Doing this allowed for the table to be cached in all of the GPUs and freed up the L1 cache in the Fermi GPUs. Using the texture interpolation provided a 1.11x speedup of straight accesses to the texture memory. Swapping the dimensions was the greatest optimization for the OpenCL ATI card, while the other improvements had little or no effect. NVIDIA GPUs received a performance increase from all of the optimizations. However, the optimizations had a smaller effect on the OpenCL implementation that the CUDA implementation. This is surprising as both the CUDA and OpenCL implementations essentially use the same kernel (only with syntax differences). However, a decrease in performance when using OpenCL coincides with the observations of other researchers [96, 97]. One interesting observation which can be seen in figure 5.23a is even though the optimizations had little effect on the performance on the ATI GPU, its runtime



(a) Runtimes



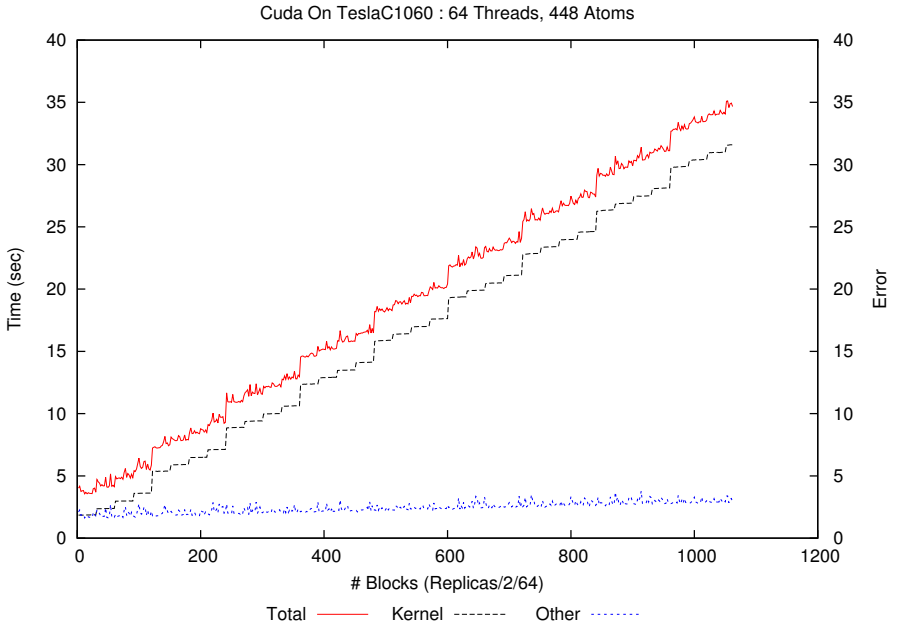
(b) Speedups

Figure 5.23: GPU optimization results over naïve implementation

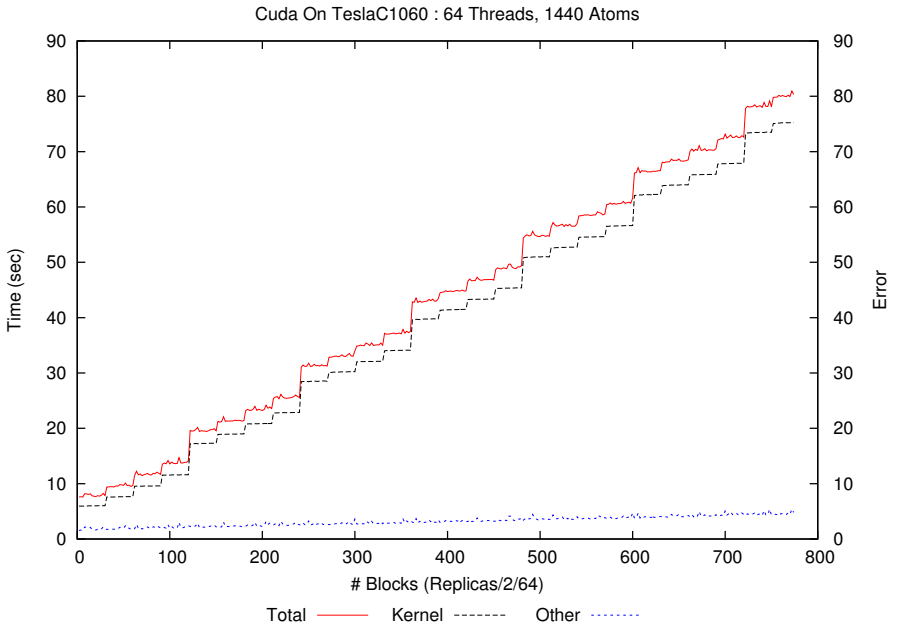
with the first optimization is close to the OpenCL implementation on the GeForce GTX 480 with all of the optimizations. Therefore, it took less work to obtain an "optimized" kernel on the ATI GPU than on the NVIDIA GPUs.

Figures 5.24-5.27 give plots of the runtime on the GPUs listed in table 5.2 with varying workloads. All of these runs are for 100 iterations. For the NVIDIA GPUs, there is a "stair step" function that indicates the saturation of the SMs with the workloads. In the case of the Tesla M2090, for example, for every multiple of the number of active blocks per SM, there is a step indicating the amount of extra work needed to handle the extra block on a new SM. For every multiple of $\#SMs * ActiveBlocks = 128$, there is a large step up that indicates all of the SMs were fully saturated prior to the step and the extra work needed to compute the additional blocks that are scheduled to the now oversaturated SM. A similar trend for the ATI 7970 could not be determined due to limitations in the drivers hindering large simulation parameters.

Before discussing the performance comparison between the GPU and multiprocessor, it is necessary to check the correctness of each of the implementations. Due to varying specifications on the accuracy of various function and operations (e.g., basic arithmetic operations and transcendental functions), each device and OpenCL implementation can produce different results depending on the specified accuracies. In some applications, these error differences may not make much of a difference since the error is so small. However, in QSATS, these errors easily build on each other as the movement of one atom affects the movements of its neighboring atoms and the atoms in the neighboring replicas. Thus, if one atom's movement is accepted on one device but rejected on another, a completely different trajectory is taken, but should still be statistically similar. Therefore, it is necessary to check for this statistical correctness. While correctness was shown in figure 4.4, it only does so for the CUDA implementation on the GeForce GTX 480. Figure 5.28 shows the correctness of the CUDA and OpenCL implementations on the devices listed in table 5.2. These results are obtained by first performing a warmup run of 10,000 iterations with the

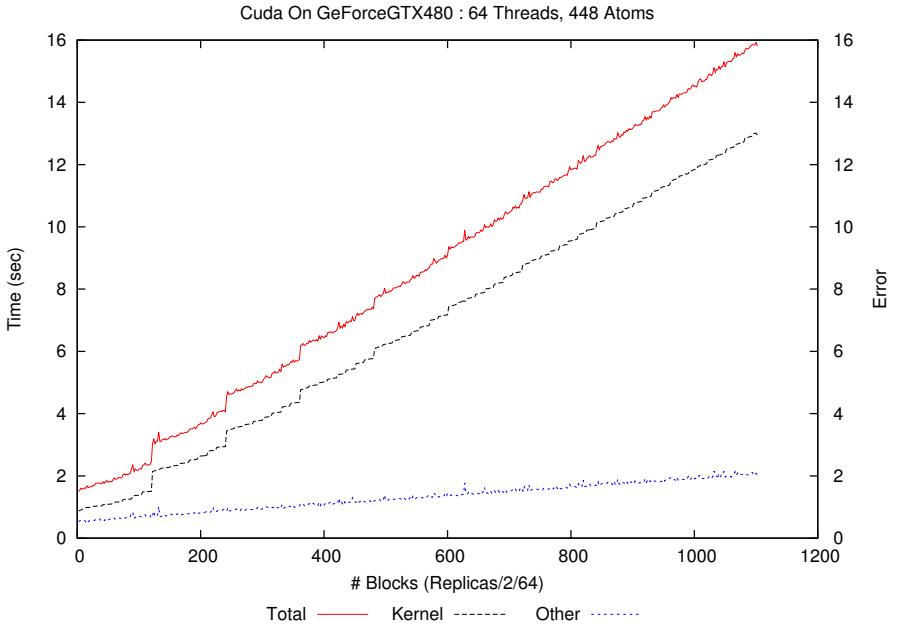


(a) 448 Atoms

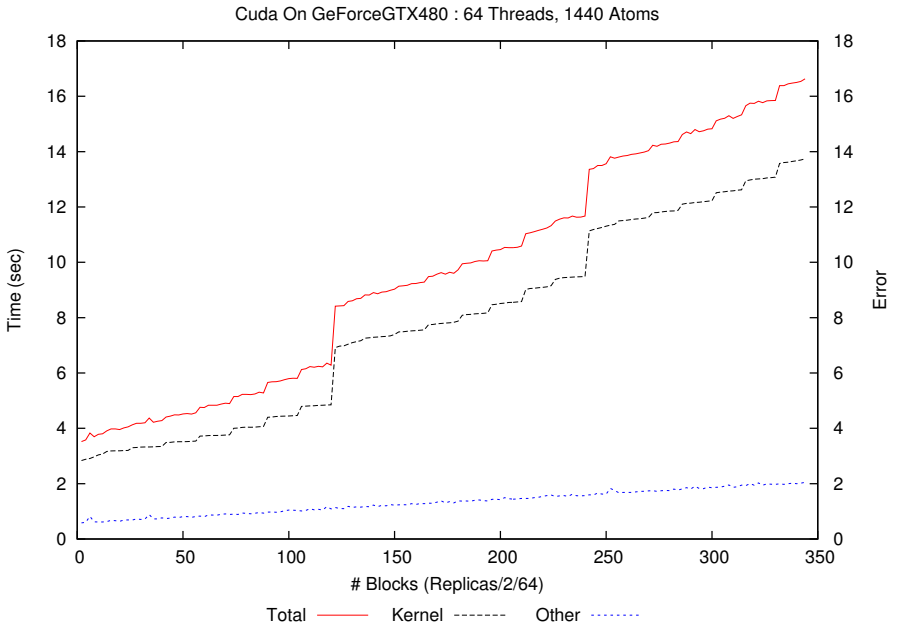


(b) 1440 Atoms

Figure 5.24: Tesla C1060 runtimes with varying numbers of replicas

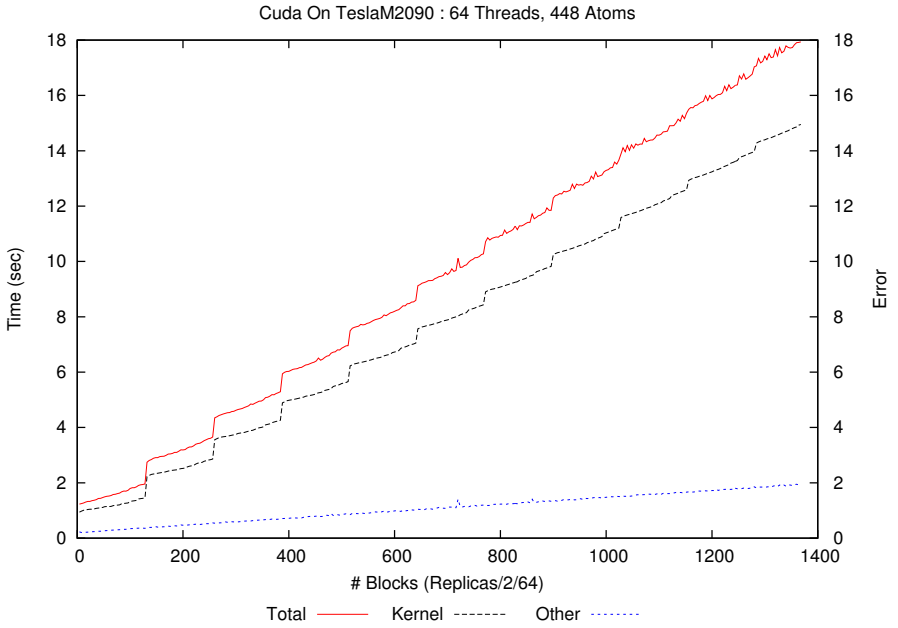


(a) 448 Atoms

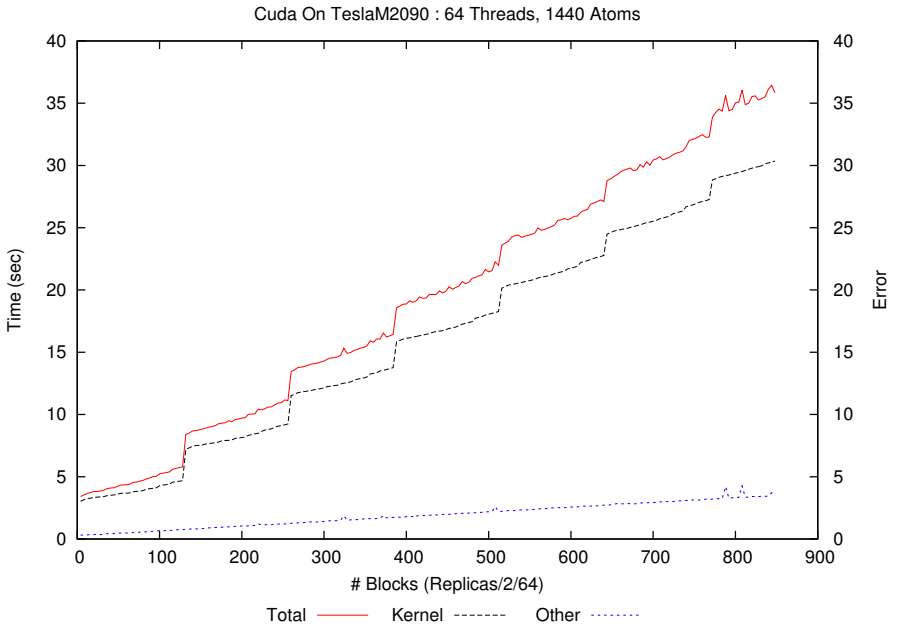


(b) 1440 Atoms

Figure 5.25: GeForce GTX 480 runtimes with varying numbers of replicas

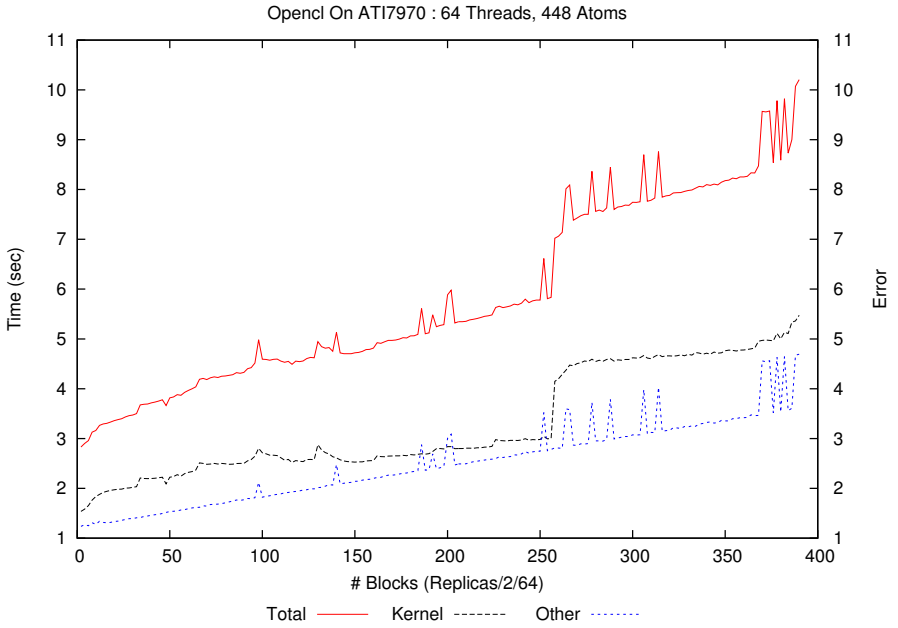


(a) 448 Atoms

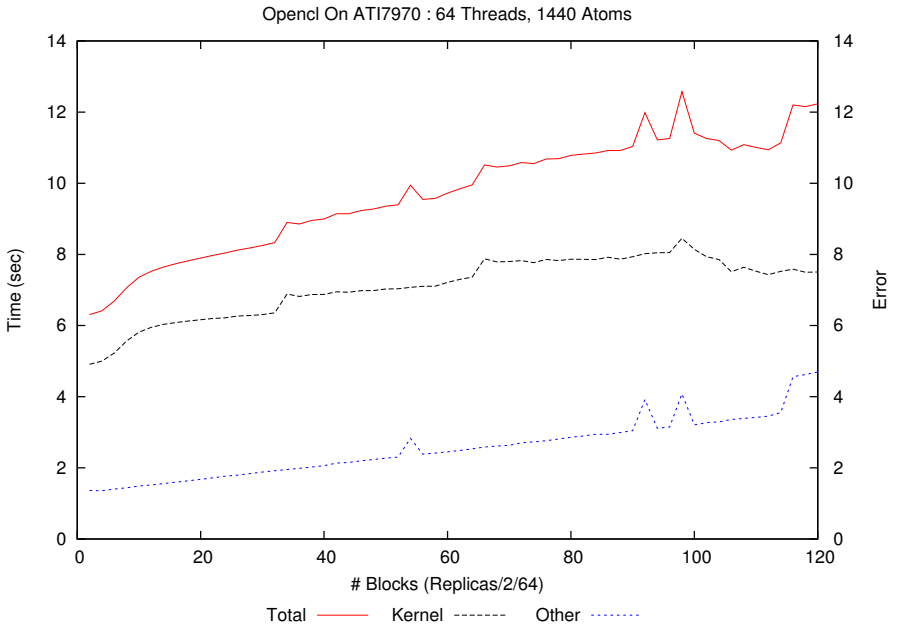


(b) 1440 Atoms

Figure 5.26: Tesla M2090 runtimes with varying numbers of replicas



(a) 448 Atoms



(b) 1440 Atoms

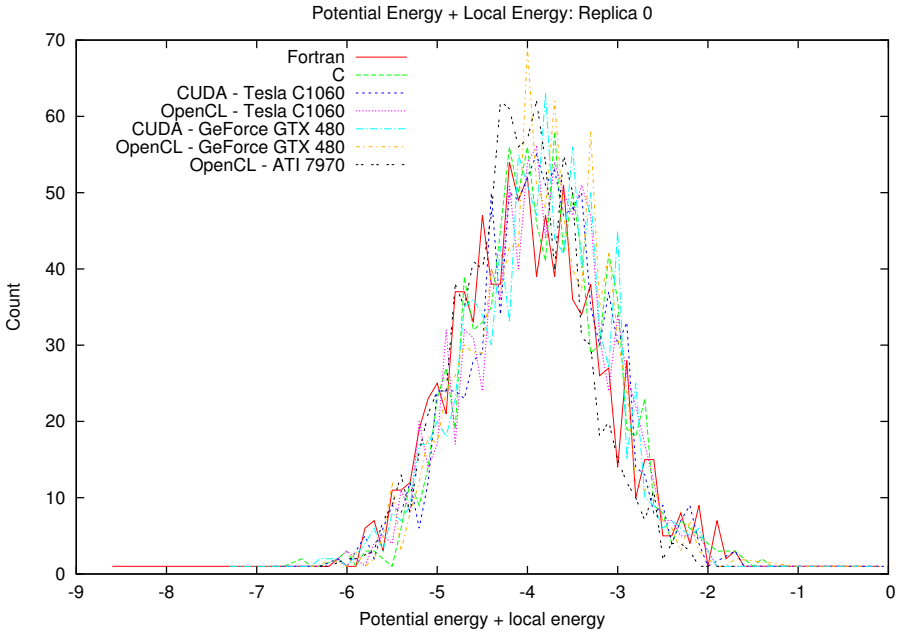
Figure 5.27: ATI Radeon 7970 runtimes with varying numbers of replicas

parameters listed in table 5.3. Then, a run of 100,000 iterations was performed taking snapshots every 100 cycles resulting in 1,000 snapshots. The plots show a histogram of the sum of potential energy and local energy for the snapshots obtained from the ELOC program [12].

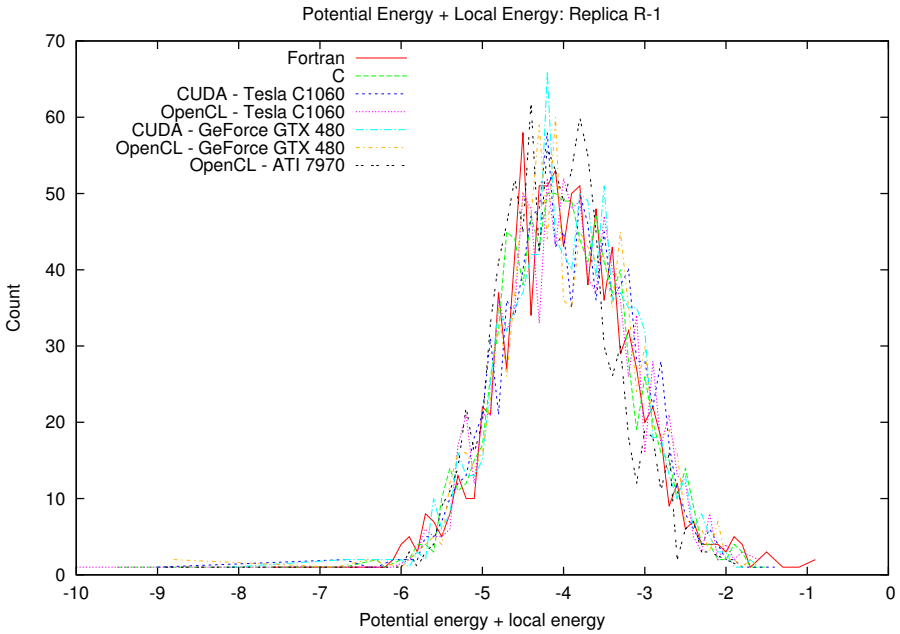
Using the original Fortran code as the gold standard, it can be seen that all of the GPU implementations follow the same statistical distribution of energies. Differences in the GPU and multiprocessor runs are due to differences in optimization techniques and accuracy of mathematical operations.

Next, the GPU and multiprocessor performances are compared. The GPU implementation is compared to two nodes of the Kraken supercomputer each containing 12 cores of the AMD Opteron Istanbul processors using SSE for a total of 24 cores. This processor was chosen because it is the direct competitor to the GPU when determining what architecture should be used for a production run. Since each device has a limitation on the number of replicas accepted due to load balancing reasons, the multiprocessor and GPU codes use different numbers of replicas. Figure 5.29 shows the rates of the CUDA NVIDIA implementations and the ATI OpenCL implementation over the multiprocessor for various GPUs. The OpenCL implementations for the NVIDIA GPU were left out since the CUDA was shown to provide higher performance. The rates given are for 1,000 iterations, 180 atom runs. The Istanbul run used 45,816 replicas, the Sandy Bridge used 46,080, and the GPU implementations used 46,080.

From this figure, it can be seen that the Fermi card, Tesla M2090, obtains the highest performance at 1.61x faster than 24 cores of the Istanbul processors in Kraken and 2.08x faster than 16 cores of the Sandy Bridge processors. What this means is a single GPU can obtain the same performance as approximately 3 nodes (i.e., 36 cores) of Kraken. Also, if we were to project the Sandy Bridge processor to 24 cores to compare directly to the Istanbul, it would be 1.15x faster resulting in the GPU only being 1.3x faster than the Sandy Bridge. This is substantial as using the GPU there is a reduction in compute hours needed and a reduction in power consumption.



(a) First replica



(b) Last replica

Figure 5.28: Check for GPU statistical correctness

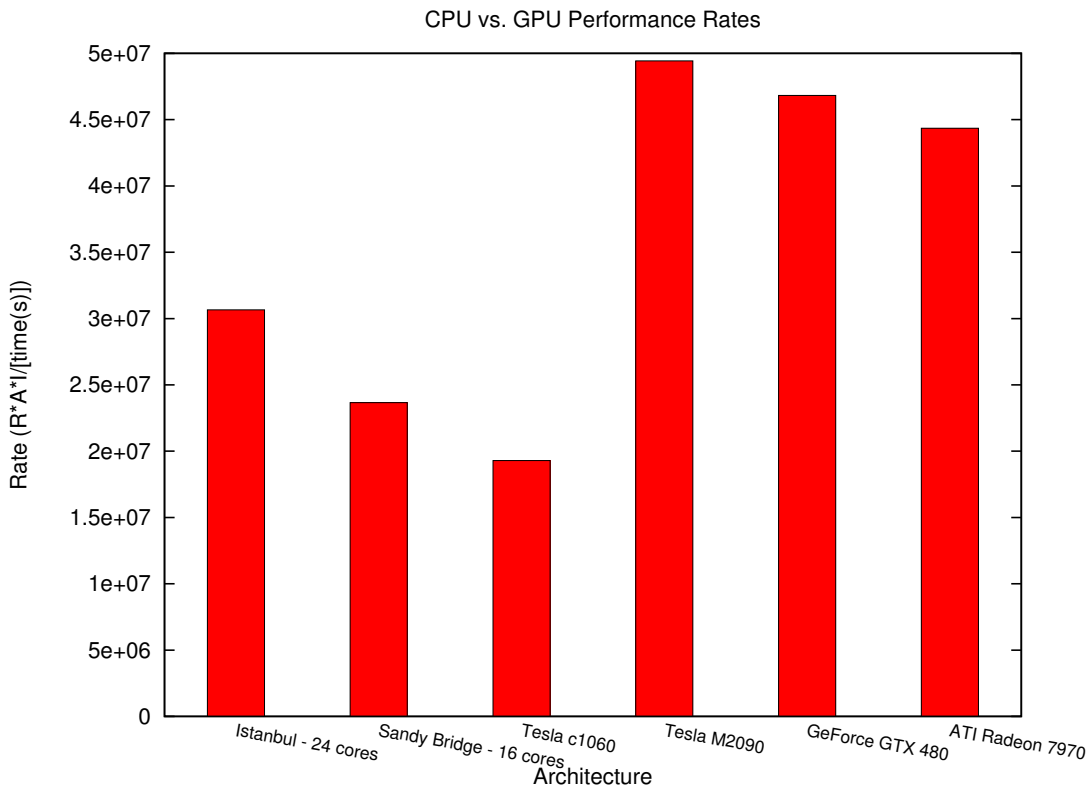
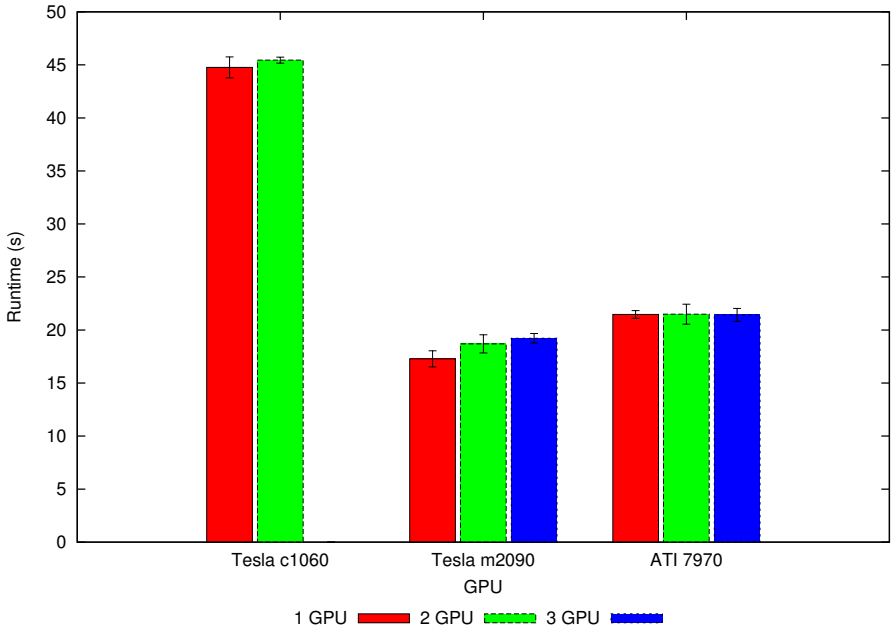


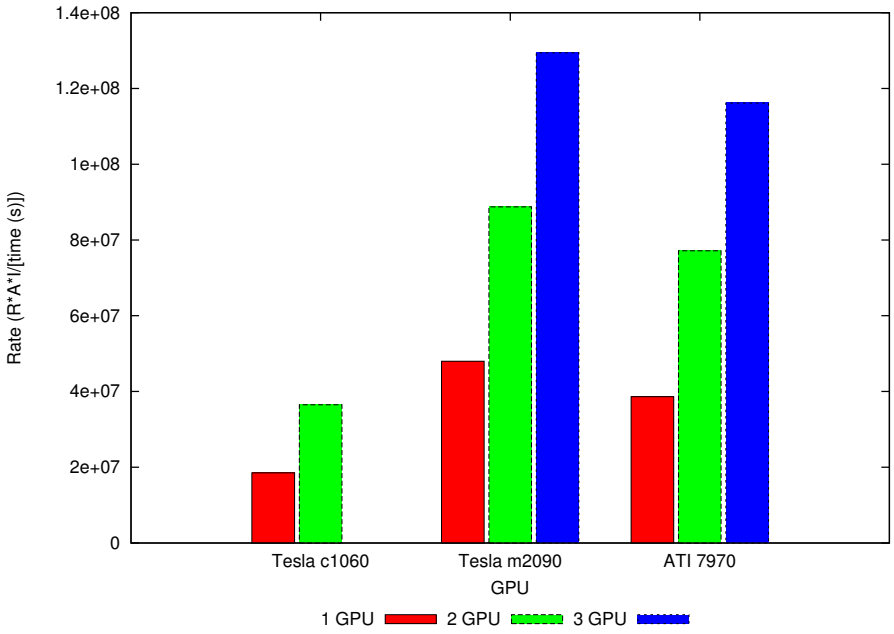
Figure 5.29: Performance rates of the CPU and GPU implementations

For a single Tesla M2090, the power consumption is 255 Watts [98]. For a single Istanbul multiprocessor, the power consumption is 75W [99]. Therefore, in order to power 3 nodes of 2 processors each, the total power consumption from the processors alone is $75 * 3 * 2 = 450W$. Because of these reasons, the GPU is the best architecture available for this application.

QSATS also has the capability of performing multiple independent simulations concurrently by using multiple GPUs. As figure 5.30 shows, using multiple GPUs roughly gives the linear speedup that we expect due to the independence among the GPUs. The CUDA code was run on the Tesla C1060 and Tesla M2090 as it was shown earlier to produce the highest performance. OpenCL was used for the ATI 7970. Each run used 180 atoms and 100 iterations and each GPU executed 46,080 replicas for these runs.



(a) Runtime



(b) Rate

Figure 5.30: Results of the multi GPU implementation

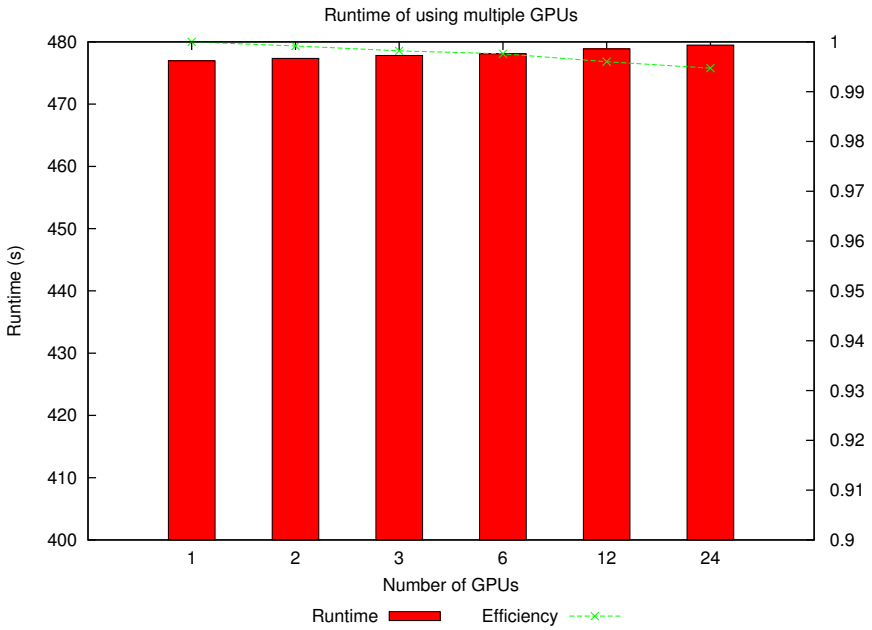


Figure 5.31: Multiple GPU runs on Keeneland

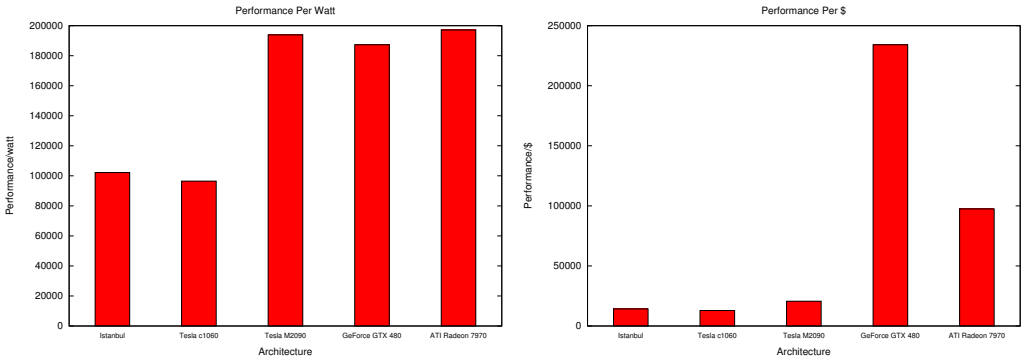
QSATS is also capable of scaling on the GPUs beyond a single node. As shown in figure 5.31, using 24 GPUs provides a near perfect efficiency of approximately 99% running using 16,384 replicas per GPU, 1440 atoms, 1,000 iterations, and 100 snapshots. This is due to all the devices performing completely independent simulations with no need to exchange information.

5.4 Architecture Comparisons

In comparing these architectures, there are many pros and cons for each. From a cursory glance, it would appear that the NVIDIA Tesla M2090 is the dominant platform. However, when comparing performance per watt or performance per dollar, a different conclusion could be reached. Table 5.5 gives the power consumption and current price as found on Amazon.com for each device. In comparing architecture performance, the NVIDIA Fermi GPUs provide greater performance than the ATI 7970 as can be seen in figure 5.29. In particular, the Tesla M2090 has the highest performance. From a power consumption perspective, the ATI 7970 has a slight

Table 5.5: Power consumption and current price for each compute device

Device	Power	Current Cost
Istanbul	75 [99]	$540
Sandy Bridge	75 [100]	$1727
Tesla c1060	200 [85]	$1500
Tesla M2090	255 [98]	$2400
GeForce GTX 480	250 [101]	$200
ATI 7970	225 [102]	$455



(a) Performance per watt **(b)** Performance per $

Figure 5.32: Architecture performance per watt and dollar

advantage over the Fermi cards when looking at the ratio of performance per watt shown in figure 5.32a. Saying this, the performance per watt of the Tesla M2090 is only 2% less than that of the 7970 so the difference is negligible. In terms of performance per price (dollar) as shown in figure 5.32b, the GeForce dominates as the price is only $200 and the performance is on par with the Tesla M2090. However, if using the release price ($500), the GeForce GTX 480 performance per price is 4% less than the 7970.

Weighing all these metrics, it would appear that the GeForce GTX 480 is the architecture to use for this application. However, it only has 1.5GB of global memory so the crystal size and number of replicas that can be used is extremely limited. Also, it is only a consumer card and will not be adopted in any high performance clusters or supercomputers. Finally, for compute intensive applications, the GeForce GTX 480 is not a good choice if double precision is needed as the performance for double precision

operations is very poor, similiarly with other consumer cards as recognized in [46]. All of these cons make the GeForce GTX 480 unworthy of general high performance computing.

While the ATI 7970 has better double precision support, with performance similar to the Fermi cards, and has a relatively low price point, it too is not a prime choice for general high performance computing. Since AMD/ATI intend for the card to be used for consumer graphics use, they have not eased the development of general code for the device besides enabling OpenCL support. In particular, the 7970 does not allow for the allocation of a buffer larger than 512MB while the card itself has 3GB of memory. With the algorithm as it stands, the size of the simulation possible is severely limited. For a crystal structure of 7920 atoms, for instance, the maximum number of replicas that can be simulated is $512\text{MB}/(7920*3*8) \approx 2,824$. In addition, ATI GPUs are not readily available in high performance computing systems as only two machines in the Top 500 list have ATI Radeon GPUs while 53 machines have NVIDIA [103].

The Tesla c1060 is an old, outdated GPU that has long been replaced with newer cards such as the M2090. Lack of availability in HPC systems and lack of performance in comparison to other devices makes this architecture undesirable for many computational science applications.

The Istanbul multiprocessor is also an old processor; however, it is in two of the top 30 machines [103], one of which is the University of Tennessee machine, Kraken. While not one of the highest rating architectures explored in this work, it is still an important one to consider. The advantage of this architecture is the availability (112,800 cores in Kraken), programming familiarity as it can be programmed with a tradition language such as C or Fortran, and the ability to develop high performance, highly scalable applications.

The Intel Sandy Bridge processor used in this work, Xeon E5-2680, is one of the newest and most powerful processors tested here. In comparison to the Istanbul, it provides a higher performance courtesy of the high clock rate and AVX vector units.

This processor is so powerful, the Stampede supercomputer [104] will house thousands of these to work alongside Intel MIC cards. However, this performance comes at a rather high price at nearly 3x the cost of the Istanbul processor.

In regards to programmability of GPU languages, CUDA is superior to OpenCL for numerous reasons. There are development tools for the CUDA language such as debuggers and profilers. This makes debugging, tuning, and optimizing code much easier as these tools can be used natively on the device. In order to debug OpenCL code, one must first target the code to the processor and use a traditional debugger such as `gdb`. However, this comes at the penalty of significantly slower runtimes and lack of truly emulating the GPU architecture. One benefit of the OpenCL language is the ability to write the code once then target it to any other architecture such as NVIDIA GPUs ATI GPUs, multiprocessors, and even FPGAs. CUDA, on the other hand, can only run on NVIDIA GPUs. Despite this disadvantage, the availability of support tools, in my experience, make CUDA a better option than OpenCL by reducing development and debugging time.

As it stands right now, the Intel MIC is definitely an interesting upcoming architecture. For this application, the scalability as the number of cores increases is lacking and is therefore not a viable architecture when using the Intel cards. Due to other architectural constraints, it is not beneficial to use this architecture on problem sizes other than toy problems, especially in regards to snapshotting. Some or all of these issues might be fixed with the release of newer cards providing a higher performance for this code. Programmability for the Intel MIC is very convenient as it can be programmed in a traditional language such as C or Fortran and using familiar libraries such as OpenMP and MPI. Using intrinsics, it is possible to use the 512 bit vector units to accelerate computationally expensive code. Since QSATS becomes a memory bound problem when using multiple threads, the effect of vectorization is minimal since a large number of memory accesses must be serialized.

From this chapter, it can be seen that most of the architectures here provide a high performance platform for this application. The multi-node microprocessor

implementation provides high performance for a relative small number of replicas (i.e., tens of thousands) for a single simulation. The GPU implementation provides high throughput for multiple simulations executing extremely large number of replicas (i.e., hundreds of thousands). After viewing all of these results, it is logical to ask, "Is there a way to predict these results?".

Chapter 6

Performance Modeling

Performance modeling is an important aspect of developing high performance applications. Models give the ability to analyze various architectural features of different platforms to determine what characteristics are best suited for an application. Models also have the capability to predict certain aspects of the application such as runtime. This gives the engineer the ability to determine which architectures to invest time and money.

This chapter will discuss the performance models for the microprocessor, Intel MIC, and GPU. For the microprocessor and Intel MIC architectures, the models express the runtime in terms of clock cycles for arithmetic operations (addition, multiplication, transcendentals, etc) and memory accesses. In addition, models are presented that predict the scalability of the microprocessor MPI implementation on the Kraken supercomputer. The GPU model proves to be more difficult as low level architectural details are not readily accessible; however, a model is presented that accurately predicts the runtime on multiple types of GPUs.

6.1 Microprocessor Performance Model

The microprocessor implementation has two different models: single core implementation and multi-process implementation. The single core model is used to obtain a

parameter needed in the multi-process scalability model. This section goes into details about both of these model and presents the accuracy relative to actual runtimes presented in the previous section.

6.1.1 Single Core Model

This section will give the details of the single microprocessor core model that will describe the runtime. First, the original naïve model that lays the ground work for this portion of the dissertation is discussed. Then, this dissertation dives deeper into the performance model to include various architectural details that can be used to explain the performance differences of different microprocessor architectures.

Original Model

Originally, this work hypothesizes that the model for each architecture will be dependent on the number of replicas, iterations, and atoms. These three factors along with a base runtime combine to create a rate, α, that describes how fast an architecture can perform the QSATS method per atom per replica per iteration. This rate can then be used to extrapolate a runtime given any amount of replicas, atoms, and iterations as shown in 6.1.

$$RAI\alpha \tag{6.1}$$

In order to obtain α, a small preliminary run was needed to get the base runtime to plug into equation 6.2.

$$\alpha = \left(\frac{t}{RAI}\right)_0 \tag{6.2}$$

This proved to be an accurate model with an error of 6.42% in the worst case. However, such a model does not provide any insight into the architecture nor can it be used to compare or predict for hardware that is not physically available.

Table 6.1: QSATS specific parameters for the performance model

Name	Description
R	Number of replicas
A	Number of atoms
I	Number of iterations
P	Number of interacting pairs
S_I	Number of iterations between snapshots

Improved Model

Because the original model did not provide any valuable insight into architecture, it was necessary to develop a model that did. QSATS can be broken down into the following steps:

- Initialization
- For each iteration, replica, and atom
  - Calculate a Gaussian random displacement
  - Calculate potential energy for moving to new state
  - Accept or reject the move
- Occasionally, take a snapshot of the current state
- Finalization

Each of these steps have a different complexity that contribute to the runtime of the program. This is shown in figure 6.1. It can be seen that the total complexity of this application is $\mathcal{O}(R + RAI(2 + P))$. The QSATS method has the parameters shown in table 6.1 that dictate the performance of the program.

In order to develop a model, it is necessary to first start at a high level description of the runtime. Equation 6.3 gives a general description of the QSATS runtime split into various time-consuming portions of the application.

$$t_{serial} = t_{init} + \sum_{i=1}^{I} \sum_{r=1}^{R} [t_{disp} + t_{poten}] + \sum_{s=1}^{I/S_I} t_{snap} + t_{final} \tag{6.3}$$

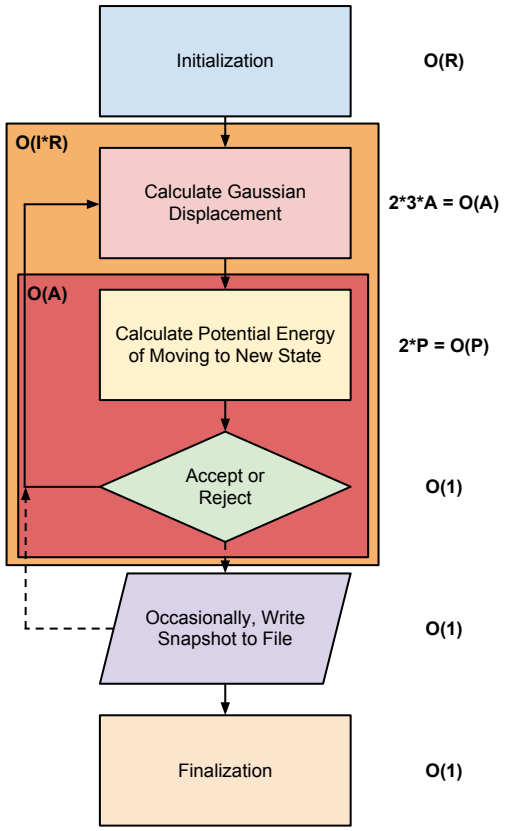


Figure 6.1: Complexity of the QSATS steps

where the times are described in table 6.2. The initialization time, t_{init}, can be defined as the sum of the time it takes to allocate all arrays, set the displacement vector to zero, and set the initial RNG states for all replicas. If a save file is specified, the additional time it takes to read the file is added to this. While negligible for the serial case, this is an important factor for the scalability of the program when attempting to scale beyond hundreds of cores.

The displacement time, t_{disp} is defined as the sum of time taken to calculate the average of atom positions in the neighboring replicas for all atoms plus the time taken to generate three Gaussian random numbers for all the atoms. In other words,

$$t_{disp} = A * (3 * t_{avg} + 3 * t_{gauss}) \tag{6.4}$$

Table 6.2: Break down of times in the QSATS program

Symbol	Description
t_{serial}	Total serial simulation time
t_{init}	Initialization time
t_{disp}	Displacement time
t_{poten}	Potential energy time
t_{snap}	Snapshot time
t_{final}	Finalization time

This is a $\mathcal{O}(A)$ operation as both the averages and the Gaussians must be calculated for all atoms.

The potential energy calculation time, t_{poten}, is split into two time consuming parts as shown in equation 6.5: squared distance calculations and potential energy look-up time.

$$t_{poten} = 2 * P * (t_{sqdist} + t_{lookup}) \tag{6.5}$$

The factor of 2 accounts for the square distance calculation and potential energy look-up for both of the old and new atom positions. This timing does not account for the predetermination of acceptance or rejection of an atom move described in section 3.3.1. To incorporate this into the timing, the lookup time is split into two parts as shown in equation 6.6: the look-up time for the inner P_I and outer P_O pairs.

$$t_{poten} = 2 * P * t_{sqdist} + 2 * t_{lookup}(P_I + \beta * P_O) \tag{6.6}$$

In this equation, the β factor is the percentage of times that the look-up has to be calculated due to the a squared distance being outside of the monotonically increasing section of the potential energy curve. For the parameters relevant to this work, β in equation is set to .03, as the potential energy for the outer pairs only needs to be calculated 3% of the time.

The time it takes to write a single snapshot, t_{snap}, is a factor of the size of the packet being written and the I/O speed, D, in bytes per second, of the system as

shown in equation 6.7.

$$t_{snap} = \frac{3 * A * R * 8}{11 * D} \tag{6.7}$$

The divisor, 11, is because QSATS only writes the 11^{th} replica when performing snapshots in order to get replicas that are far enough apart so that they have minimal influence on each other. The 8 multiplier is to calculate the number of bytes since there are 8 bytes in a double precision number.

The finalization time, t_{final}, is dominated by the time it takes to write the state of all the replicas to a file on disk. Similar to the snapshot time, equation 6.8 gives the finalization time, which is simply the time it takes to write a file to disk.

$$t_{final} = \frac{8 * R * (3 * A + 8)}{D} \tag{6.8}$$

The additional 8 is to store the RNG state so a continuation run can start from the state in the stream that it left off.

Equations 6.4 through 6.8 are fed into equation 6.3 to deliver a model that is dependent on the time needed for each step in the process. As it stands, the model does not provide any insight on the underlying architecture.

The next step is to incorporate various aspects of the microprocessor architecture. For the microprocessor implementation, the algorithm is compute bound for every section described above except for the potential energy look-up. This allows us to count clock cycles for the different floating point operations in the algorithm (e.g., multiplication, sine, cosine). In the following, each portion of the code is described as the composition of cycle counts that can take the place of times given above in the model equations. All operation counts are determined by looking at the C code only. Assembly code was used to determine what instructions the fundamental operations (i.e., multiplication, addition, subtraction) mapped to. Table 6.3 gives the essential parameters that are used in this model.

Table 6.3: Operation cycles used in the microprocessor performance model

Parameter	Description
c_{add}	Addition cycles
c_{sub}	Subtraction cycles
c_{mult}	Multiplication cycles
c_{sqrt}	Square root cycles
c_{log}	Logarithm cycles
c_{sin}	Sine cycles
c_{cos}	Cosine cycles

First, the uniform random number generator consists of a total of 17 floating point operations to generate a single uniform random number in the range (0,1]. As such, the cycle composition is shown in equation 6.9.

$$c_{uniform} = 11 * c_{mult} + 6 * c_{add} \tag{6.9}$$

Next, the Gaussian random number generator consists of a significant number of operations. In this work, the Box-Muller method is used to convert two uniform random number generators to two Gaussian random numbers using the following transformation.

$$\begin{aligned} Z_0 &= \sqrt{-2 * ln(U_0)} * cos(2\pi * U_1) \\ Z_1 &= \sqrt{-2 * ln(U_0)} * sin(2\pi * U_1) \end{aligned} \tag{6.10}$$

where $U_{\{0,1\}}$ are uniform random numbers and $Z_{\{0,1\}}$ are the Gaussian random numbers. As can be seen from this equation there are a total of 8 multiplications, two square roots, one natural logarithm, two trigonometric functions, and two uniform random numbers. The 8 multiplications can be reduced down to 4 by reusing values and using hard coded constants ($2*\pi$). Likewise, the two square roots can be reduced to one by using the computed value twice. However, since Box-Muller generates two random numbers at a time, the second random number is stored into memory; therefore, the cost of this is only encountered half of the times the Gaussian RNG

is called. Equation 6.11 gives the average number of cycles needed to generate the Gaussian random numbers for all of the atoms in a replica.

$$c_{gaussian} = \frac{3 * A}{2} * (2 * c_{uniform} + c_{sqrt} + 4 * c_{mult} + c_{log} + c_{sin} + c_{cos}) \tag{6.11}$$

Third, calculating the average of the "neighboring" replicas is pretty straight forward. Since it is simply calculated as the average of two numbers, the number of cycles to compute all of atoms is shown in equation 6.12.

$$c_{average} = 3 * A * (c_{add} + c_{mult}) \tag{6.12}$$

Fourth, the squared distances are calculated between the current atom location and all of its P interacting neighbors. This has to also be done with the new proposed positions that are calculated by adding the Gaussian displacements. Calculating this involves three additions to connect the atoms to their stencil coordinates, 3 subtractions between the old coordinates and the neighbor coordinates, 3 subtractions between the new coordinates and the neighbor coordinates, 6 multiplications to square the calculated differences, and 4 additions to sum the squares (2 for each of the old and new positions).

$$c_{calcd^2} = P * (7 * c_{add} + 6 * c_{sub} + 6 * c_{mult}) \tag{6.13}$$

It should be mentioned that calculating the squared distances is heavily pipelined so that the program achieves the highest throughput possible with this portion of the code. To represent this in the model, the cost of the addition, subtraction, and multiplication are all set to 1 since we get a throughput of one calculation per cycle turning equation 6.13 into equation 6.14

$$c_{calcd^2} = 19 * P \tag{6.14}$$

Fifth, the difference in potential energy between the old and new positions is calculated. This step takes the cost of two potential energy table look-ups which can be described as the average memory look-up time (AMAT), a multiplication, an addition, and a subtraction. While the AMAT is not something that can be analytically determined, it is still possible to feed a set of look-up addresses to a cache simulator such as Dinero IV [105]. Using a cache simulator to calculate the AMAT provides insight into the underlying architecture that can be used to predict runtime, eliminating the using of ambiguous "fudge factors" used in section 6.1.1. We can still obtain the AMAT without having physical access to the machine so it is possible to experiment with different cache layouts. The number of cycles to calculate the difference in potential energy for a set of two squared distances (one for the old and new positions) is given in equation 6.15.

$$c_{dv} = 2 * AMAT_{lookup} + c_{mult} + c_{add} + c_{sub} \tag{6.15}$$

where AMAT is defined as

$$AMAT = HitRate * HitTime + MissRate * MissTime \tag{6.16}$$

To model the optimization where the outer neighbor potential energy look-ups are skipped, equation 6.15 split into two equations. For the inner neighbors, this equation is multiplied by the number of inner neighbors, P_I, since the program will calculate the potential energy at least this many times. For the outer neighbors, this equation is multiplied by the number of outer neighbors, P_O, and a percentage value, β that describes the percentage of the time the energy contribution from the outer neighbors is influential to the outcome of the atom's movement. These can be shown in equations 6.17 and 6.18.

$$c_{innerdv} = P_I * c_{dv} \tag{6.17}$$

$$c_{outerdv} = \beta * P_O * c_{dv} \tag{6.18}$$

Next, the cost to determine whether to accept or reject a move after calculating the potential energy of the inner neighbors is modeled. A single table look-up, three multiplications, a log, and two subtracts make up the majority of the calculations in determining this decision. Equation 6.19 gives the number of cycles needed to determine whether a move is accepted or rejected.

$$c_{predetermine} = AMAT_{lookup} + 3 * c_{mult} + c_{log} + 2 * c_{sub} \tag{6.19}$$

All of these cycle equations can be summed together to make the CPU cycle model given equation 6.20.

$$\begin{aligned} c_{serial} = I * R * (c_{average} + c_{gaussian} + \\ A * (c_{uniform} + P * c_{calcd^2} + c_{innerdv} + c_{outerdv} + c_{predetermine})) \end{aligned} \tag{6.20}$$

To calculate the time needed to execute the program, it is possible to simply divide equation 6.20 by the clock frequency of the processor as shown in equation 6.21.

$$t_{serial} = \frac{c_{serial}}{Frequency} \tag{6.21}$$

This model is powerful enough to model the various differences across architectures by simply using the latencies for the different operations. This next question is: how are the latencies determined? For the basic arithmetic and square root operations, these are listed in the documentation about the processor families [106, 107]. As for the transcendental functions, it's slightly more complicated as these do not map directly to a single assembly instruction. Instead, these are function calls that call the standard C math library. Upon inspection of the C code, the sine and cosine have multiple paths to calculate the appropriate value based on the input number. This provides accuracy and speed for different inputs. However, there is a significant amount of optimization that occurs during the compilation stage. Also, different versions of compilers produced different codes most probably due to different

optimization techniques employed by the compilers and different versions of the C math library. For the same reasons, it is also difficult to derive latency counts for the logarithm operation. To handle this, a simple code is used that generates a large amount of random numbers that were in the same range that QSATS would use the functions. The code will then loop through all of the numbers performing the operation on each. Using PAPI [93], the total number of cycles taken to execute the operation on all of the numbers in the array is available. Dividing this total number of cycles by the number of operations performed gives an average count of cycles per operation. Using these numbers and the cycles obtained from the documentation, it is now possible to use the model to predict runtimes.

This model will also work for the SSE and AVX implementations. Since the table look-ups are not guaranteed to be stored in consecutive memory locations, the look-ups must also be serialized; therefore, these cycles cannot be divided by these factors either. Modifications to this model are necessary to account for architectural differences and compiler optimizations that are available for each processor. These modifications are discussed in the following section.

Model Performance

The first step to verifying the single core model is to obtain the latency clock counts for all of the operations. Basic arithmetic operation latency can be freely found in the documentation for each processor. The transcendental operations are estimated as described previously. Also, the AMAT for performing the potential energy look-up is calculated by using the Dinero cache simulator. Table 6.4 gives the parameters used for the models on various architectures that have been tested.

Each of these parameters can be plugged directly into equation 6.21 to obtain the runtime of the program on the given architecture.

For each processor and compiler pair, the model needs to be altered to reflect architectural and optimization differences. For the Intel E5-2680 Romley processor, the Intel compiler version 12.1.4 is used to compile the code. For the "non-vectorized"

Table 6.4: Parameters to the single core microprocessor model

Operation	Core i7 920	Intel E5-2680	AMD Opteron Istanbul	AMD Opteron 6272
Clock	2.67 GHz	3.5 GHz	2.6 GHz	2.4 GHz
c_{add}	4	4	5	6
c_{sub}	4	4	5	6
c_{mult}	6	6	5	6
c_{cos}	124	50	133	138
c_{sin}	124	55	134	126
c_{log}	150	84	79	81
c_{sqrt}	34	16	27	38
$AMAT_{lookup}$	4.03	4.03	3.05	4.96

version of this code, the loops to calculate the difference in potential energy are automatically vectorized using the AVX units by the compiler. To reflect this, equations 6.17 and 6.18 are divided by 4, the number of doubles that can be held in an AVX vector. For the SSE implementations on this processor compiler pair, all operations in equation 6.20, including equation 6.14 and excluding $AMAT_{lookup}$, are divided by the number of double precision elements stored in a vector, V=2. The Intel compiler is able to better overlap computations and memory accesses in the AVX implementation so all of the operations including $AMAT_{lookup}$ are divided by V=4. This is to account for multiple replicas being executed simultaneously.

Using the GNU gcc compiler version 4.6.1 on the AMD Opteron 6272, the model can be used as is for the non-vectorized implementation. The GCC compiler is not able to effectively optimize the SSE and AVX intrinsics. This could be due to the adoption of nontraditional AVX vectorized units: fusing two 128-bit units to create a single 256-bit wide vector unit [108]. Two cores share a single 256-bit unit, but each of the cores can execute 128-bit instructions simultaneously. To model the SSE implementation, nothing needs to be changed in the model as the resulting code executes with similar performance as the nonvectorized implementation. For the AVX implementation, the multiplication, addition, subtraction and square root operations are divided by two to account for the fused 128-bit vector units and the

low stream rates from cache to registers [108]. In addition, the memory subsystem has an extremely high L2 load to use latency (approximately 20 cycles) when compared to other the processors indicating a low memory throughput penalizing achievable performance. Since there are no vectorized instructions for the transcendental operations, these operations are performed serially and cannot be divided by these factors.

For the Intel Core i7-920 and gcc version 4.4.5, the model does not need to be altered to represent the nonvectorized code. For the SSE implementation, the multiplication, addition, subtraction, and square root operations are divided by V=2 to account for the combined replica processing. Again, since there are no vectorized instructions for the transcendental operations, these operations are performed serially and cannot be divided by this factor.

Using the Intel compiler version 12.1.2 on the AMD Opteron Hex-Core Istanbul processor, the model can be used as is for the nonvectorized implementation. For the SSE implementation, all of the operations are divided by V=2, including the transcendental functions as the compiler provides vectorized implementations through the SVML library.

All details about the compiler optimizations mentioned above were obtained by examining the resulting assembly code by hand. To test the claims made here, various compiler flags were enabled/disabled and the resulting assembly codes were compared.

Figures 6.2-6.5 shows the actual and predicted runtimes of various parameter runs on different microprocessor architectures. Overlaid onto each of these plots is the error of the predicted runtime with the error values listed on the right y-axis. Due to the long runtimes of this serial implementation, only a few runs are completed in order to show the model accuracy on various architectures and problem sizes.

It can be seen from these graphs that for the non-vectorized implementations, the performance model is accurate to within at most a little below 5% error (on the Intel Romley E5-2680 figure 6.3) and the average error is less than 2%. As for the vectorized implementations, the model is accurate for Intel processors with at

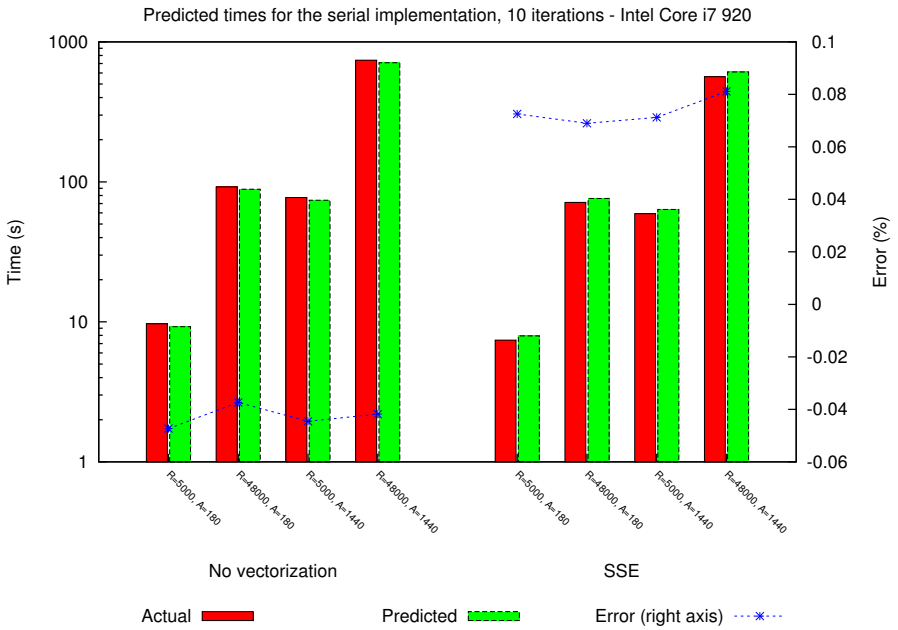


Figure 6.2: Serial, single core model accuracy for the Intel Core i7 920

most 8% error and the average error is less than 4%. For the AMD Opteron hex-core Istanbul processors 6.4, the vectorized implementation is accurate within 1%. However, the model is accurate within less than 16% for the vectorized version of the code on the AMD Opteron 6272 (figure 6.5). This is due to the inability for gcc to produce highly optimized vectorization code for the AMD Interlagos architecture and the new configurations of the architecture mentioned above. With the exception of the vectorized AMD Opteron 6272, the model error is acceptable when attempting to predict the runtime of the program across the different microprocessor architectures.

6.1.2 Parallel

At this point, it is logical to ask, "Running serial jobs is not very feasible for real problems, so how is this applicable to executing the program on multiple cores?" Before answering this, we should review how the MPI implementation works. There have been a few MPI implementations developed to test the scalability of the algorithm to thousands of cores. First, a traditional master-worker implementation

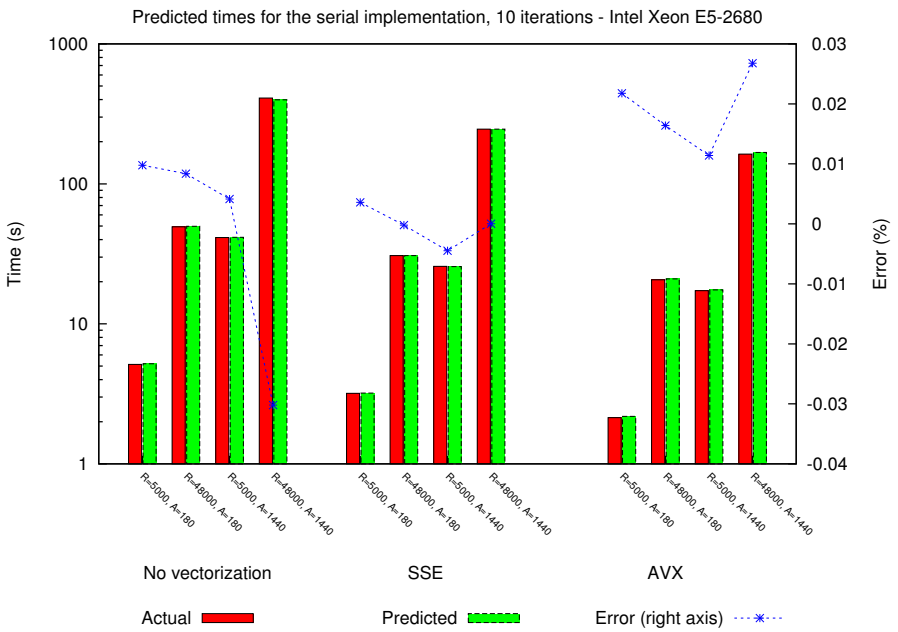


Figure 6.3: Serial, single core model accuracy for the Intel E5-2680

was developed where there is a central master that distributes work to all of the workers as needed. The master would iterate through the odd and even replicas, calculate the average neighboring replica information for each atom, package it with the corresponding RNG state information, and send the package to a worker that is requesting work. In the meantime, the master waits for more results from the worker indicating their desire for more work. Upon receipt of replica information from the master, each worker will attempt to move each atom in the replica according to the algorithm described in algorithm 2.1. Once complete, the results are sent back to the master and the worker will wait for more work from the master. A more in depth description of this implementation is given in section 3.3.4. The number and size of communications in this implementation are a major bottleneck in this implementation. No model for this is given because of the sheer lack of scalability of this method on small clusters.

To combat this limitation, the next implementation assigns work to each worker at the beginning of the program such that each worker has the same number of

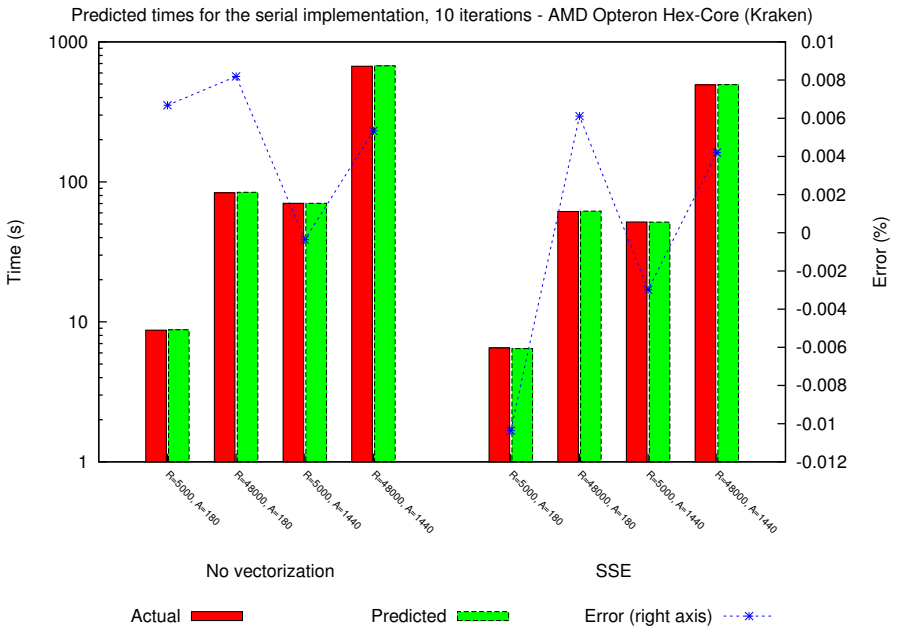


Figure 6.4: Serial, single core model accuracy for the AMD Opteron Istanbul

replicas. At the end of each odd or even replica processing, each worker will send the last/first replica in its workload to the neighboring workers. This is necessary for the displacement vector calculation for each atom as described in 3.2.2. This reduces the number of communications tremendously while reducing the bottleneck of a single master constantly handling all requests from each worker. However, the master is still a bottleneck for taking the snapshots as each worker must send its replica information to the master. Since the workers cannot send its replica information asynchronously to the master and continue to do work without risk of overwriting data, each worker must wait until its send is complete. At large numbers of workers (more than 480 as seen in testing), this becomes a major problem, dramatically reducing efficiency. Also, it is not possible to take advantage of the parallel filesystem by having a single master. More details on this implementation are supplied in section 3.3.4.

Finally, a third implementation avoids this issue by having multiple masters. Each worker has their local master that they will send snapshot information to. Each master will write their own snapshot and save files in order to take advantage of the

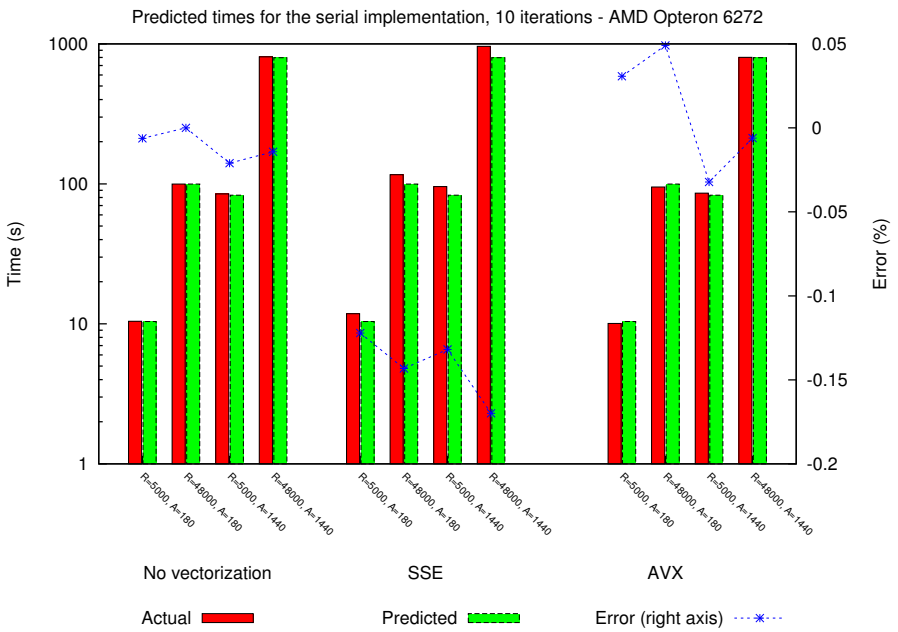


Figure 6.5: Serial, single core model accuracy for the AMD Opteron 6272

parallel filesystem (e.g., Lustre) if available. This allows for each worker to send their snapshot information in an efficient, timely fashion, giving more time to do more computational work. More details are available in section 3.3.4.

In its most general form, the runtime of the MPI implementation can be written as the sum of initialization/finalization time, worker time, snapshot time, and communication time (equation 6.22).

$$t_{MPI} = t_{init} + I * t_{worker} + S * t_{snapshot} + t_{final} \tag{6.22}$$

All of the MPI models use this equation in some form, as will be discussed below.

Coming back to the question, "How does the serial model help if all the important scientific work requires a parallel implementation?" As described in section 3.3.4 and above, each MPI worker process, with the exception of the masters, runs a serial version of the QSATS algorithm on a subset of replicas. Therefore, we are able to use this timing model to predict the runtime of each worker process. Equation 6.23 gives

Table 6.5: Parameters for the MPI models

Symbol	Description
B_{write}	Filesystem write bandwidth
B_{read}	Filesystem read bandwidth
B_{send}	Interprocess communication bandwidth
L	Interprocess communication latency

the time it takes for each for each worker to complete its work in a single iteration.

$$t_{worker} = \frac{R_w}{Frequency} * (c_{average} + c_{gaussian} + A * (c_{uniform} + P * c_{calcd^2} + c_{innerdv} + c_{outerdv} + c_{predetermine})) \quad (6.23)$$

These models are dependent on all of the factors described for the serial model plus the parameters listed in table 6.5

The models developed in this section are described in terms of the scalability/efficiency of the different implementations. In this dissertation, efficiency is calculated as the speed up of using the parallel implementation over the serial divided by the number of processors used:

$$\text{Efficiency} = \frac{t_{serial}}{\text{Procs} * t_{parallel}} \quad (6.24)$$

Since modern supercomputers such as Kraken [95] and Jaguar [2] charge by the compute hours used, the cost function in equation 6.25 is used to determine how the run will be "billed" given a specific set of parameters:

$$\text{Cost} = N * t_{MPI} \quad (6.25)$$

Single Master-Dependent Workers

A model for the single master, dependent worker implementation was created to show the lack of scalability of this method.

First, initialization can mostly be neglected for large problems and small numbers of workers. However, the initialization is a contributing factor to the lack of scalability thus is included in this model. At the start of the program, the master must read the save file if it is a continuation of a previous run. This can be modeled as the product of the size of the file and B_{write}, as shown in equation 6.26.

$$t_{read\ save} = \frac{R * (A * 3 + 8) * 8}{B_{read}} \tag{6.26}$$

If this is not a continuation run, the displacement vector is set to zero and the RNG state information is initialized as described in section 3.2.1. In this case, reading the save file is not necessary making this time zero.

Once the save file has been read in, the master distributes the work to each of the workers. The master must send two packets to each worker: R_w replica displacement information and R_w RNG state information arrays. Equation 6.27 models the work distribution time.

$$t_{distribute\ work} = \left(\frac{(3 * A + 8) * R_w * 8}{B_{send}} + 2 * L \right) * W \tag{6.27}$$

The sum of these two times (shown in equation 6.28) makes up the total initialization time.

$$t_{init} = t_{read\ save} + t_{distribute\ work} \tag{6.28}$$

Next, the work performed by the workers must be modeled in order to get an accurate time. Luckily, this is already developed for the most part. Equation 6.23 gives the approximation of the runtime time for each worker where $R_w = \frac{R}{W}$. This was already shown to be an accurate model for the serial implementation in section 6.1.1 so there will not be any further explanation here. The next item that needs to be modeled is the exchange of replica information that each child performs with the neighboring workers. For each iteration, each child sends $2 * 3 * A * V * 8 = 48 * A * V$ bytes of information. Sending the replica information to the neighbors

is asynchronously overlapped with receiving the data. Therefore, the time taken to do the exchanges exchanges is simply the time it takes for a single child to do the exchange as shown in equation 6.29.

$$t_{exchange} = \frac{48 * A * V}{B_{send}} + L \tag{6.29}$$

Third, the time taken to send a single snapshot from the worker to master is given in equation 6.30.

$$t_{singlesnap} = \frac{3 * A * R_w * 8}{11 * B_{send}} + L \tag{6.30}$$

However, despite the children sending all their snapshots simultaneously, the master can only receive a single snapshot at a time. Therefore, the snapshots are received serially, essentially synchronizing all of the children. Because of this, the real cost of the snapshot communications is given in 6.31.

$$t_{commsnap} = W * (t_{singlesnap}) \tag{6.31}$$

Once the master has all of the snapshots from the workers, all of the workers can continue with the next iteration. While the children are continuing their work, the master can simultaneously write the snapshot file to disk. Equation 6.32 gives the time taken to write a single snapshot.

$$t_{writesnap} = \frac{3 * A * R * 8}{11 * B_{write}} \tag{6.32}$$

Since there is an overlap of work with the master and workers, the simulation runtime excluding the initialization and finalization is maximum of the time it takes to write the snapshot and the time to perform S_I iterations, given in equation 6.33.

$$t_{simulation\ time} = \max\{t_{writesnap}, S_I * (t_{worker} + t_{exchange})\} + t_{commsnap} \tag{6.33}$$

Finally, the time it takes to finalize the program is another major contributing factor to the scalability of this implementation. Finalizing QSATS involves the time taken for the master to gather the final configurations from all of the children. The master gathers all of these configurations then writes them to disk. Equation 6.36 describes the time to finalize the program and write the save file.

$$t_{gather\ results} = \left(\frac{(3 * A + 8) * R_w * 8}{B_{send}} + 2 * L \right) * W \tag{6.34}$$

$$t_{write\ save} = \frac{R * (A * 3 + 8) * 8}{B_{write}} \tag{6.35}$$

$$t_{final} = t_{gather\ results} + t_{write\ save} \tag{6.36}$$

Adding these together, the total runtime of this MPI implementation is given by the sum of the previous times, as shown in equation 6.37.

$$t_{single\ master} = t_{init} + S * t_{simulation\ time} + t_{final} \tag{6.37}$$

$$\begin{aligned} &= t_{read\ save} + t_{distribute\ work} \\ &\quad + S * (\max\{t_{writesnap}, S_I * (t_{worker} + t_{exchange})\} + t_{commsnap}) \\ &\quad + t_{gather\ results} + t_{write\ save} \end{aligned} \tag{6.38}$$

In order to test this model, various numbers of runs were performed on the Kraken supercomputer. Due to time allocation limitations, extensive tests for this implementation are not possible as it would require a large amount of hours. Table 6.6 lists the Kraken parameters used in the model. All of the parameters used in the t_{worker} are in table 6.4 under the AMD Istanbul column. I/O performance numbers are obtained from running the IOZone Benchmark [109] on Kraken. These are by no means completely accurate as these numbers can vary depending on the load of the system at the time; however, they are suitable estimates. As for the communication bandwidth and latency, these are obtained from the Innovative Computing Lab HPC Challenge website that lists the results of various benchmarks on clusters and

Table 6.6: Parameters for the Kraken supercomputer

Parameter	Value
B_{write}	225.576 MB/s [109]
$B_{maxwrite}$	11.00 GB/s [83]
B_{read}	731.423MB/s [109]
B_{send}	1.6 GB/s [110]
L	9μs [110]

Table 6.7: Model results of the single master implementation

Single Master								
N	R	A	I	S	Runtime	Predicted	Error	Efficiency
24	25576	180	1000	10	209.08	191.88	-8.22%	95.4%
48	20868	448	1000	10	211.65	191.57	-9.49%	97.0%
120	80444	448	1000	10	321.52	295.66	-8.04%	97.0%
480	53648	1440	1000	10	179.71	168.17	-6.42%	91.4%
960	111244	1440	1000	10	199.92	188.92	-5.50%	84.3%
120	80444	448	100	100	37.66	40.43	7.36%	71.6%

supercomputers around the world [110]. Table 6.7 gives a list of runtimes using various parameters that test the scalability and the accuracy of the model.

As can be seen, the model is accurate within less than 10%. Inaccuracies in the serial model (described in the previous section) and lack of current bandwidth, latency and I/O speed values contribute to the inaccuracy of the model. For instance, if there is another job running that is extremely I/O intensive, the Lustre filesystem could be overloaded with requests thus reducing the amount of available bandwidth. This table is also shown in figure 6.6.

As mentioned in the previous chapter, this implementation has the downfall of scaling poorly due to the large amount of communications the master has to handle as the number of children increase. Using this model, it is possible to see what the scalability, runtime, and cost of using various parameters indicating this limitation and hinting what should be addressed when developing a new implementation. The plots in figure 6.7 show the predicted runtime, efficiency, and cost for running the

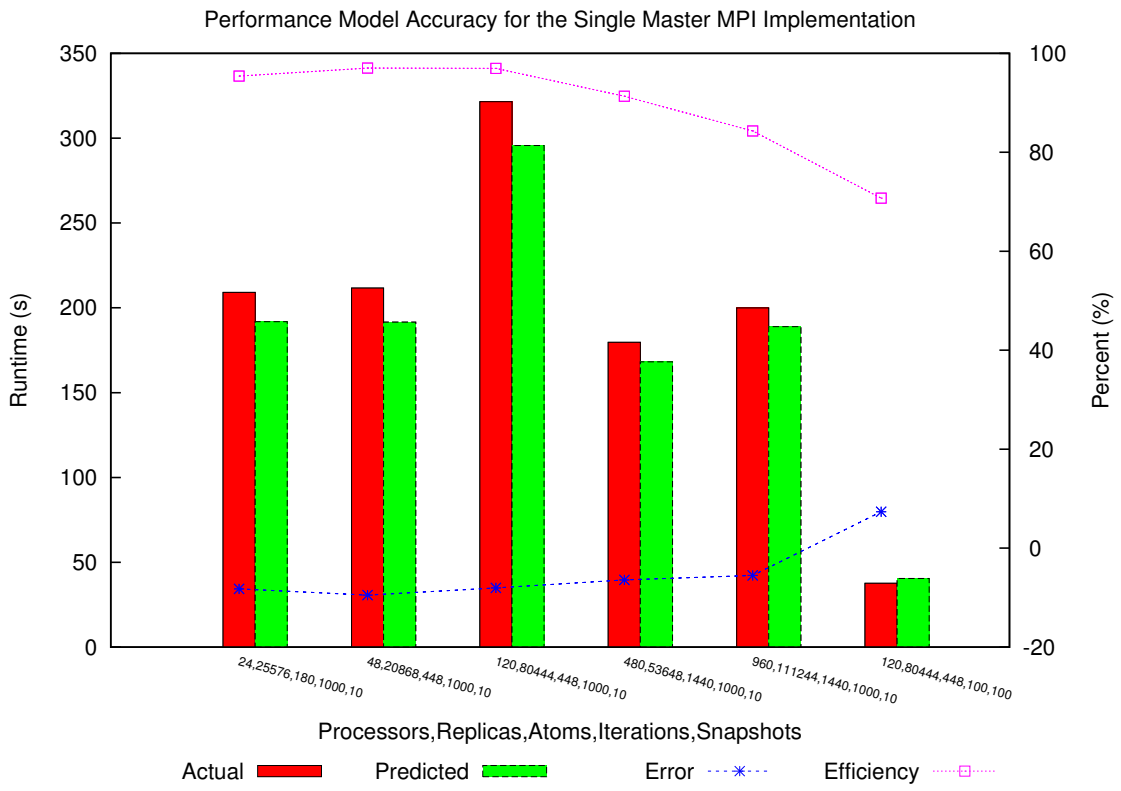
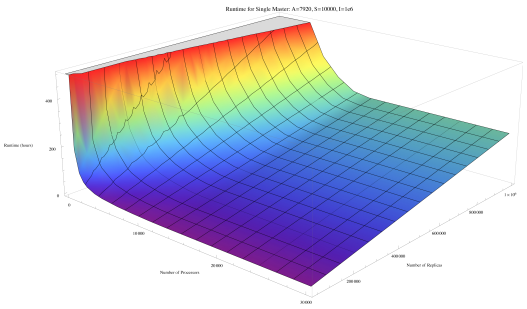


Figure 6.6: Model results of the single master implementation

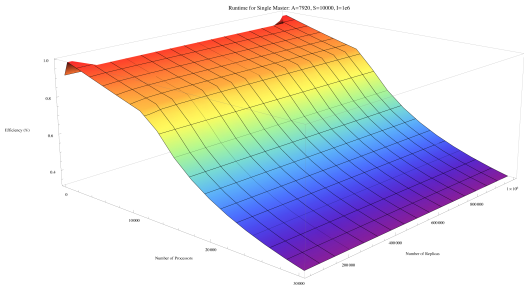
QSATS program on the Kraken supercomputer for 7920 atoms, 1 million iterations, 50,000 replicas, and 10,000 snapshots.

A note: these graphs are zoomed in to show the shape of the plots; otherwise with small number of processors and high numbers of replicas, the plot for the time would dominate. This is why for smaller number of processors it seems the plot is "cut off".

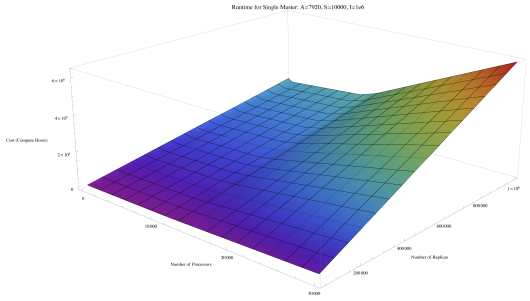
An interesting fact shown in figure 6.7a is that there is a point where adding more processors to deal with the same sized problem has no benefit. In fact, looking at the corresponding cost function in figure 6.7c, we can see that simply adding more processors to the problem only wastes allocation hours. Figure 6.7b shows that this increase in cost is due to a major decrease in efficiency so that we are not effectively utilizing the resources. From these plots, it can be seen that the single master implementation should be able to scale with more than 80% efficiency up to



(a) Single master-dependent workers runtime model



(b) Single master-dependent workers efficiency model



(c) Single master-dependent workers cost model

Figure 6.7: Single master-dependent workers models

about 10,000 processors at which point the performance drops dramatically. However, even 80% efficiency is not satisfactory as there is a 20% waste of compute resources, thus the need for a more advanced implementation.

Multiple Masters - Dependent Workers

As a reminder, the number of processors, N, is split into M work groups where each group has a single master and $W_g = N/M - 1$ workers.

First, initialization can mostly be neglected for large problems and small numbers of workers. However, the initialization can be a contributing factor to the scalability performance thus is included in this model. At the start of the program, each master must read its save file if it is a continuation of a previous run. This can be modeled

as shown in equation 6.39.

$$t_{read\ save} = \frac{R * (A * 3 + 8) * 8}{M * B_{read}} \tag{6.39}$$

If this is not a continuation run, the displacement vector is set to zero and the RNG state information is initialized as described in section 3.2.1. In this case, reading the save file is not necessary making this time zero.

Once the save file has been read in, each master distributes the work to each of the workers. The master must send two packets to each worker: R_w replica displacement information and R_w RNG state information arrays. Equation 6.40 models the work distribution time.

$$t_{distribute\ work} = \left(\frac{(3 * A + 8) * R_w * 8}{B_{send}} + 2 * L \right) * W_g \tag{6.40}$$

The sum of these two times makes up the total initialization time:

$$t_{init} = t_{read\ save} + t_{distribute\ work} \tag{6.41}$$

Next, the work performed by the workers must be modeled in order to get an accurate time. Luckily, this is already developed for the most part. Equation 6.23 gives the approximation of the runtime time for each worker where $R_w = \frac{R}{W}$. This was already shown to be an accurate model for the serial implementation in section 6.1.1 so there will not be any further explanation here. The next item that needs to be modeled is the exchange of replica information that each child performs with the neighboring workers, which is the same as in the single master case given in equation 6.29.

Third, the time taken to send a single snapshot from the worker to master is given in equation 6.30. As with the single master case, each master can only receive a single snapshot at a time. Therefore, the snapshots are received serially, essentially synchronizing all of the children. Because of this, the real cost of the snapshot

communications is given in 6.42.

$$t_{commsnap} = W_g * (t_{singlesnap}) \tag{6.42}$$

Once each master has all of the snapshots from its workers, all of the workers can continue with the next iteration. While the children are continuing their work, the master can simultaneously write the snapshot file to disk. Also, each master writes their file individually to the filesystem simultaneously with the other masters. This allows for maximum throughput to the parallel filesystem. Equation 6.43 gives the time taken to write a single snapshot.

$$t_{writesnap} = \frac{3 * A * R * 8}{11 * M * B_{write}} \tag{6.43}$$

Since there is an overlap of work with the master and workers, the simulation runtime excluding the initialization and finalization is maximum of the time it takes to write the snapshot and the time to perform S_I iterations, given in equation 6.44.

$$t_{simulation\ time} = \max\{t_{writesnap}, S_I * (t_{worker} + t_{exchange})\} + t_{commsnap} \tag{6.44}$$

Finally, the time it takes to finalize the program is another major contributing factor to the scalability of this implementation. Finalizing QSATS involves each worker sending the final configurations for its replicas to the master. The master gathers all of these configurations then writes them to disk. Equation 6.47 shows the time to finalize the program and write the save file.

$$t_{gather\ results} = \left(\frac{(3 * A + 8) * R_w * 8}{B_{send}} + 2 * L\right) * W_g \tag{6.45}$$

$$t_{write\ save} = \frac{R * (A * 3 + 8) * 8}{M * B_{write}} \tag{6.46}$$

$$t_{final} = t_{gather\ results} + t_{write\ save} \tag{6.47}$$

Table 6.8: Accuracy of the multi-master MPI model

N	M	R	A	I	S	Actual	Pred.	Error	Eff.
960	15	109968	1440	1k	10	134.68	130.22	-3.31%	95.4%
1920	12	122112	1440	10k	100	729.06	711.65	-2.93%	92.2%
3840	15	153000	1440	10k	100	473.25	459.38	-2.93%	89.4%
7668	142	210728	1440	100k	1000	3372.74	3110.80	-7.77%	85.6%
15.3k	300	120000	3584	100k	1000	2863.92	2639.93	-7.82%	71.6%

Adding these together, the total runtime of this MPI implementation is given by the sum of the previous times, as shown in equation 6.48.

$$t_{multiple\ master} = t_{init} + S * t_{simulation\ time} + t_{final} \tag{6.48}$$

$$\begin{aligned} &= t_{read\ save} + t_{distribute\ work} \\ &\quad + S * (\max\{t_{writesnap}, S_I * (t_{worker} + t_{exchange})\} + t_{commsnap}) \\ &\quad + t_{gather\ results} + t_{write\ save} \end{aligned} \tag{6.49}$$

As before, in order to test this model, various numbers of runs were performed on the Kraken supercomputer. Due to time allocation limitations, extensive testing for this implementation is not possible as it would require a large amount of hours. Table 6.6 lists the Kraken parameters used in the model. All of the parameters used in the t_{worker} are in table 6.4 under the AMD Istanbul column.

Table 6.8 gives a list of runtimes using various parameters that test the scalability and the accuracy of the model. Figure 6.8 visualizes this information in terms of performance as shown in equation 5.3.

What can be drawn from this information is that the model is accurate within less than 8%. However, it must be noted that for the 7,668 and 15,300 core runs the bandwidth from the workers to the masters was significantly reduced due to the 30MB/s and 6MB/s, respectively, described in section 5.1.4. Using the ideal bandwidth of 1.6GB/s as in the first three parameter sets results in a model error around 30%. This bandwidth was verified by outputting the total snapshot time

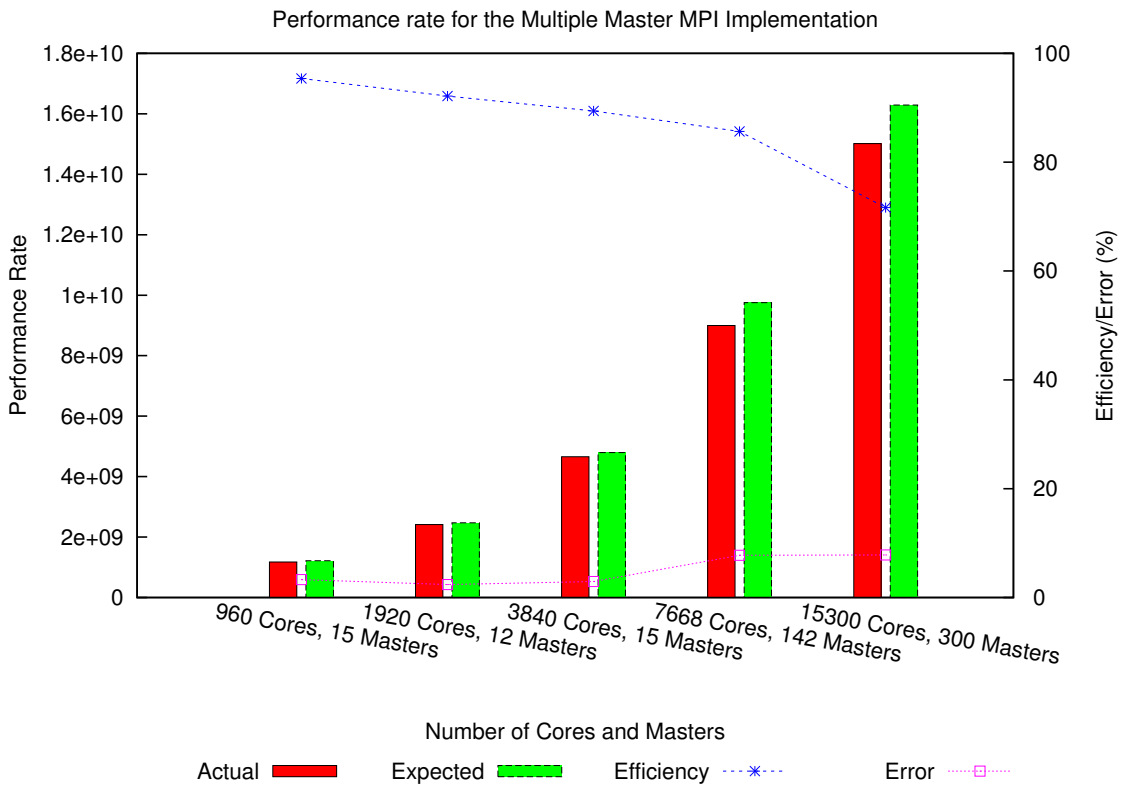
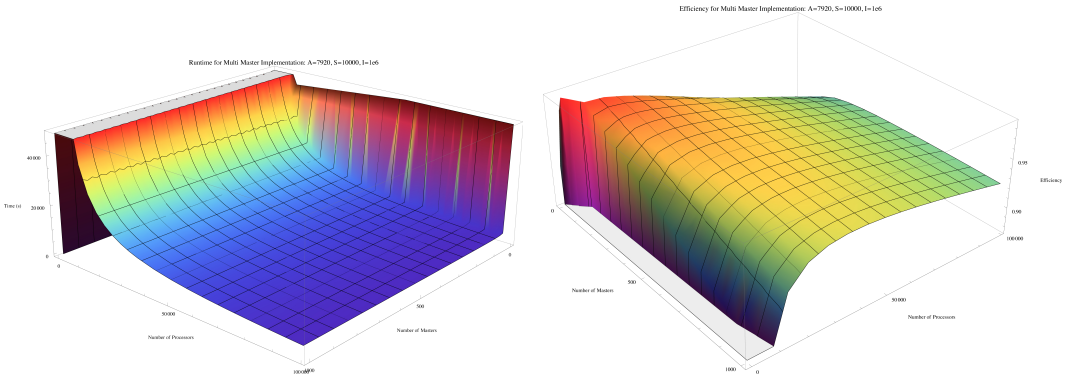


Figure 6.8: Accuracy of the multi-master scalability model

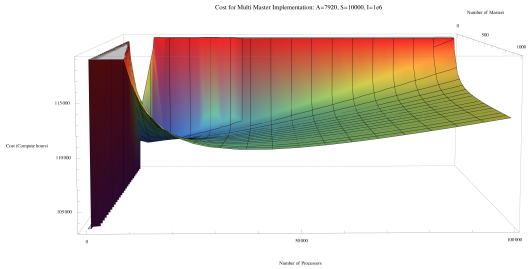
for all of the snapshots. The computation time is accurate within less than 5% for each of these runs. The reason why the first three runs are able to modeled with the ideal bandwidth is because these runs were performed at a completely different time, approximately 1 month prior to seeing the bandwidth issues. As mentioned in section 5.1.4, the bandwidth issue was confirmed to have nothing to do with the algorithm by rerunning some of the other parameter sets to observe the variances in bandwidth. One possible explanation for this degradation in bandwidth could be that other applications running on the machine are bandwidth heavy causing contention on the network thereby reducing bandwidth.

Figure 6.9 shows the runtime, efficiency, and cost for running the QSATS program on the Kraken supercomputer for 7920 atoms, 1 million iterations, 50,000 replicas, and 10,000 snapshots, assuming the ideal bandwidth (table 6.6) and not the troublesome bandwidth described above.



(a) Multi master-dependent workers runtime model

(b) Multi master-dependent workers efficiency model



(c) Multi master-dependent workers cost model

Figure 6.9: Multi master-dependent workers models

Figure 6.9 is different than the last plots as it shows the runtime, efficiency, and cost in terms of number of processors and masters. This exemplifies the effect of using multiple masters on the program and provides insight on an optimal configuration of masters. In order to choose the optimal number of masters to use with a given problem, equations 6.50 to 6.53 shows a method of determining the number of masters that minimizes the runtime.

$$M_{min} = Solve[t'_{multiple\ master}(M) == 0 \&\& M > 0, M] \tag{6.50}$$

$$Div_P = t'_{multiple\ master}(PrevDivisor[N, M_{min}]) \tag{6.51}$$

$$Div_N = t'_{multiple\ master}(NextDivisor[N, M_{min}]) \tag{6.52}$$

$$\tilde{M}_{min} = (Div_P < Div_N ? Div_P : Div_N) \tag{6.53}$$

First, the minimum of the time function is found by taking the derivative with respect to M, setting the result to zero, and solving for M to find M_{min}. M_{min} is the point at which the runtime is minimized; however, the number of masters must be an integer. Also, for effective load balancing, it is required that N is divisible by the number of masters. Therefore, it is necessary to find what divisor of the N results in a near minimum runtime. To do this, the first divisors of N less than and greater than M_{min} are chosen, input into the model, and compared. The one resulting in the lower runtime is the near-optimal number of masters that should be used. The PrevDivisor and NextDivisor functions are used to find the previous and next divisor of N from M_{min}. Using this method, we can predict the optimial number of master to use that will result in the highest performance.

It is possible to predict the runtime and scalability of the microprocessor implementation using the models developed in this section. By simply counting operations and looking up the corresponding documented latencies, we can obtain predicted runtimes that are accurate within less than 10%.

6.2 Graphics Processing Units

Unlike the microprocessor implementation, the GPU implementation is 100% memory bound as determined by using the NVIDIA Visual Profiler. Because of this, the model can safely ignore any computations as they are completely hidden by memory accesses. Using this knowledge, this section creates a performance model that includes various memory aspects of the Tesla and Fermi architectures of NVIDIA GPUs. Therefore, it is necessary to determine the cost of accessing the various forms of memory. The global memory latency is supplied by NVIDIA to be in the range of 400-800 clock cycles [38]. The texture memory is not explicitly given by NVIDIA; however, Wong et al. [111] have developed benchmarking tools that were intended for the Tesla architecture. According to the benchmarks, texture memory accesses have a latency of 230 clock cycles on the Tesla architecture and 219 on the Fermi architecture.

To develop a performance model, this work first attacked the model for the Tesla c1060 (values subscripted with T) then adapted that model to describe the Fermi architecture (values subscripted with F).

6.2.1 NVIDIA Tesla c1060

The first step to developing the GPU performance model is to count the number of memory accesses in the CUDA kernel. The uniform random number generator contains 18 global memory accesses on average. Due to some conditionals in the code to avoid negative and zero values, there are occasionally more than 18 memory accesses; however, this is rare so the extras accesses can safely be ignored.

$$\text{Uniform}_T = 18 * global \tag{6.54}$$

To generate a Gaussian random number there is the cost of generating two uniform random numbers plus three more accesses to store/read from the state vector if a number has already been generated and stored. Saying this it is also only necessary to generate two uniform random numbers every other time the function is called; therefore, the cost of generating a Gaussian random number is only the cost of one uniform number plus 3 global accesses.

$$\text{Gaussian}_T = \text{Uniform}_T + 3 * global \tag{6.55}$$

Next, the cost of reading the current atom position and calculating the new position is determined. The current atom position is read from global memory and stored into registers. To calculate the projected atom position, two global memory accesses for each component is required to calculate the average of the neighboring replicas. Once the average is calculated, the Gaussian displacements are added to the average accounting for three Gaussian random number generations.

$$\begin{aligned}\text{OldNew}_T &= 3 * global + 2 * 3 * global + 3 * \text{Gaussian}_T \\ &= 9 * global + 3 * \text{Gaussian}_T \end{aligned} \tag{6.56}$$

To calculate the potential energy, a total of $2 * P$ table look-ups are read from the texture memory. One of the reasons why this is purely memory bound is due to the use of the hardware linear interpolation so the calculations associated with this step is safely ignored. To determine the indexes into the potential energy look-up table, there are a total of $4 * P$ global memory accesses to calculate the square distance between the positions and all of the neighbors. However, since this is a pretty tight loop, these global memory accesses are highly pipelined so the cost of each of these accesses are counted with a cost of 1.

$$\text{Potential}_T = 4 * P + 2 * P * texture \tag{6.57}$$

The rest of the kernel consists of generating a uniform random number to determined whether to accept or reject the move, three global memory accesses to store the new position, and two global memory accesses to account for the decision of accept or reject.

$$\text{AccRej}_T = \text{Uniform}_T + 5 * global \tag{6.58}$$

Summing equations 6.56-6.58 together accounts for all of the memory accesses within a single thread for the movement of a single atom. Multiplying this sum by the number of atoms, A, will give the total number of cycles it takes to move all of the atoms in a replica per thread per iteration.

$$\text{ThreadCycles}_T = \text{OldNew}_T + \text{Potential}_T + \text{AccRej}_T \tag{6.59}$$

However, we cannot simply multiply this by the number of replicas to account for an entire simulation as this would be neglecting architectural features from which this program greatly benefits such as memory coalescing. One architectural feature that affects the runtime is the assignment of thread blocks to streaming multiprocessors (SMs). Once the SMs have been assigned blocks, the block scheduler begins to give more blocks to each of the SMs. Each SM computes at the same time. Therefore, to have an efficient execution, it is imperative to use enough blocks to fill all of the SMs equally. Otherwise, if some SMs, have more blocks than the others, the runtime of the kernel is dependent on the execution of the SM with the most blocks. This results in a "stair-step" function that has steps as wide as the number of SMs. Because of this, it is best to adapt this model to describe the runtime of the algorithm at the block level rather than thread level. To do this, we multiply equation 6.59 by the number of threads per block, 64, and a ceiling function of the number of SMs used. However, one of the benefits of this implementation is the ability to take advantage of coalesced memory accesses. We can divide the number of threads per block by the number of doubles that can fill the data bus from global memory to the SM to give the number of coalesced reads for a single access to memory. To get the time in seconds to perform a single iteration, we can just divide this by the clock rate of the GPU.

$$\text{KernelTime}_T = \frac{64}{\frac{DatapathWidth}{8}} * \frac{\text{ThreadCycles}_T}{Clock} * Ceiling(\frac{Blocks}{SM}) \tag{6.60}$$

In the case of the Tesla c1060, there are a total of 30 SMs each with 8 streaming processors each for a total of 240 cores. This model already accounts for the total number of cores so there are no extra factors involved with the equation in 6.60.

To account for snapshotting, the model adds the time it takes to transfer all of the replica configurations from the device to the host. Since there is a risk of data overwrite/corruption, this transfer cannot be done asynchronously. However,

once the data is on the host, the host process can write the data to disk while the GPU continues with more simulations. This is accomplished by having a thread waiting for the transfer to complete, at which point it will write the 11^{th} replica to disk while the GPU continues with the simulation. To model the snapshotting process, the time required to transfer the replica information is simply the size of the packet transferred divided by the transfer bandwidth using pinned memory (obtained from the bandwidthTest program in the SDK). Since writing the snapshot to file is performed simultaneously with GPU execution, the runtime is dependent on the slower of the two; therefore, the maximum runtime of the two is used to model the overall time. This is shown in equation 6.63.

$$FromDevice = \frac{2 * A * R}{B_{pinned}} \tag{6.61}$$

$$WriteSnapshot = \frac{2 * A * R}{11 * B_{write}} \tag{6.62}$$

$$\begin{aligned} GPUTime_T = S * max\left\{S_I * \text{KernelTime}_T, WriteSnapshot\right\} \\ + FromDevice \end{aligned} \tag{6.63}$$

Initialization costs are ignored in this model for various reasons. The initialization times seems to varying across platforms, GPU models, installations, and which GPU in a node is used. For instance, on one of the machines used for testing, there are two Tesla c1060 cards available. The initialization cost for the cost for device 0 was sometimes larger than using device 1 and sometimes the other way around. There was no real consistency with the runtime to determine an accurate model However, as the length of the simulation increases, the cost for initialization is almost completely hidden and can be safely ignored.

Using the parameters in table 6.9, the model in equation 6.63 can be used to predict the runtime of the program on the Tesla c1060 cards.

Table 6.9: Tesla c1060 model parameters

Parameter	Description	Value
global	Global memory latency	400 [38]
texture	Texture memory latency	230 [111]
SM	Number of SMs	30 [92]
B_{pinned}	Device to host pinned memory bandwidth	5.1 GB/s [92]
Clock	Core clock frequency	1.3 GHz [92]

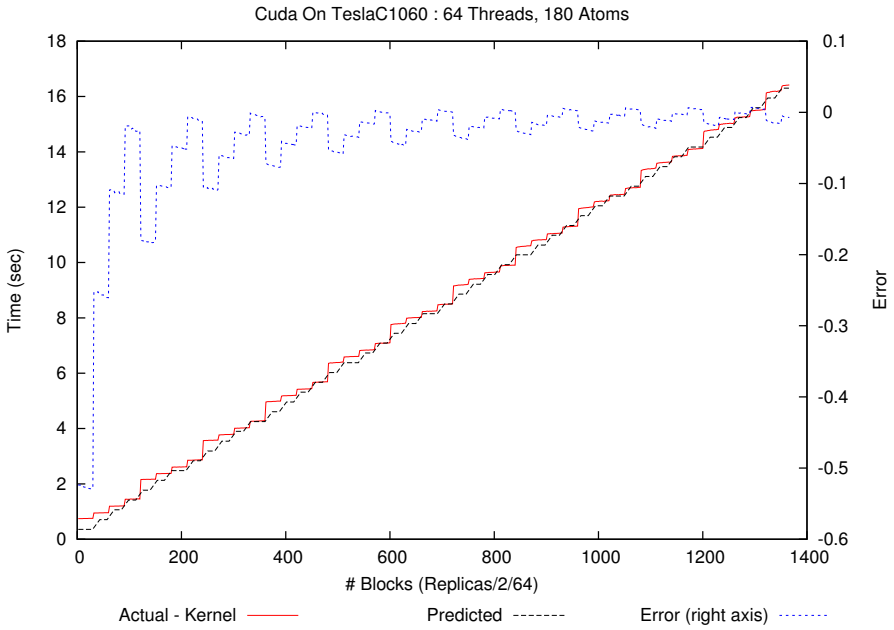


Figure 6.10: Model accuracy for the TeslaC1060 GPU - 180 Atoms

The figures in 6.10-6.12 show the model performance of varying the amount of work the GPU has to perform in terms of blocks (x-axis). Each figure uses a different sized crystal lattice.

It can be seen in these plots that the model accurately represents the runtime of the program to less than 3% error when the number of blocks is large. The large errors with fewer blocks is simply due to the small runtime making the relative error much larger. Using this model it is possible to tweak the various parameters such as the global and texture memory latencies, clock rate, and number of SMs to predict the performance impact that new architectural features would have on the runtime of this application.

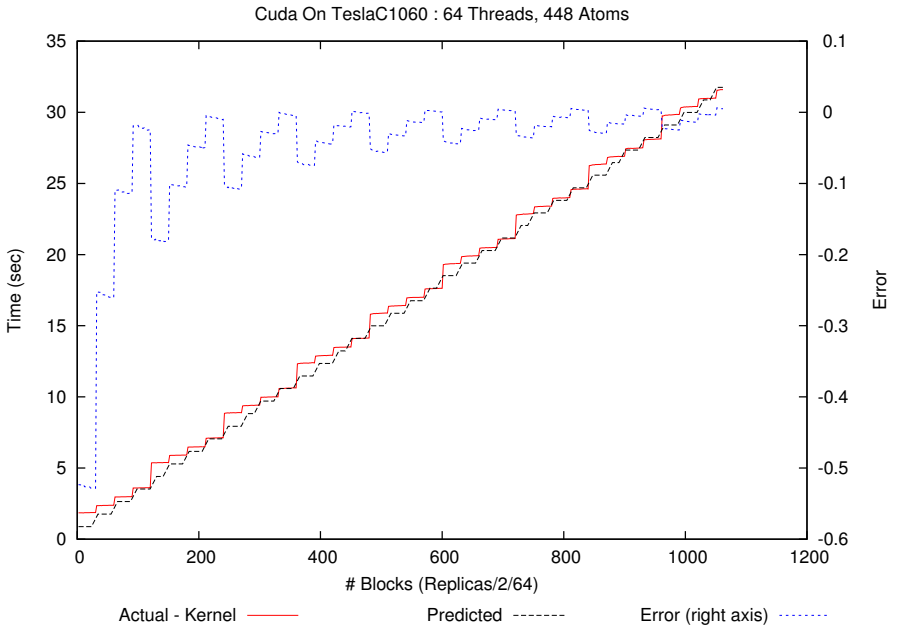


Figure 6.11: Model accuracy for the TeslaC1060 GPU - 448 Atoms

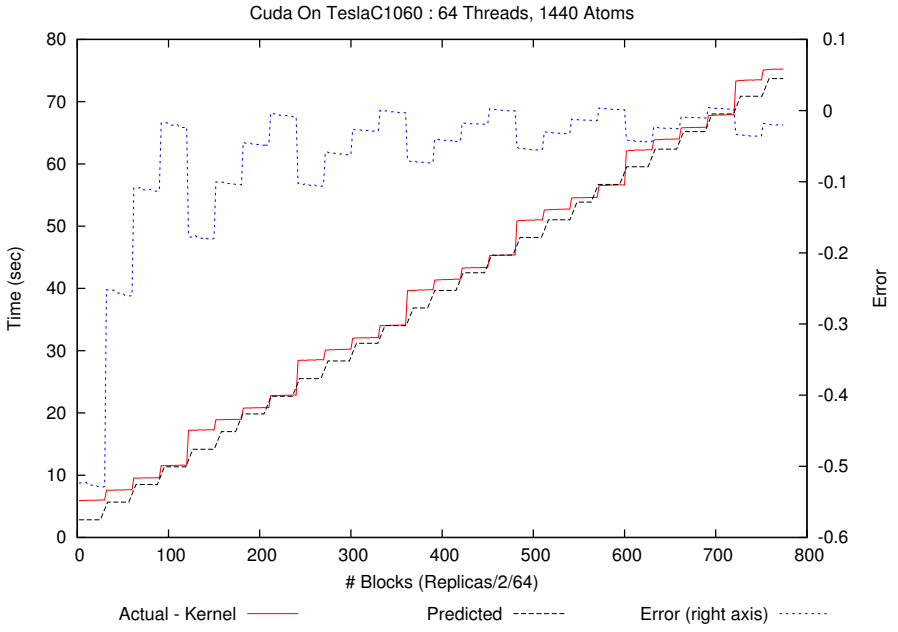


Figure 6.12: Model accuracy for the TeslaC1060 GPU - 1440 Atoms

6.2.2 Fermi Architecture

As mentioned in the previous chapters, the Fermi architecture provides many new features that enhance the performance of codes running on GPU devices. One of which is the number of compute cores and SMs. There are an increased number of cores from the Tesla architectures (2x more in the GeForce GTX 480 and 2.133x more in the Tesla M2090). Another new feature is the presence of a configurable 16KB or 48KB L1 hardware managed cache for global memory accesses per SM. Third, the Tesla architecture grouped sets of 3 SMs into a texture processing cluster (TPC) where each cluster shares a single texture unit. The Fermi architecture has made the improvement to give each SM access to its own private texture unit. These three new features have a dramatic effect on the performance of applications, especially for memory bound applications such as QSATS. To account for these new features, the Tesla model needed to be tweaked.

To account for caching, it was necessary to dive back into the code to determine what portions of the code exhibit good caching behaviors. Since accesses to cached memory are much faster than accessing global memory, this model counts cached accesses as free (i.e., zero latency). First, the uniform random number generator memory accesses can be reduced dramatically. Recall, the random number generator uses six states. Each time the uniform random number generator is called, these six states must be read from global memory multiple times. However, in the case of the Fermi architecture, the states are read from global once (i.e., six global memory accesses) and cached so that all subsequent accesses have a much lower access penalty.

$$\text{Uniform}_F = 6 * global \tag{6.64}$$

Likewise for the Gaussian random number generator, there are three extra state accesses but only two of them are full penalty global memory accesses. In addition, since the uniform random number generator is called twice in a single call to the Gaussian generator, the first call is "full" cost as shown in equation 6.64 while the

second call is fully cached making it free. Therefore on average, the URNG penalty is incurred .5 times per Gaussian call.

$$\text{Gaussian}_F = \frac{1}{2} * \text{Uniform}_F + 2 * global \tag{6.65}$$

Since the SMs are no longer grouped into TPCs, there is less contention for the use of the texture units. This enables the potential energy texture look-ups to be highly pipelined with less interference from other SMs, changing equation 6.57 to

$$\text{Potential}_F = 4 * P + P * texture \tag{6.66}$$

To account for the increased number of cores, this model has a dividing factor, γ, that represents the factor of number of cores the Fermi GPU in comparison to the Tesla c1060 (2x more in the GeForce GTX 480 and 2.133x more in the Tesla M2090). All of these improvements make up the kernel model shown in equation 6.70.

$$\begin{aligned} \text{OldNew}_F &= 3 * global + 2 * 3 * global + 3 * \text{Gaussian}_F \\ &= 9 * global + 3 * \text{Gaussian}_F \end{aligned} \tag{6.67}$$

$$\text{AccRej}_F = \text{Uniform}_F + 5 * global \tag{6.68}$$

$$\text{ThreadCycles}_F = \text{OldNew}_F + \text{Potential}_F + \text{AccRej}_F \tag{6.69}$$

$$\text{KernelTime}_F = \frac{64}{\frac{DatapathWidth}{8}} * \frac{\text{ThreadCycles}_F}{\gamma * Clock} * Ceiling(\frac{Blocks}{SM}) \tag{6.70}$$

Table 6.10: Fermi model parameters

Parameter	Description	GTX480	M2090
global	Global latency	400 [38]	
texture	Texture latency	219 [111]	
SM	Number of SMs	15	16 [92]
B_{pinned}	Device to host pinned bandwidth	5.1 GB/s	6.5GB/s [92]
Clock	Core clock frequency	1.4 GHz	1.3 GHz [92]
γ	Factor of cores over c1060	2	2.133

The same approach is taken with the Fermi architecture as with the Tesla to account for the cost of snapshotting, as shown in equation 6.71.

$$\begin{aligned} \text{GPUTime}_F = S * max\left\{S_I * \text{KernelTime}_F, WriteSnapshot\right\} \\ + FromDevice \end{aligned} \tag{6.71}$$

Using the parameters in table 6.10, the model in equation 6.71 can be used to predict the runtime of the program on the following Fermi cards: GeForce GTX 480 and Tesla M2090.

The figures in 6.13- 6.18 show the model performance of varying the amount of work the GPU has to perform in terms of blocks (x-axis). Each figure uses a different sized crystal lattice.

From these plots, it can be seen that the model is accurate within less than 5-10% with large problem sizes. The larger errors with the smaller problem sizes is a result of the small runtime making the relative errors larger. Problems this small are not scientifically important so there is no need to run jobs of this size. Therefore, the important portion of the model is for larger problem sizes. As with the previous GPU model, it is possible to tweak various parameters in the model to accurately see what the effect new features in future GPU models have on runtime. Also, when planning a full run on these devices, it is possible to use this model to determine how much time allocation is needed to complete the run.

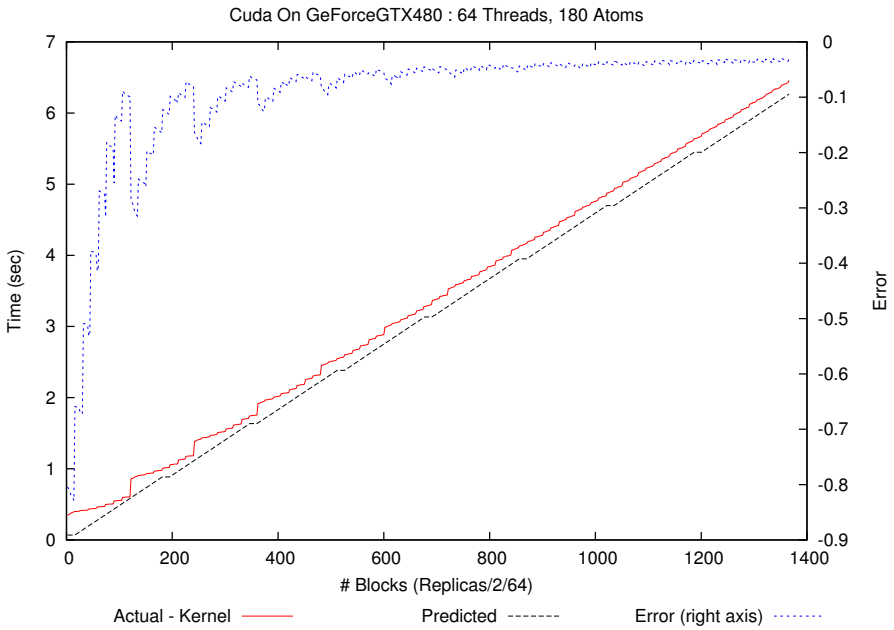


Figure 6.13: Model accuracy for the GeForce GTX 480 GPU - 180 Atoms

It can seen that for this memory bound application, simply counting memory accesses and summing their latencies can provide for accurate performance modeling. To examine optimizations performed by the compiler such as loop unrolling, it is necessary to look through some of the PTX code; however, most of the accesses can be counted directly from the CUDA code. In order to account for architectural differences across generations, it is required to alter the model. Using these guidelines, it is possible to develop models for memory bound, computational science GPU applications.

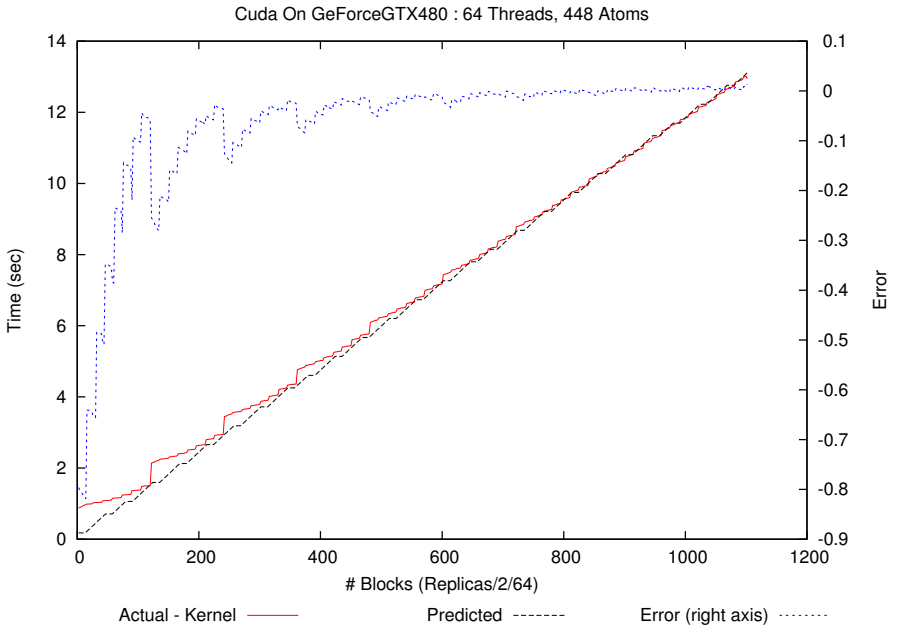


Figure 6.14: Model accuracy for the GeForce GTX 480 GPU - 448 Atoms

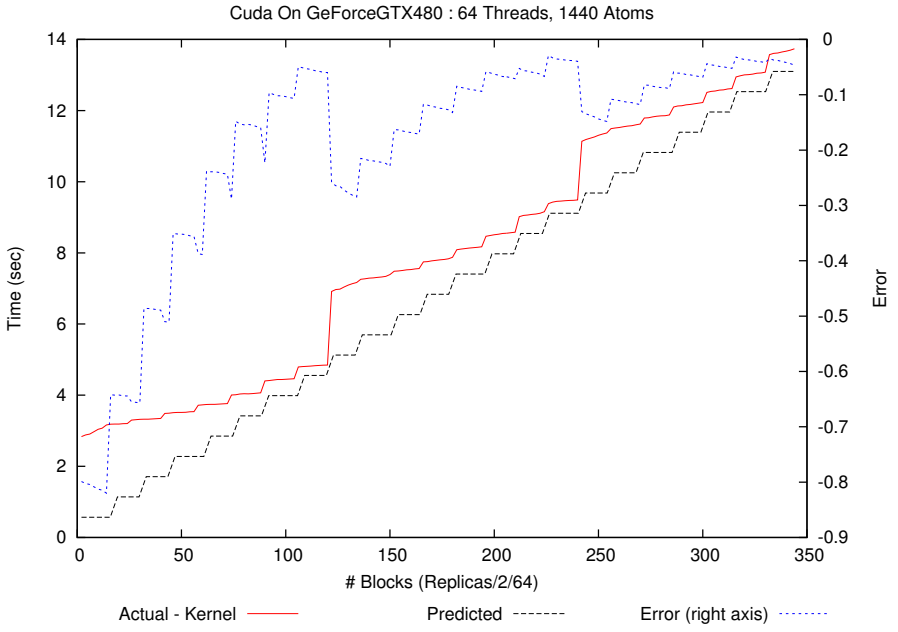


Figure 6.15: Model accuracy for the GeForce GTX 480 GPU - 1440 Atoms

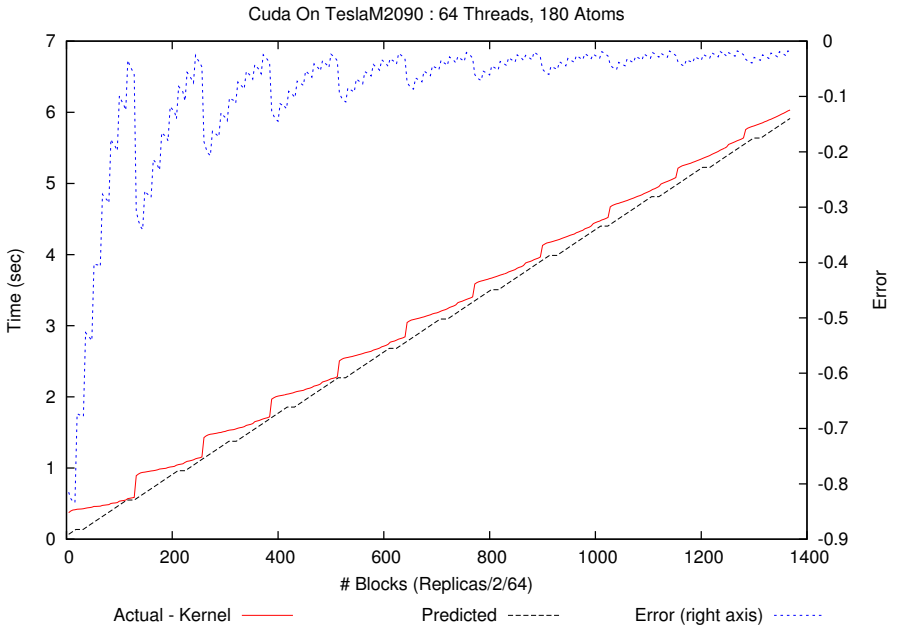


Figure 6.16: Model accuracy for the Tesla M2090 GPU - 180 Atoms

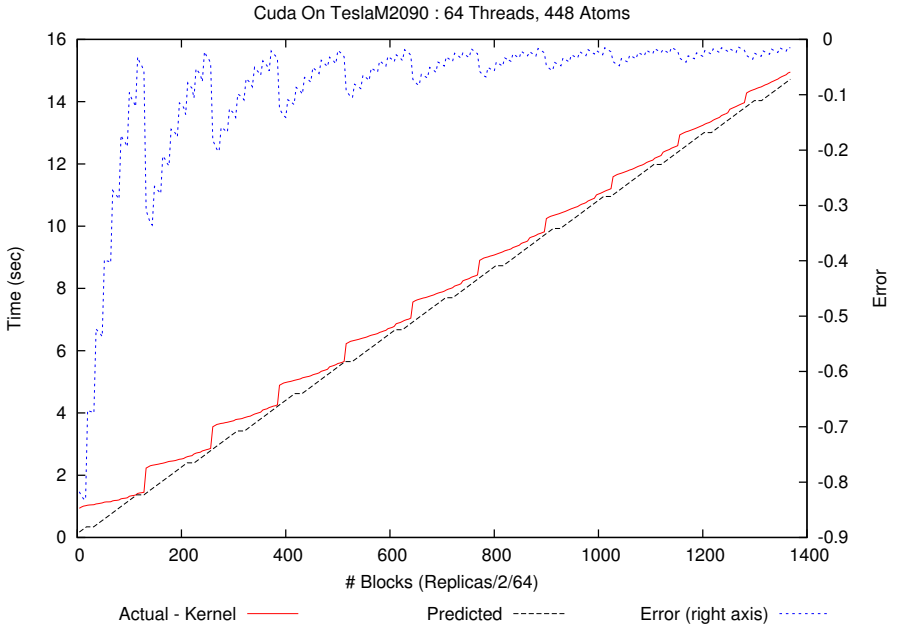


Figure 6.17: Model accuracy for the Tesla M2090 GPU - 448 Atoms

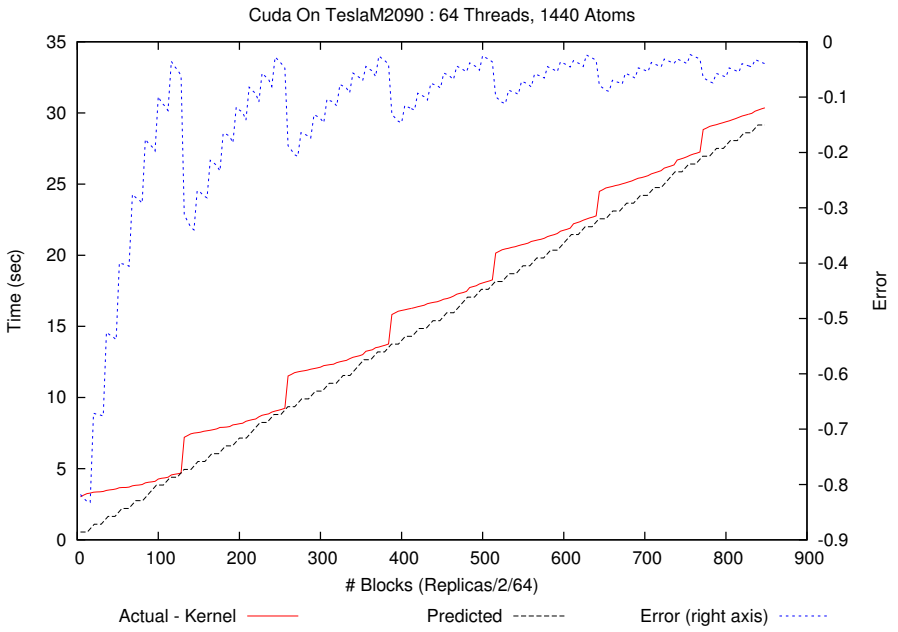


Figure 6.18: Model accuracy for the Tesla M2090 GPU - 1440 Atoms

Chapter 7

Future Work and Conclusions

Within the last decade, there has been a massive growth of high performance architectures providing developers the tools to tackle some of science's biggest problems. Emerging architectures such as multicore processors, graphics processing units, and the Intel MIC assist developers and scientists in answering scientific questions that were previously thought to be impossible. This dissertation investigates the use of multiprocessors (intra- and internode), multiple graphics processing units, and the Intel MIC to determine their practicability in regards to the computational chemistry Quantum Monte Carlo method, Variational Path Integral (VPI). This chapter summarizes contributions, results, and pinpoint areas of future research.

7.1 Future Work

Although this dissertation provides a large portion of work on the QSATS program, there is still plenty to explore in the future. In terms of science, three-body interactions can be added to the code to explore their effect on the movement of atoms in a crystal in comparison to the two-body explored in this work. The basic framework for the GPU and MPI implementations is already laid out and should be a straightforward addition to the code. Doing the three-body interactions increases the

memory transaction requirements and the number of calculations. The performance models can be altered to account for these once the algorithm has been planned out.

From a computer engineering perspective, the Intel MIC architecture can be explored more with this application once a final product is available. This involves tweaking some of the code to account for any minor architecture changes from the alpha boards used in this work.

Another possible improvement to the code as a whole is to discretize the crystal movement space and eliminate the need for floating point arithmetic by using integers, as discussed in [112]. This optimization could have major impacts on performance as using integers is significantly less computationally expensive. Also, using 32-bit integers (if able to obtain necessary accuracy) would reduce the memory footprint and increase the amount of cacheable data.

Finally, if someone is brave enough to do so, QSATS could be implemented onto FPGAs to accelerate this application. Use of integers instead of floating-point numbers would lend itself nicely to an FPGA implementation. This would allow for an extremely low powered, potentially high performance implementation to be used on reconfigurable high performance computing systems such as Maxwell [113].

7.2 Conclusions

High performance computing architectures are ever growing and adapting to the computational demands of scientific applications. Traditional multiprocessors are progressively improving architectural features (e.g., larger, faster caches and wider vector units) to provide support to computational algorithms while still providing the convenience of familiar programming capabilities. Linking these multiprocessors together with fast interconnects give the ability to scale applications to hundreds of thousands of cores. The Intel MIC architecture provides the developer with an architecture with many general purpose cores containing wide vector units. While not released yet, it promises to deliver high performance with little to no code change.

Graphics processing units (GPUs) are another architecture that has exploded into the high performance computing realm within recent years. With hundreds of cores, these devices give developers the ability to exploit high parallelism in many applications. GPUs are rapidly being adopted into supercomputer across the world due their high performance per watt/cost ratios.

This dissertation develops a multiprocessor, multinode implementation using C, vector units, OpenMP, pthreads, and MPI. This work starts by optimizing the serial C implementation as it is the backbone of all of the other implementations. This involved memory layout and algorithmic optimizations that led to an implementation that was 1.76x faster than the original Fortran implementation. Next, exploiting the use of vector units and C intrinsics, QSATS is able to extract an additional 1.35x using SSE and the Intel icc compilers on the Kraken supercomputer and 2.51x using AVX and the Intel icc compiler on the Intel Sandy Bridge architecture. The threaded implementations using pthreads and OpenMP do not provide the type of performance increase that is desired to fully utilize all resources. The efficiency of these implementations quickly dropped as the number of cores were increased. This is due to the large amount of data transfers needed for each thread interfering with that of another thread. However, the MPI implementation provides an efficient and highly scalable approach to parallelizing the QSATS program. Scaling to thousands of cores with approximately 80-90% efficiency, it is possible to use this implementation to tackle real science questions such as ones that involve thousands of atoms and hundreds of thousands of replicas.

The Intel MIC architecture is definitely an up and coming platform that holds a lot of promise. While not currently appropriate for this application, it is still an architecture that should be considered when developing new applications. The lack of performance in this application can be attributed to the early stages of the prototype board that was used in this work. It is possible that a final, full release board will be more polished with more advanced features than the current board. Therefore, it would be advantageous to explore this device more in the future.

GPUs hold the most promise as using a single NVIDIA Tesla M2090 equates to the same performance as three nodes of Kraken. GPUs are readily available in many supercomputers and consumer machines. This architecture is relatively straightforward to program due to the development of specialized parallel languages such as CUDA and OpenCL. Optimizing and debugging code for the GPU is much easier using CUDA than OpenCL since NVIDIA provides numerous tools to aid in this process for CUDA. In this work, the CUDA implementation provides approximately a 3.3x speedup over 12 cores of the AMD Opteron Istanbul. This translates a higher performance per watt and performance per cost since significantly more work can be accomplished with this architecture with less resources. For the QSATS program, GPUs are the best architecture for the job if looking to simulate hundreds of thousands of replicas

In addition to these implementations, I have also developed performance models for the multiprocessor, multinode scalability, and GPU architectures. These models provide insight into each of the devices, showing bottlenecks in the implementations allowing for performance enhancements. With these models, there is no guessing as to what portions of the code comprise the total runtime of the program. The scalability model is able to predict within less than 10% the resources needed to execute a VPI simulation so that we can minimize cost, which in this case is compute hours. With the GPU model, we can predict the overall runtime of the kernel code within less than 5% for large problem sizes. These models also provide insight into architectural features that both benefit (e.g. pipelines) and hinder performance (e.g. texture processing clusters in the Telsa GPU architecture). Developing the models as shown in this work contributes to the performance modeling field by providing an approach to create application specific models. That is, this work shows that it is possible to develop performance models by mostly reading the source code and counting arithmetic instructions or memory operations.

From these results, emerging architectures such as multicore, many core, and GPUs are the wave of the future for supercomputing technologies. This is

apparent in the growing adoption of GPUs (11% of the Top 500 in June 2012). The implementations and performance models in this dissertation will provide an approach to explore new and upcoming architectural features as they relate to the computational science realm.

Bibliography

Bibliography

[1] MPSAC Working Group, "Computational Science," National Science Foundation, Tech. Rep., 2010. 1

[2] G. Jones, D. Levy, L. Williams, C. Rockett, W. Wade, and A. Bardoe, "Petascale Science Delivered," Oak Ridge Leadership Computing Facility, Tech. Rep., 2009. [Online]. Available: http://www.olcf.ornl.gov/wp-content/uploads/2010/03/OLCFAR2009.pdf 1, 119

[3] S. Asano, M. Bolliger, M. Day, H. Hofstee, C. Johns, J. Kahle, A. Kameyama, J. Keaty, Y. Masubuchi, M. Riley, D. Shippy, D. Stasiak, M. Suzuoki, M. Wang, J. Warnock, S. Weitzel, D. Wendel, T. Yamazaki, and K. Yazawa, "The design and implementation of a first-generation CELL processor," in *Solid-State Circuits Conference*, Austin, TX, 2005, pp. 184–185. [Online]. Available: http://en.wikipedia.org/wiki/Cell_(microprocessor) 2

[4] J. Kurzak, A. Buttari, P. Luszczek, and J. Dongarra, "The PlayStation 3 for High Performance Scientific Computing," University of Tennessee, Knoxville, TN, Tech. Rep. February, 2008. [Online]. Available: http://eprints.ma.man.ac.uk/1022/ 2

[5] T. Elgar, "Intel Many Integrated Core Architecture," Intel Corporation, Tech. Rep. December, 2010. [Online]. Available: http://www.many-core.group.cam.ac.uk/ukgpucc2/talks/Elgar.pdf 2, 14

[6] J. S. Vetter, R. Glassbrook, J. Dongarra, K. Schwan, B. Loftis, S. Mcnally, J. Meredith, J. Rogers, P. Roth, K. Spafford, and S. Yalamanchili, "Keeneland: Bringing Heterogeneous GPU Computing to the Computational Science Community," *Computing in Science and Engineering*, vol. 13, no. 5, pp. 90–95, 2011. 2, 14, 45, 47

[7] J. Wells, T. Schulthess, and R. Pennington, "Scientists Assess a New Supercomputing Approach and Judge It a Win," in *Accelerating Computational Science Symposium*, Washington, D.C., 2012, pp. 1–12. 2, 45

[8] X. J. Yang, X. K. Liao, K. Lu, Q. F. Hu, J. Q. Song, and J. S. Su, "The TianHe-1A Supercomputer: Its Hardware and Software," *Journal of Computer Science and Technology*, vol. 26, no. 3, pp. 344–351, 2011. 2, 45

[9] M. P. Nightingale and C. J. Umrigar, *Quantum Monte Carlo Methods in Physics and Chemistry*. Springer, 1998. [Online]. Available: http://www.amazon.com/Quantum-Methods-Physics-Chemistry-Science/dp/0792355520 2

[10] D. M. Ceperley, "Path Integrals in the Theory of Condensed Helium," *Reviews of Modern Physics*, vol. 67, no. 2, p. 279, 1995. 2, 8

[11] A. Sarsa, K. E. Schmidt, and W. R. Magro, "A Path Integral Ground State Method," *The Journal of Chemical Physics*, vol. 113, no. 4, p. 1366, 2000. 2, 7, 8

[12] R. J. Hinde, "QSATS: MPI-Driven Quantum Simulations of Atomic Solids at Zero Temperature," *Computer Physics Communications*, vol. 182, no. 11, pp. 2339–2349, May 2011. 3, 8, 11, 17, 20, 37, 55, 93

[13] N. Metropolis and S. Ulam, "The Monte Carlo Method," *Journal of the American Statistical Association*, vol. 44, no. 247, pp. 335–341, 1949. 6, 8

[14] P. L'Ecuyer, "Good Parameters and Implementations for Combined Multiple Recursive Random Number Generators," *Operations Research*, vol. 47, no. 1, pp. 159–164, Feb. 1999. 6, 18

[15] M. Mascagni and A. Srinivasan, "Algorithm 806: SPRNG: a Scalable Library for Pseudorandom Number Generation," *ACM Transactions on Mathematical Software*, vol. 26, no. 3, pp. 436–461, Sep. 2000. 6, 18

[16] P. Colella, "Defining Software Requirements for Scientific Computing," DARPA HPCS presentation, Tech. Rep., 2004. 7

[17] K. Asanovic, R. Bodik, J. Demmel, T. Keaveny, K. Keutzer, J. Kubiatowicz, N. Morgan, D. Patterson, K. Sen, J. Wawrzynek, D. Wessel, and K. Yelick, "A View of the Parallel Computing Landscape," *Communications of the ACM*, vol. 52, no. 10, pp. 56–67, 2009. 7

[18] N. Cerf and O. Martin, "Finite Population-Size Effects in Projection Monte Carlo Methods," *Physical Review E*, vol. 51, no. 4, p. 3679, 1995. 8

[19] N. Metropolis, A. W. Rosenbluth, M. N. Rosenbluth, A. H. Teller, and E. Teller, "Equation of State Calculations by Fast Computing Machines," *The Journal of Chemical Physics*, vol. 21, no. 6, p. 1087, 1953. 8

[20] G. Box and M. Muller, "A Note on the Generation of Random Normal Deviates," *The Annals of Mathematical Statistics*, vol. 29, no. 2, pp. 610–611, 1958. 8, 23

[21] R. A. Aziz, F. R. McCourt, and C. C. Wong, "A New Determination of the Ground State Interatomic Potential for He 2," *Molecular Physics*, vol. 61, no. 6, pp. 1487–1511, Aug. 1987. 9, 54

[22] J. E. Cuervo, P.-N. Roy, and M. Boninsegni, "Path Integral Ground State with a Fourth-Order Propagator: Application to Condensed Helium." *The Journal of Chemical Physics*, vol. 122, no. 11, p. 114504, Mar. 2005. 11

[23] D. Galli, M. Rossi, and L. Reatto, "Bose-Einstein Condensation in Solid 4He," *Physical Review B*, vol. 71, no. 14, pp. 1–4, Apr. 2005. 11

[24] S. Mudhasani, "GPU-based Implementation of the Variational Path Integral Method," Master's Thesis, Electrical Engineering and Computer Science, University of Tennessee, 2011. 11, 17

[25] P. Kakani, "Implementation of Variational Path Integral Simulations on GPUs," Electrical Engineering and Computer Science, University of Tenessee, Knoxville, TN, Tech. Rep., 2009. 11, 17

[26] M. Hagog, "Looking for 4x speedups ? SSE to the rescue !" Intel Microprocessor Technology Labs, Tech. Rep., 2007. [Online]. Available: http://software.intel.com/file/1000 12

[27] P. Mehrotra, "How To Optimize Your Software For The Upcoming Intel Advanced Vector Extensions (Intel AVX)," p. 52, 2010. [Online]. Available: http://software.intel.com/file/32266 12

[28] "IEEE Std 1003.1, 2004 Edition. The Open Group Technical Standard. Base Specifications, Issue 6. Includes IEEE Std 1003.1-2001, IEEE Std 1003.1-2001/Cor 1-2002 and IEEE Std 1003.1-2001/Cor 2-2004. Shell and Utilities," *In IEEE Std 1003.1, 2004 Edition. The Open Group Technical Standard. Base Specifications*, no. 6, p. 616, 1994. 12

[29] L. Dagum and R. Menon, "OpenMP: An Industry Standard API for Shared-Memory Programming," *IEEE Computational Science and Engineering*, vol. 5, no. 1, pp. 46–55, 1998. 12

[30] J. E. Stone, D. Gohara, and G. Shi, "OpenCL: A Parallel Programming Standard for Heterogeneous Computing Systems," *Computing in Science & Engineering*, vol. 12, no. 3, pp. 66–73, 2010. 12, 13, 48, 49

[31] M. Snir, S. W. Otto, S. Huss-Lederman, D. W. Walker, and J. Dongarra, *MPI: The Complete Reference*, 2nd ed., Janusz S. Kowalik, Ed. Cambridge: MIT Press, 1998. [Online]. Available: http://www.mcs.anl.gov/mpi/ 12, 37

[32] M. Gordon and M. Schmidt, "Advances in Electronic Structure Theory: GAMESS a Decade Later," in *Theory and Applications of Computational Chemistry: The First Forty Years*. Amsterdam: Elsevier Science, 2005, pp. 1167 – 1189. 12

[33] M. Valiev, E. Bylaska, N. Govind, K. Kowalski, T. Straatsma, H. Van Dam, D. Wang, J. Nieplocha, E. Apra, and T. Windus, "NWChem: A Comprehensive and Scalable Open-Source Solution for Large Scale Molecular Simulations," *Computer Physics Communications*, vol. 181, no. 9, pp. 1477–1489, Sep. 2010. 12, 14

[34] S. F. Altschul, W. Gish, W. Miller, E. W. Myers, and D. J. Lipman, "Basic local alignment search tool," *Journal of Molecular Biology*, vol. 215, no. 3, pp. 403–410, 1990. 12

[35] S. R. Eddy, "Accelerated Profile HMM Searches." *PLoS Computational Biology*, vol. 7, no. 10, p. e1002195, Oct. 2011. 12

[36] J. C. Phillips, R. Braun, W. Wang, J. Gumbart, E. Tajkhorshid, E. Villa, C. Chipot, R. D. Skeel, L. Kalé, and K. Schulten, "Scalable molecular dynamics with NAMD." *Journal of Computational Chemistry*, vol. 26, no. 16, pp. 1781–802, Dec. 2005. 12

[37] M. Harris, "General-Purpose Computation on Graphics Hardware." [Online]. Available: gpgpu.org 13

[38] Nvidia, "NVIDIA CUDA C Programming Guide," *NVIDIA Corporation*, 2012. 13, 45, 47, 48, 49, 50, 52, 53, 131, 136, 140

[39] "Brook+ SC07 BOF Session," in *Supercomputing 2007*. ACM, 2007. [Online]. Available: http://developer.amd.com/gpu_assets/AMD-Brookplus.pdf 13

[40] J. Phillips, J. Stone, and K. Schulten, "Adapting a Message-Driven Parallel Application to GPU-Accelerated Clusters," in *Supercomputing 2008*. ACM, 2008, pp. 1–9. 14

[41] D. D. Jenkins and G. D. Peterson, "AESS : Accelerated Exact Stochastic Simulation," *Computer Physics Communications*, vol. 182, no. 12, pp. 2580–2586, 2011. 14, 17

[42] Y. Liu, B. Schmidt, and D. L. Maskell, "CUDASW++2.0: Enhanced Smith-Waterman Protein Database Search on CUDA-Enabled GPUs Based on SIMT and Virtualized SIMD Abstractions," *BMC Research Notes*, vol. 3, no. 1, p. 93, Jan. 2010. 14

[43] J. Walters, V. Balu, S. Kompalli, and V. Chaudhary, "Evaluating the Use of GPUs in Liver Image Segmentation and HMMER Database Searches," in *IEEE International Symposium on Parallel & Distributed Processing*. IEEE, 2009, pp. 1–12. 14

[44] J. E. Stone, J. C. Phillips, P. L. Freddolino, D. J. Hardy, L. G. Trabuco, and K. Schulten, "Accelerating Molecular Modeling Applications with Graphics Processors." *Journal of Computational Chemistry*, vol. 28, no. 16, pp. 2618–40, Dec. 2007. 14

[45] R. Weber, A. Gothandaraman, R. J. Hinde, and G. D. Peterson, "Comparing Hardware Accelerators in Scientific Applications: A Case Study," *IEEE Transactions on Parallel and Distributed Systems*, vol. 6, no. 1, pp. 1–10, 2010. 14, 17

[46] V. Volkov and J. Demmel, "Benchmarking GPUs to Tune Dense Linear Algebra," in *High Performance Computing, Networking, Storage and Analysis,*

2008. SC 2008. International Conference for, no. November. IEEE, 2008, pp. 1–11. 14, 16, 47, 99

[47] S. Tomov, J. Dongarra, and M. Baboulin, "Towards Dense Linear Algebra for Hybrid GPU Accelerated Manycore Systems," *Parallel Computing*, vol. 36, no. 5-6, pp. 232–240, Jun. 2010. 14

[48] E. Agullo, J. Demmel, J. Dongarra, B. Hadri, J. Kurzak, J. Langou, H. Ltaief, P. Luszczek, and S. Tomov, "Numerical Linear Algebra on Emerging Architectures: The PLASMA and MAGMA Projects," *Journal of Physics: Conference Series*, vol. 180, p. 012037, Jul. 2009. 14

[49] D. B. Kirk, "NVIDIA CUDA Software and GPU Parallel Computing Architecture," in *International Symposium on Memory Management*. Montreal: ACM, 2007. [Online]. Available: http://www.nvidia.co.kr/content/cudazone/download/showcase/kr/Tutorial-DKIRK.pdf 14

[50] H. Meuer, E. Strohmaier, J. Dongarra, and H. Simon, "Top500 November 2010," 2010. [Online]. Available: http://top500.org/lists/2010/11 14

[51] S. Barak, "Intel unveils 1 TFLOP/s Knight's Corner." [Online]. Available: http://www.eetimes.com/electronics-news/4230654/Intel-unveils-1-TFLOP-s-Knight-s-Corner 14

[52] N. Satish, C. Kim, J. Chhugani, A. D. Nguyen, V. W. Lee, D. Kim, and P. Dubey, "Fast Sort on CPUs, GPUs and Intel MIC Architectures," in *ACM Special Interest Group on Management of Data*, New York City, 2010, pp. 351–362. [Online]. Available: http://en.wikipedia.org/wiki/Intel_MIC 14, 43

[53] "Intel Many Integrated Core Architecture," 2011. [Online]. Available: http://www.intel.com/content/www/us/en/architecture-and-technology/many-integrated-core/intel-many-integrated-core-architecture.html 14

[54] Intel, "Intel Many Integrated Core (Intel MIC) Architecture ISC11 Demos and Performance Description," Intel, Tech. Rep., 2011. [Online]. Available: http://newsroom.intel.com/servlet/JiveServlet/download/2152-4-5220/ISC_Intel_MIC_factsheet.pdf 14

[55] L. Hu and I. Gorton, "Performance Evaluation for Parallel Systems: A Survey," UNSW-CSE-TR-9707, School of Computer Science and Engineering, University of New South Wales, Sydney, Australia, Tech. Rep. October, 1997. [Online]. Available: http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.12.9079&rep=rep1&type=pdf 14

[56] K. Kant, *Introduction To Computer System Performance Evaluation*. Mcgraw-Hill College, 1992. 15

[57] M. C. Smith, "Analytical Modeling of High Performance Reconfigurable Computers : Prediction and Analysis of System Performance," Ph.D. dissertation, Electrical and Computer Engineering, University of Tennessee, 2002. 15

[58] G. D. Peterson, "Parallel Application Performance on Shared, Heterogeneous Workstations," Ph.D. dissertation, Department of Computer Science and Engineering, Washington University, Dec. 1994. 16

[59] M. Smith and G. Peterson, "Parallel Application Performance on Shared High Performance Reconfigurable Computing Resources," *Performance Evaluation*, vol. 60, no. 1-4, pp. 107–125, May 2005. 16

[60] S. Hong and H. Kim, "An Integrated GPU Power and Performance Model," *Proceedings of the 37th Annual International Symposium on Computer Architecture - ISCA '10*, p. 280, 2010. 16

[61] J. Sim, A. Dasgupta, and H. Kim, "A Performance Analysis Framework for Identifying Potential Benefits in GPGPU Applications," *Principles and Practice of Parallel Programming*, p. 11, 2012. 16

[62] A. Gothandaraman, "Accelerating Quantum Monte Carlo Simulations with Emerging Architectures," Ph.D. dissertation, Electrical and Computer Engineering, University of Tennessee, 2009. 16, 17

[63] E. Ipek, B. de Supinski, M. Schulz, S. McKee, J. Cunha, and P. Medeiros, "An Approach to Performance Prediction for Parallel Applications," *Euro-Par 2005 Parallel Processing*, vol. 3648, pp. 627–628, 2005. 16

[64] S. Baghsorkhi, I. Gelado, and M. Delahaye, "Efficient Performance Evaluation of Memory Hierarchy for Highly Multithreaded Graphics Processors," in *Principles and Practice of Parallel Programming*. New York, New York, USA: ACM Press, 2012, p. 23. 16

[65] E. Abdikamalov, A. Burrows, and C. Ott, "A New Monte Carlo Method for Time-Dependent Neutrino Radiation Transport," *Astrophysics Journal*, vol. Submitted, 2012. 16

[66] P. D. Moral and A. Doucet, "An Adaptive Sequential Monte Carlo Method for Approximate Bayesian Computation," *Statistics and Computing*, no. 1, Aug. 2011. 17

[67] D. Gillespie, "A general method for numerically simulating the stochastic time evolution of coupled chemical reactions," *Journal of Computational Physics*, vol. 22, no. 4, pp. 403–434, Dec. 1976. 17

[68] R. J. Needs, M. D. Towler, N. D. Drummond, and P. López Ríos, "Continuum variational and diffusion quantum Monte Carlo calculations." *Journal of Physics. Condensed Matter : an Institute of Physics Journal*, vol. 22, no. 2, p. 023201, Jan. 2010. 17

[69] H. Li and L. Petzold, "Efficient Parallelization of the Stochastic Simulation Algorithm for Chemically Reacting Systems On the Graphics Processing Unit," *International Journal of High Performance Computing Applications*, vol. 24, no. 2, pp. 107–116, Jun. 2009. 17

[70] D. D. Jenkins and G. D. Peterson, "GPU Accelerated Stochastic Simulation," in *Symposium on Application Accelerators in High-Performance Computing*, Knoxville, TN, 2010. 17

[71] D. D. Jenkins, "Accelerating the Stochastic Simulation Algorithm Using Emerging Architectures," Master's Thesis, Electrical Engineering and Computer Science, University of Tennessee at Knoxville, 2009. 17

[72] F. Heymann and R. Siebenmorgen, "Gpu-Based Monte Carlo Dust Radiative Transfer Scheme Applied To Active Galactic Nuclei," *The Astrophysical Journal*, vol. 751, no. 1, p. 27, May 2012. 17

[73] A. G. Anderson, W. a. Goddard, and P. Schröder, "Quantum Monte Carlo on graphical processing units," *Computer Physics Communications*, vol. 177, no. 3, pp. 298–306, Aug. 2007. 17

[74] M. Gokhale, J. Frigo, C. Ahrens, and J. Tripp, "Monte Carlo radiative heat transfer simulation on a reconfigurable computer," *Field Programmable Logic*, pp. 95–104, 2004. 17

[75] G. Zhang, P. Leong, and C. Ho, "Reconfigurable Acceleration for Monte Carlo Based Financial Simulation," in *International Conference on Field Programmable Techology*, 2005, pp. 215–222. 17

[76] G. C. T. Chow, A. H. T. Tse, Q. Jin, W. Luk, P. H. Leong, and D. B. Thomas, "A Mixed Precision Monte Carlo Methodology for Reconfigurable Accelerator Systems," in *Proceedings of the ACM/SIGDA International Symposium on*

Field Programmable Gate Arrays - FPGA '12. New York, New York, USA: ACM Press, 2012, p. 57. 17

[77] J. Lee, Y. Bi, G. D. Peterson, R. J. Hinde, and R. J. Harrison, "HASPRNG: Hardware Accelerated Scalable Parallel Random Number Generators," *Computer Physics Communications*, vol. 180, no. 12, pp. 2574–2581, Dec. 2009. 18

[78] S. Gao and G. D. Peterson, "GPU Accelerated Scalable Parallel Random Number Generators," in *Symposium on Application Accelerators in High-Performance Computing*, vol. In Review, 2010. 18, 50

[79] NVIDIA, "CUDA Toolkit 4.2 CURAND Guide," NVIDIA, Santa Clara, CA, Tech. Rep., 2012. [Online]. Available: http://developer.download.nvidia.com/compute/DevZone/docs/html/CUDALibraries/doc/CURAND_Library.pdf 18, 50

[80] P. L'Ecuyer, R. Simard, E. J. Chen, and W. D. Kelton, "An Object-Oriented Random-Number Package with Many Long Streams and Substreams," *Operations Research*, vol. 50, no. 6, pp. 1073–1075, Nov. 2002. 22

[81] A. Munshi, "OpenCL specification v1.1," 2009. [Online]. Available: http://www.khronos.org/registry/cl/specs/opencl-1.1.pdf 37, 57

[82] P. Schwan, "Lustre: Building a file system for 1000-node clusters," in *Proceedings of the Linux Symposium*, 2003. 43

[83] H. You, Q. Liu, Z. Li, and S. Moore, "The Design of an Auto-Tuning I/O Framework on Cray XT5 System," in *Cray User Group Meeting*, Oak Ridge, TN, 2011, p. 10. 43, 123

[84] E. Lindholm, J. Nickolls, and S. Oberman, "NVIDIA Tesla: A Unified Graphics and Computing Architecture," *IEEE Micro*, vol. 28, no. 2, pp. 39–55, 2008. 46

[85] NVIDIA, "Board Specification: Tesla C1060 Computing Processor Board," Nvidia Corporation, Tech. Rep. September, 2008. 46, 98

[86] ——, "GeForce GTX 480 Specifications," 2010. [Online]. Available: http://www.geforce.com/hardware/desktop-gpus/geforce-gtx-480/specifications 47

[87] "Fermi Architecture," 2011. [Online]. Available: http://www.nvidia.com/object/fermi_architecture.html 47

[88] Nvidia, "Whitepaper: NVIDIAs Next Generation CUDA Compute Architecture: Fermi," NVIDIA Corporation, Tech. Rep., 2009. 47

[89] J. Nickolls, I. Buck, and M. Garland, "Scalable Parallel Programming with CUDA," *ACM Queue*, vol. 6, no. 2, 2008. 48

[90] NVIDIA, "CUDA GPUs," 2012. [Online]. Available: http://developer.nvidia.com/cuda-gpus 48

[91] D. D. Jenkins, "CUDA/OpenCL Hardware Interpolation Example." [Online]. Available: https://github.com/daviddjenkins/CUDA-OpenCL-Hardware-Interpolation 55, 58

[92] NVIDIA, "Getting Started with CUDA SDK Samples," Tech. Rep. January, 2012. 57, 136, 140

[93] S. Browne, J. Dongarra, N. Garner, G. Ho, and P. Mucci, "A Portable Programming Interface for Performance Evaluation on Modern Processors," *International Journal of High Performance Computing Applications*, vol. 14, no. 3, pp. 189–204, 2000. 62, 112

[94] R. Weber and G. D. Peterson, "Improved OpenCL Programmability with clUtil," in *Supercomputing*. Salt Lake City, UT: ACM, 2012. [Online]. Available: http://code.google.com/p/clutil/ 79

[95] A. F. Szczepanksi, S. Ahern, and M. R. Fahey, "A Tale of Two Systems Flexibility of Usage of Kraken and Nautilus at the National Institute for," in *Proceedings of the 1st Conference of the Extreme Science and Engineering Discovery Environment: Bridging from the eXtreme to the Campus and Beyond*. Chicago: ACM, 2012, pp. 6:1–6:8. 79, 119

[96] E. Christophe, J. Michel, and J. Inglada, "Remote Sensing Processing: From Multicore to GPU," *IEEE Journal of Selected Topics in Applied Earth Observations and Remote Sensing*, vol. 4, no. 3, pp. 643–652, Sep. 2011. 86

[97] S. McConnell, R. Sturgeon, G. Henry, A. Mayne, and R. Hurley, "Scalability of Self-organizing Maps on a GPU cluster using OpenCL and CUDA," *Journal of Physics: Conference Series*, vol. 341, p. 012018, Feb. 2012. 86

[98] NVIDIA, "Tesla M2090 Dual-Slot Computing Processor," p. 13, 2011. [Online]. Available: http://www.nvidia.com/docs/io/43395/tesla-m2090-board-specification.pdf 95, 98

[99] AMD, "AMD Family 10h Server and Workstation Processor Power and Thermal Data Sheet," Tech. Rep., 2010. 95, 98

[100] CPUWorld, "Intel Xeon E5-2680," 2012. [Online]. Available: http://www.cpu-world.com/CPUs/Xeon/Intel-XeonE5-2680.html 98

[101] "Comparison of Nvidia graphics processing units." [Online]. Available: http://en.wikipedia.org/wiki/Comparison_of_Nvidia_graphics_processing_units 98

[102] "Comparison of AMD graphics processing units," 2012. [Online]. Available: http://en.wikipedia.org/wiki/Comparison_of_AMD_graphics_processing_units 98

[103] H. Meuer, E. Strohmaier, J. Dongarra, and H. Simon, "Top500.org," 2012. [Online]. Available: http://www.top500.org 99

[104] A. Rane, J. Browne, and L. Koesterke, "A systematic process for efficient execution on Intel's heterogeneous computation nodes," *Conference of the Extreme Science and Engineering Discovery Environment on Bridging from the eXtreme to the Campus and Beyond - XSEDE '12*, no. 8, p. 1, 2012. 100

[105] J. Edler and M. D. Hill, "Dinero IV Trace-Driven Uniprocessor Cache Simulator," 2007. [Online]. Available: http://pages.cs.wisc.edu/~markhill/DineroIV 110

[106] Intel, "Intel 64 and IA-32 Architectures Optimization Reference Manual," p. 800, 2012. [Online]. Available: http://www.intel.com/content/dam/doc/manual/64-ia-32-architectures-optimization-manual.pdf 111

[107] "Software Optimization Guide for AMD Family 10h and 12h Processors," Advanced Micro Devices, Tech. Rep. February, 2011. [Online]. Available: http://support.amd.com/us/Processor_TechDocs/40546.pdf 111

[108] "Software Optimization Guide for AMD Family 15h Processors," Advanced Micro Devices, Tech. Rep., 2012. [Online]. Available: http://support.amd.com/us/Processor_TechDocs/47414_15h_sw_opt_guide.pdf 113, 114

[109] W. D. Norcott, "IOZone Filesystem Benchmark," 2006. [Online]. Available: http://www.iozone.org/ 122, 123

[110] "Kraken Specification," 2012. [Online]. Available: http://icl.cs.utk.edu/hpcc/hpcc_results.cgi?display=combo 123

[111] H. Wong and M. Papadopoulou, "Demystifying GPU microarchitecture through microbenchmarking," in *International Symposium on Performance Analysis of Systems and Software*. Ieee, Mar. 2010, pp. 235–246. 131, 136, 140

[112] R. J. Hinde, "Variational path integral simulations using discretized coordinates," *Chemical Physics Letters*, vol. 418, no. 4-6, pp. 481–484, Feb. 2006. 146

[113] R. Baxter, S. Booth, M. Bull, G. Cawood, J. Perry, M. Parsons, A. Simpson, A. Trew, A. McCormich, G. Smart, R. Smart, A. Cantle, R. Chamberlain, and G. Genest, "Maxwell-a 64 FPGA supercomputer," *Second NASA/ESA Conference on Adaptive Hardware and Systems, 2007*, 2007. 146

Vita

David Jenkins joined the Department of Electrical Engineering and Computer Science at the University of Tennessee, Knoxville as an undergraduate student in August 2001. He received his B.S. degree in Computer Engineering and his M.S. degree in Computer Engineering from University of Tennessee, Knoxville. He has been part of the Tennessee Advanced Computing Laboratory since August 2008. His current research focuses on exploring the viability of emerging architectures for accelerating computational science applications. He was a recipient of the NSF funded IGERT SCALE-IT Fellowship in 2010.