8-2009

Accelerating Quantum Monte Carlo Simulations with Emerging Architectures

Akila Gothandaraman
University of Tennessee - Knoxville

To the Graduate Council:

I am submitting herewith a dissertation written by Akila Gothandaraman entitled "Accelerating Quantum Monte Carlo Simulations with 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:

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

Accepted for the Council:

Carolyn R. Hodges

Vice Provost and Dean of the Graduate School

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

To the Graduate Council:

I am submitting herewith a dissertation written by Akila Gothandaraman entitled “Accelerating Quantum Monte Carlo Simulations with 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:

Itamar Arel

Robert J. Harrison

Robert J. Hinde

Xiaorui Wang

Accepted for the Council:

Carolyn R. Hodges

Vice Provost and Dean of the Graduate School

Accelerating Quantum Monte Carlo Simulations with Emerging Architectures

A Dissertation
Presented for the
Doctor of Philosophy
Degree
The University of Tennessee, Knoxville

Akila Gothandaraman
August 2009

ACKNOWLEDGEMENTS

I would like to thank God for giving me the strength and the opportunity to work with an excellent team of professors, and colleagues as well as blessing me with a wonderful family, and friends. My heartfelt thanks, admiration, and appreciation goes to my advisor, Dr. Gregory Peterson, who has been instrumental towards my professional development. I am grateful to him for always believing in my abilities and for constantly motivating me to reach the highest standards in my research. I would also like to thank other members of my doctoral committee, Dr. Itamar Arel, Dr. Robert Harrison, Dr. Robert Hinde, and Dr. Xiaorui Wang for their constructive feedback and comments that have greatly improved this dissertation. As a team member of the National Science Foundation (NSF) funded Computational Chemistry project, I got the opportunity to interact with Dr. Harrison and Dr. Hinde on a number of occasions about chemistry related topics that have been invaluable in this work. I would also like to thank Dr. Hinde for not only allowing us to be a part of his graduate chemistry course at UT, but also tailoring it to make it most interesting to benefit non-chemistry students. He is a wonderful teacher and I regret for not having had the opportunity to attend more of his classes. I would also like to extend my thanks and gratitude to my graduate advisor, Dr. Syed Islam, for his encouragement and support, right from the day I started my graduate education at the University of Tennessee to this date. My initial research discussions with Dr. Lee Warren, a former post-doctoral researcher in this project, were invaluable towards the successful implementation of my research ideas and I gratefully acknowledge his help and constant feedback. I would also like to thank Mr. Rick Arthur, and Mr. Steve Zingelewicz, who supervised me during my internship in the Advanced Computing Laboratory at GE Global Research. My interactions with them greatly helped strengthen my graphics programming and software engineering skills.

Over the last few years, I have had the opportunity to work with a smart group of colleagues at the Tennessee Advanced Computing Laboratory (TACL) - JunKyu Lee, Junquing Sun, Rick Weber, Depeng Yang, Yu Bi, Saumil Merchant, and Bhanu Rekapalli. Our research-related discussions promoted mutual learning and conversations such as IPhone game development, video games, politics, to name a few, served as pleasant distractions from our research.

I have been very fortunate to have the love and unconditional support of my parents, Mrs. Rani and Mr. Gothandaraman. I am also grateful to my grandparents, and my in-laws for their love and prayers. Last but not the least, my love and thanks to my dearest husband, Balajee, for his love, patience, understanding, and support at every stage of this work.

This work was supported by the National Science Foundation grant NSF CHE-0625598 and we gratefully acknowledge prior support from the University of Tennessee Science Alliance.

ABSTRACT

Scientific computing applications demand ever-increasing performance while traditional microprocessor architectures face limits. Recent technological advances have led to a number of emerging computing platforms that provide one or more of the following over their predecessors: increased energy efficiency, programmability/flexibility, different granularities of parallelism, and higher numerical precision support. This dissertation explores emerging platforms such as reconfigurable computing using field-programmable gate arrays (FPGAs), and graphics processing units (GPUs) for quantum Monte Carlo (QMC), a simulation method widely used in physics and physical chemistry. This dissertation makes the following significant contributions to computational science. First, we develop an open-source user-friendly hardware-accelerated simulation framework using reconfigurable computing. This framework demonstrates a significant performance improvement over the optimized software implementation on the Cray XD1 high performance reconfigurable computing (HPRC) platform. We use novel techniques to approximate the kernel functions, pipelining strategies, and a customized fixed-point representation that guarantees the accuracy required for our simulation. Second, we exploit the enormous amount of data parallelism on GPUs to accelerate the computationally intensive functions of the QMC application using NVIDIA's Compute Unified Device Architecture (CUDA) paradigm. We experiment with single-, double- and mixed- precisions for the CUDA implementation. Finally, we present analytical performance models to help validate, predict, and characterize the application performance on these architectures. Together, this work that combines novel algorithms and emerging architectures, along with the performance models, will serve as a starting point for investigating related scientific applications on present and future heterogeneous architectures.

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

1. INTRODUCTION

1.1. Motivation

The Cornell Theory Center defines Computational Science as "a field that concentrates on the effective use of computer software, hardware, and mathematics to solve real problems. It is a term used when it is desirable to distinguish the more pragmatic aspects of computing from (1) computer science, which often deals with the more theoretical aspects of computing; and from (2) computing engineering, which deals primarily with the design and construction of computers themselves. Computational science is often thought of as the *third leg of science* along with experimental and theoretical science" [1]. A recent National Science Foundation report from a team of leading researchers, "*International Assessment of Simulation-Based Engineering and Science*," emphasizes the need for computer simulation and modeling to advance science and engineering research [2]. This report also underlines the need to exploit new computer architectures, and improve the capabilities to use them before they become ubiquitous [2].

Many applications within science and engineering are characterized by rising performance demands. Current high performance computing (HPC) systems have provided the computing power required by scientific applications. These systems, such as supercomputers or cluster-based systems, consist of a collection of processors or processing nodes connected over a suitable interconnect and work collectively to solve a scientific problem that typically involves large volumes of data and complex calculations. HPC systems from vendors like Cray and IBM facilitate large-scale simulations and high-end computations [3, 4]. The Cray XT5 codenamed "Jaguar" installed at Oak Ridge National Laboratory [5] is a 1.059 petaflop

(sustained performance) supercomputer [6] that aims to address challenging problems in areas such as climate modeling, materials science, fusion and combustion. A supercomputer from the Los Alamos National Laboratory, nicknamed “Roadrunner”, is a hybrid architecture combining AMD Opterons with IBM Cell processors, providing a sustained performance of 1.105 petaflops [4]. At the time of this writing, it is also the world’s fastest supercomputer [6]. This supercomputer is intended for nuclear materials, scientific, and financial simulations. A Cray XT5 supercomputer, called “Kraken” from University of Tennessee’s National Institute for Computational Science (NICS) [7], is now the world’s sixth fastest supercomputer with a sustained performance of 463.3 teraflops [6].

Architectures such as multi-core processors, reconfigurable computing using field-programmable gate arrays, graphics processors, and Cell processors have emerged as alternatives for scientific computing. Next generation supercomputers are likely to be made of one or more of these emerging technologies. Reconfigurable computing [8], graphics processing units (GPUs) [9, 10] and the Cell broadband engine (BE) [11] can provide a boost in performance and productivity for high performance computing. Graphics processors and the Cell processor, originally developed for the gaming market, have also been targeted to applications such as bioinformatics [12], the Smith-Waterman algorithm [13], and molecular modeling [14]. We provide an overview of these emerging architectures in the landscape of high performance computing.

Multi-core Processors: The performance gains in traditional microprocessors (single-core) from increasing the clock frequency have greatly diminished due to the memory wall and power wall limitations. Cache memories mitigated the effects of the memory wall or the widening gap between processor speed and memory latencies, while optimizations such as pipelining, superscalar, and out-of-order execution exploited the available instruction-level parallelism (ILP) [15] . The power wall and the increased heat dissipation make it more difficult to design processors with increased clock rates. The

increased design complexity, power dissipation and the demand for increased thread-level parallelism (TLP) led to a paradigm shift - the development of multi- and many-core processors, which replicate the processor cores with reduced clock rates, providing increased parallelism and performance in an energy efficient manner. Multi-core processors combine two or more cores in a single package. Multi-core technology provided by chipmakers like AMD and Intel is mainstream in today's desktops and high performance systems. A challenge while using these platforms is designing software applications to take advantage of the compute power from additional cores, taking into account the speed and memory-access capabilities of the cores.

Reconfigurable Computing (RC): RC is the combination of reconfigurable logic and a general-purpose microprocessor (GPP). Reconfigurable logic devices such as field-programmable gate arrays (FPGAs) contain a number of logic blocks that can be configured to implement the required logic functions by connecting them using programmable routing resources. The processor performs operations such as control and memory accesses that cannot be done efficiently using reconfigurable logic and the computational cores are mapped onto the FPGA [16]. FPGAs operate at much lower clock speeds (often 100-200 MHz) and do not have the same amount of heat dissipation problems faced by microprocessors (operating at GHz). The tasks of FPGA programming commonly done using hardware description languages and physical routing of the circuit are arduous, requiring a great deal of time and effort, but with careful design, optimization, and resource usage, FPGAs have the potential to yield excellent performance. In particular, applications using integer or logic operations such as digital signal processing, cryptography, and DNA sequencing are extremely suited for FPGA implementation. The advent of higher density FPGAs [17] with embedded multipliers has also made the much-needed floating-point arithmetic in scientific applications feasible on these devices [18]. FPGAs have been widely used in applications such as cryptography [19] and signal processing [20]. [21] provides a survey on reconfigurable

architectures and a discussion of their representative applications including signal and image processing, bioinformatics, and supercomputing.

High performance reconfigurable computing (HPRC) refers to the combination of traditional HPC systems with RC elements to provide increased performance and flexibility. HPRC systems consist of a number of computing nodes connected using some interconnection network with some or all nodes equipped with one or more reconfigurable logic elements. This allows users to exploit the *polygranular* parallelism by allowing users to exploit the fine-grained parallelism offered by reconfigurable computing in addition to the parallelism that is normally achievable using parallel computing.

Graphics Processing Units (GPUs): GPUs have evolved from fixed-function pipelines to programmable pipelines making them useful for applications other than 3D graphics rendering. The GPUs with their parallel single-instruction multiple-data (SIMD) processing units and tremendous on-chip memory bandwidth provide vast amounts of computational power for scientific applications. Along with the increase in performance, the ease of programmability with languages that provide a non-graphics API for general-purpose computing has made them an attractive platform for computationally intensive applications. [22] surveys a broad range of applications that have utilized graphics hardware for general-purpose computation.

Cell Processor: The Sony-Toshiba-IBM (STI) Cell Processor is a heterogeneous multi-core system originally developed for the Sony Playstation 3 game console. It consists of one 64-bit Power Processing Element (PPE), which is the control unit of the Cell Processor. The real processing power of the Cell comes from the eight Synergistic Processing Elements (SPEs) which are floating-point vector processors [11] and suited for data-intensive media or scientific applications.

Previous research has focused on accelerating scientific applications by outsourcing the computationally intensive kernels to accelerators like GPUs or FPGAs [14, 23], while the original application runs on the compute node. We can see from previous work that these emerging platforms used independently demonstrate promising results over the optimized CPU implementations for science and engineering applications [24, 25].

1.2. Problem

This dissertation explores emerging architectures such as field-programmable gate arrays and graphics processing units for accelerating a widely used simulation technique in physics and physical chemistry called the Quantum Monte Carlo method (QMC). This work investigates using emerging architectures to provide cost-effective simulation capabilities as well as constructing performance models to better understand how to best map the application to present and future computing platforms.

The QMC simulation method [26, 27], is compute bound. This method involves sampling a number of configurations of particles and averaging the properties over a large number of iterations. With increased computational resources, one can simulate a larger physical system for a longer time. Simulating for a longer time, i.e., repeating the simulation for a larger number of iterations, gives us good estimates of the properties. Simulating a larger physical system allows us to study the interactions among a large number of particles under different environments, assumptions etc., which is not feasible in actual experimental studies.

1.3. Approach

Quantum Monte Carlo simulation methods are used in this work to obtain the ground-state properties of a cluster of atoms. The application is implemented on FPGA and GPU platforms (as coprocessor

accelerators) by identifying computationally intensive kernels that are suitable for mapping onto these platforms, depending on the complexity of the kernels, required numerical precision, and resources available (e.g., number of processing elements/pipelines/cores, on-chip/device memory, logic resources). We also develop detailed analytical performance models to understand the best ways of mapping the QMC application onto these platforms and validate the model using empirical results. The results from targeting the individual computing platforms and the performance models will help us predict the performance on next-generation heterogeneous or hybrid computing platforms. Our definition of a hybrid computing platform is in alignment with that in [28] which denotes an architecturally diverse system, implying a disparate set of computing engines, for example, a dual-node system where node 1 consists of a processor (single or multi-core) with a plug-in FPGA device and node 2 consists of a processor (single or multi-core) with a plug-in GPU device. This dissertation will provide practical experience and mathematically-based modeling insight into the potential for deploying hybrid computing platforms for next-generation scientific computing applications.

1.4. Contributions

This dissertation explores the use of emerging architectures such as field-programmable gate arrays and graphics processing units to accelerate a Quantum Monte Carlo application. We develop novel algorithms and architectures to accelerate the computationally intensive kernels of the application. Along with the architectural implementations, we also develop analytical performance models for the CPU, FPGA, and GPU implementations.

This dissertation has the following contributions:

- This is the first work to explore reconfigurable computing architectures for Quantum Monte Carlo simulations. Our work provides a significant speedup over the CPU implementation.

- A novel parallel and pipelined FPGA architecture is developed that uses a fixed-point representation for all calculations, providing speed and area efficiency without any loss of accuracy.

- A general-purpose user-friendly generic simulation framework is developed for Quantum Monte Carlo simulations. This framework gives the capability to simulate a variety of atomic clusters, and also the capability to change the kernel functions depending on the atoms involved in the simulation.

- This is the first work to explore graphics processing units for Quantum Monte Carlo simulations to study the ground-state properties of atomic clusters. This work explores different numerical precisions (single-, double-, and a combination of single- and double- precision) to investigate the implementation that provides the best performance without any compromise in accuracy.

- This work develops accurate performance models for the QMC simulation on FPGAs and extends performance models in previous research work to explore new reconfigurable computing systems.

1.5. Outline of the Document

The rest of this dissertation is organized as the following chapters.

Chapter 2 provides an overview of the MD and QMC simulation methods. A survey of the hardware and software development environments for FPGAs and GPUs is presented. Next, we discuss the related research efforts that have utilized current HPC systems, special-purpose processors, FPGAs and GPUs for MC in general, and for MD and QMC simulations. Finally, we present related work in the area of performance evaluation of HPC and HPRC systems.

Chapter 3 outlines the CPU implementation of the QMC application. We discuss the serial and parallel software implementations of the QMC algorithm.

Chapter 4 presents the details of the pipelined reconfigurable architecture for QMC simulations. We discuss the rationale for our design choices for the various building blocks of the architecture. We also provide an overview of our target platform, the Cray XD1 high performance reconfigurable computer and discuss the implementation details of the hardware accelerated QMC application on the Cray XD1.

Chapter 5 presents the GPU implementation of the QMC simulation. We present our naïve implementation of the kernels of the QMC application on the target NVIDIA GPU using the Compute Unified Device Architecture (CUDA) programming environment. Following this, we discuss optimization strategies to further accelerate the QMC application. We also present an overview of the NVIDIA GPU architecture and the CUDA environment.

In Chapter 6, we compare and discuss the results obtained while targeting the QMC application on the CPU, reconfigurable computing, and GPU platforms. We also present the results from the parallel implementations on the CPU, FPGA, and GPU platforms.

Chapter 7 presents the details of the analytical performance models for the individual CPU, FPGA and GPU platforms.

In Chapter 8, we provide concluding remarks and directions for future research.

2. BACKGROUND AND RELATED WORK

In this chapter, we provide an overview of the two widely used simulation methods in chemistry, namely Molecular Dynamics and Quantum Monte Carlo. We discuss in detail the Quantum Monte Carlo method, which is the focus of this dissertation. We also present the details of the model chemical system that is used for our simulations. We discuss the state-of-the art FPGA and GPU platforms and the development environments available in these platforms. We will then survey previous work related to this dissertation, both in the area of architectures to accelerate the simulations as well as existing software simulation packages. Finally, we introduce prior research in the area of analytical performance modeling.

2.1. Computer Simulations

Computer simulations are indispensable tools to obtain solutions for problems which are otherwise intractable or can only be solved using approximate methods. Simulation methods serve as a bridge between the underlying model and theoretical predictions or between the model and results from real experiments [29]. These methods are used to obtain the macroscopic properties of interest using the microscopic details of a system of atoms or molecules [29]. They can be used to accurately calculate the structural and thermodynamic properties by replicating the macroscopic system with manageable numbers of atoms or molecules. They aid in probing a wide range of length scales (size of the system) and time scales (length of time) to study many processes and phenomena that span multiple length and time scales. Two flavors of simulation methods commonly used in physics and physical chemistry are Molecular Dynamics (MD) and Quantum Monte Carlo (MC). Classical MD is a deterministic approach

that simulates the time evolution of a system of particles, given the initial positions and velocities of the particles in the system and provides us the actual trajectory of the system. This method uses Newton's laws of motion to generate the successive configurations for the *N*-body system. MC is a stochastic approach that explores the configuration space of a system of particles and relies on high quality random number generators and a Markov process to generate the configurations. There are a number of hybrid techniques that combine the two methods, using each method for the most appropriate part of the simulation or using a simulation algorithm that alternates between MD and MC [30]. We provide an overview of the two methods describing the MC method to a sufficient detail as it is applied in this research to study quantum clusters.

2.2. Molecular Dynamics Method

Classical Molecular Dynamics is a deterministic method that is used to simulate the time evolution of a chemical or biomolecular system using Newtonian mechanics, given the initial properties of the system [31]. The chemical system could consist of a protein and its surrounding environment such as a solvent, nucleic acids, or drug molecules. These simulations require millions of timesteps to simulate even a few nanoseconds and span a wide range of length and time scales (100,000 – 1,000,000 atoms and 100 ns for biomolecular systems) depending on the system being simulated. Each timestep of the MD simulation consists of two phases: (a) force computation and (b) integration. We alternate between the two phases in each timestep and repeat for a number of discrete timesteps, each representing a few femtoseconds of simulated time. In the force computation phase, the force on each particle due to the interactions with other particles in computed. The forces computed in the first phase are used to update the atomic positions and velocities in the integration phase using Verlet or similar algorithms [31]. For simulation purposes, the atoms can be assumed to be within a box or a container. However, when the simulation is confined within a box with rigid boundaries, the atoms collide with the walls of the box, leaving very few interior

atoms. To create a simulation region that is bounded but free from walls, periodic boundary conditions are used to simulate an infinite sea of neighbors [31].

The contributions to forces in an MD simulation come from bonded and non-bonded interactions. The bonded forces are due to the covalent bonds and include bond stretch, angle, and dihedral torsion. Bonded terms involve two to four nearby atoms and hence the computational complexity of bonded forces is $O(N)$ in the number of particles. Non-bonded interactions often include the Lennard-Jones (van der Waals) and Coulombic interactions, which occur between every pair of particles in the system and the computational complexity of the non-bonded forces is $O(N^2)$ in the number of particles. A number of methods have been proposed to reduce the complexity of the non-bonded force calculations to $O(N)$ or $O(N\log N)$. For the rapidly converging Lennard-Jones force, a distance cut-off method reduces the computational complexity to $O(N)$. This method approximates the force to zero for particles separated by more than some cut-off distance. However, using the cut-off method for the slowly converging Coulombic force results in a loss of accuracy. A common approach is to split the Coulombic force into the rapidly decaying short-range and slowly decaying long-range interactions. The short-range Coulombic interactions are approximated using the distance cut-off method like the range-limited Lennard-Jones force. Methods such as Multi-level Summation [32], Particle Mesh Ewald [33] , and the Multigrid Method [34] have been developed to accelerate the long-range Coulombic interactions.

2.3. Monte Carlo Method

Monte Carlo methods were first developed by Metropolis and Ulam in 1949 [35]. Monte Carlo methods are used to solve problems that are too difficult to solve analytically or using other numerical techniques. In essence, this method works on a set of data and computes an estimated value of a property for a large number of iterations. As the number of iterations gets large, the estimate converges to the actual value. These methods rely on a large number of high-quality random numbers to obtain an estimate of the

property with long run times and hence are ideal candidates for hardware acceleration or porting to parallel computers.

There are three types of Monte Carlo methods (i) Direct Monte Carlo method (ii) Monte Carlo integration and (iii) Metropolis Monte Carlo [36]. In the Direct Monte Carlo method, random numbers are used to model processes, for example random processes in ecology or economics. In Monte Carlo integration, multi-dimensional integrals are obtained using random numbers. Metropolis Monte Carlo allows us to study the properties of quantum many particle systems. This method is a sophisticated version of Monte Carlo integration in which random walks are used to solve problems in high dimensional spaces. We describe the Quantum Monte Carlo method, a Metropolis Monte Carlo method that is used in our implementation.

2.3.1. Quantum Monte Carlo Method

Quantum Monte Carlo (QMC) methods [26, 37], have the ability to accurately treat many-body quantum mechanical systems to obtain the ground-state properties of clusters of atoms or molecules. This method provides a practical and efficient way of solving the many-body Schrödinger equation of quantum mechanics [38]. The fundamental equation in quantum mechanics is the Schrödinger equation given by Equations 2.1, 2.2, and 2.3. Equation 2.2 gives the one-dimensional time-independent Schrödinger equation for a chargeless particle of mass m moving in a potential $V(x)$. The analogous three-dimensional time-independent equation is given by Equation 2.3. Solving this equation is trivial for small systems, but as the dimensions of the system increase, it is impossible to solve the equation analytically.

$$H\psi = E\psi \tag{2.1}$$

$$-\frac{\hbar^2}{2m}\frac{d^2\psi(x)}{dx^2} + V(x)\psi(x) = E\psi(x) \tag{2.2}$$

$$\left[-\frac{\hbar^2}{2m}\nabla^2 + V(r)\right]\psi(r) = E\psi(r) \tag{2.3}$$

In the above equations, H is the Hamiltonian operator, E represents the energy of the system, ψ is the wave function, $\hbar$ is the Planck's constant divided by 2π, and ∇^2 is the Laplace operator.

Two flavors of QMC methods are Variational Monte Carlo (VMC) and Diffusion Monte Carlo (DMC). In VMC, we use random walks to calculate the multi-dimensional integrals of expectation values, such as the total energy. In the DMC method, we start with a distribution of walkers, which consist of a finite number of atoms. These walkers are propagated through time and created or destroyed each iteration using a birth-death process. To control the total population of the system, the internal reference energy is constantly adjusted. After a large number of iterations, the energy of the system converges to the true energy of the system.

VMC applies the variational method to approximate the ground state of the system. This method employs a set of adjustable parameters to yield a trial wave function $\psi_T(x)$ that, when optimized, best approximates the exact wave function. It has the advantage of being simple and easy to implement. Also, it is relatively insensitive to the size of the system, and hence can be applied to large systems where some other methods are computationally infeasible. Figure 2.1 shows the Variational Monte Carlo algorithm, which is used in this work.

Step 1: Select a reference configuration, $R(x, y, z)$ at random.

REPEAT (for *I* iterations – equilibration and steady-state)

  REPEAT (for all configurations)

    Step 2: Obtain a new configuration, *R'* by adding a small random displacement to all the particles in the above configuration.

    Step 3: Compute the ground-state properties (e.g., energy, wave function) of the particles in the current configuration, *R'*.

    Step 4: Accept or reject the present configuration using the ratio of the wave function values,

$$p = \left| \frac{\psi_T(R')}{\psi_T(R)} \right|^2$$

      If $p \geq 1$, R' is accepted.

      If $p < 1$, if $p < rand()$ R' is rejected and R and its properties are retained.

  UNTIL finished (for all configurations)

UNTIL finished (for all iterations)

Figure 2.1: Variational Monte Carlo algorithm

Step 1 of the algorithm consists of choosing a *reference configuration*, *R*, for the system of particles using a Cartesian co-ordinate system. We move all the particles in this configuration to yield a new configuration, called a *proposed configuration*, *R'* in step 2. This is done by adding a small uniform displacement to the particle positions in the configuration. In step 3, we compute the properties for the particles in this configuration. We perform the following $O(N^2)$ calculations: distance calculation between pairs of particles, pair-wise potential energy, and trial wave function calculations. After obtaining the wave function, we also compute its first and second derivatives in order to compute the kinetic energy. The energies are summed to yield the total energy for this configuration. To ensure that the configurations are asymptotically drawn from the square of the known trial wave function $\psi_T(x)$, we accept or reject this configuration (and associated properties) by determining the ratio *p* in step 4. As shown in step 4 in Figure 2.1, we obtain *p* using the ratio of the values of the trial wave functions. If the value of *p* is greater than or equal to 1, we accept the proposed configuration. Otherwise, we compare *p* with a uniformly distributed random number to decide whether to accept or reject the configuration. If the proposed configuration is accepted, we retain its properties; otherwise we keep the properties of the

reference configuration. Steps 2-4 are repeated until asymptotic behavior is attained (typically requiring thousands of iterations).

2.3.2. Model System

We apply the VMC method to investigate the quantum mechanical ground states of a rare gas cluster system consisting of helium atoms. The extremely weak helium-helium interatomic interactions combined with the small atomic mass make the helium clusters weakly bound and show interesting properties, among which is their ability to attain a superfluid state. By far, the only methods that can accurately model highly quantum clusters such as helium clusters are the quantum Monte Carlo methods. They can be applied to study pure as well as doped helium clusters. In this section, we provide the analytical functions for the potential energy function and trial wave function for a model system of helium atoms that is used in this research. In VMC, we choose a form for the trial wave function, $\psi_T(x)$, characterized by a set of parameters and evaluate the energy given by Equation 2.4.

$$E \leq \langle E \rangle = \int dR \frac{H\psi}{\psi} \rho(R) \tag{2.4}$$

where R is a $3N$-dimensional vector composed of particle co-ordinates, (x, y, z). The trial wave function is a parameterized function, whose parameters can be varied to minimize the integral in Equation 2.4 and $\langle E \rangle$ is an upper bound to the exact ground-state energy, E_0. The integral is evaluated using the Monte Carlo method, where we sample the particle positions from the probability density function, $\rho(R)$, given by Equation 2.5. E is estimated using Equation 2.6. The probability density function is sampled using the Metropolis algorithm [35].

$$\rho(R) = \frac{\psi(R)}{\int dR\, \psi(R)} \tag{2.5}$$

$$\langle E \rangle \approx \frac{1}{M} \sum_{i=1}^{M} \frac{H\psi_T(R_i)}{\psi_T(R_i)} \tag{2.6}$$

Potential Energy Calculation

The Hamiltonian for the atomic co-ordinates for an *N*-atom cluster is given by Equation 2.7, where r_{ij} is the distance between two helium atoms, *i* and *j*, and ∇_i^2 is the Laplacian of atom *i*, *m* is the atomic mass, and $\hbar$ is Planck's constant divided by 2π.

$$H = \frac{-\hbar^2}{2m} \sum_{i=1}^{N} \nabla_i^2 + V_{total} \tag{2.7}$$

While studying atomic clusters, the many-body potential is usually expressed as a summation of various interaction terms as given in Equation 2.8.

$$V_{total} = \sum_{i<j} V_{i,j} + \sum_{i<j<k} V_{i,j,k} + \cdots \tag{2.8}$$

In Equation 2.8, $V_{i,j}$ and $V_{i,j,k}$ are the two- and three-body interaction terms, respectively. We will use a two-body model for our simulations and hence the potential energy, V_{total} is approximated as a sum of the *N* (*N*-1)/2 pair contributions as shown in Equation 2.9.

$$V_{total} \approx \sum_{i<j}^{N} V(r_{ij}) \tag{2.9}$$

A number of helium-helium potentials are available in the literature [39, 40] to accurately model the helium-helium interatomic interactions. The *HFDB-He* potential [39] and *SAPT2-He* potentials [40] are employed in this study. The *HFDB-He* potential is given by Equations 2.10 and 2.11. This is an accurate and frequently employed potential for helium systems. The functional form of the *SAPT2-He* potential is given in Equation 2.12. The parameters used for these potentials are given in Tables 2.1 and 2.2,

respectively. In Equation 2.12, $f(r)$ is a position-dependent retardation coefficient described using polynomials in r [40]. We plot the *HFDB-He* and *SAPT2-He* potentials as functions of the helium-helium separation in Figure 2.2.

$$V_{HFDB} = \varepsilon\left(A\exp\left(-\alpha x + \beta x^2\right) - F(x)\sum_{j=0}^{2}\frac{c_{2j+6}}{x^{2j+6}}\right) \tag{2.10}$$

where,

$$F(x) = \begin{cases} \exp\left[-\left(D/x - 1\right)^2\right] & \left(x < D\right) \\ 1 & \left(x \geq D\right) \end{cases} \tag{2.11}$$

In Equations 2.10 and 2.11, $x = r / r_m$

$$V_{SAPT2} = Ae^{-\alpha r+\beta r^2} - \left[1 - \left(\sum_{k=0}^{6}(\delta r)^k / k!\right)e^{-\delta r}\right]C_6 f(r)/r^6 - \sum_{n=4}^{8}\left[1 - \left(\sum_{k=0}^{2n}(\delta r)^k / k!\right)e^{-\delta r}\right]C_{2n}/r^{2n} \tag{2.12}$$

Table 2.1: Parameters for *HFDB-He* potential [39]

Parameter	Value
A	1.883101e5
α	10.43329537
c_6	1.36745214
c_8	0.42123807
c_{10}	0.17473318
β	-2.27965105
D	1.4826
ε	7.609 cm^{-1}
r_m	5.599 a_0

Table 2.2: Parameters for *SAPT2-He* potential [40]

Parameter	Value
A	6.56912828 E_h
α	1.88648251 $bohr^{-1}$
β	-6.20013490x10^{-2} $bohr^{-2}$
δ	1.94861295 $bohr^{-1}$
C_6	1.4609778 a.u.
C_8	14.117855 a.u.
C_{10}	183.69125 a.u.
C_{12}	3.265x10^3 a.u.
C_{14}	7.644x10^4 a.u.
C_{16}	2.275x10^6 a.u.
ε	7.68789 cm^{-1}
r_m	5.60234 bohr

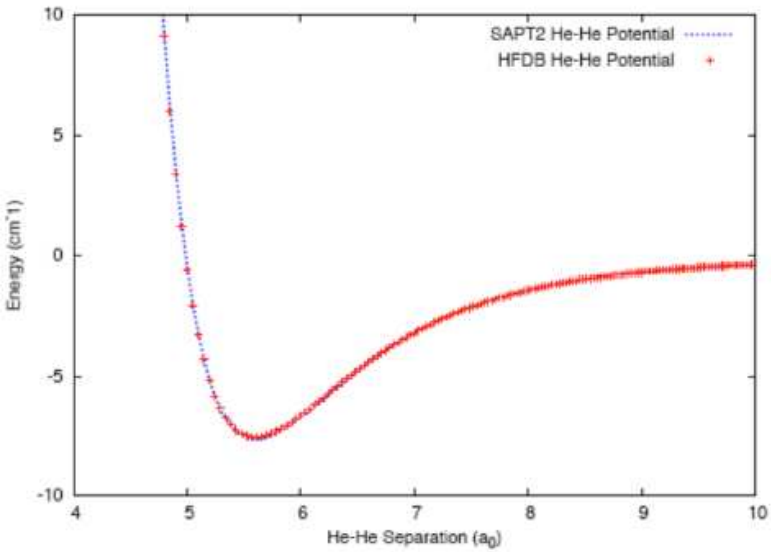

Figure 2.2: He-He potentials employed in this work

Wave Function Calculation

We provide the analytical function of the trial wave function used in our QMC algorithm. We are interested in the ground-state wave functions of bosonic systems. Wave functions with different parameters are cited in the literature [41, 42]. The wave function is the exponential of the two-body interaction term, $T_2(r)$, which is a function of particle separations as shown in Equation 2.13.

$$\psi(r) = \exp\left[T_2(r)\right] \tag{2.13}$$

We use the analytic form proposed in [41] for ψ given by Equation 2.14.

$$\psi(r) = \psi_s(r)\psi_l(r) \tag{2.14}$$

where $\psi_s(r)$ and $\psi_l(r)$ denote the short- and long- range functions shown in Equations 2.15 and 2.16, respectively. The wave function parameters are shown in Table 2.3.

$$\psi_s(r) = \exp\left[P(u)\right],\ u = r^{-1},\ P(u) = \sum_{k=0}^{5} a_k u^k \tag{2.15}$$

$$\psi_l(r) = r^b \exp\left[a\, r^{\alpha}\right] \tag{2.16}$$

Taking the natural logarithm of both sides of the wave function in Equation 2.14 and using Equations 2.15 and 2.16 yields Equations 2.17 and 2.18, respectively.

$$\ln\left[\psi(r)\right] = \ln\left[\psi_s(r)\right] + \ln\left[\psi_l(r)\right] \tag{2.17}$$

$$\ln\left[\psi(r)\right] = P(u) + b\ln r + ar^{\alpha} \tag{2.18}$$

Representing the left hand side of Equation 2.18 by $T_2(r)$ and computing the exponential function of $T_2(r)$ yields the wave function $\psi(r)$, given by Equation 2.13. We plot the He-He wave function in Figure 2.3.

Table 2.3: Parameters for the trial wave function (in atomic units) [41]

Parameter	Value
a	-0.01
b	-0.85
α	0.545031
a_0	-1.30801
a_1	-38.8646
a_2	310.061
a_3	-1370.01
a_4	2484.45
a_5	-3674.60

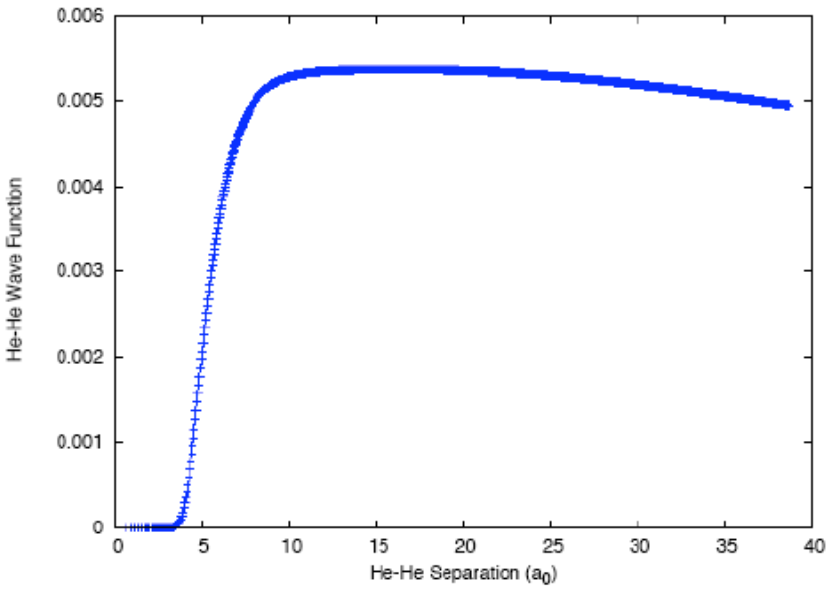

Figure 2.3: He-He wave function

The first and second derivatives, namely the gradient and laplacian functions of the wave function, ψ, are used in the kinetic energy calculations and given in Equations 2.19 and 2.20.

$$\nabla\psi = \frac{d}{dr}\left[\exp T_2(r)\right] = \exp T_2(r)\left[\frac{b}{r} + a\alpha r^{\alpha-1} - \sum_{k=0}^{5} k.a_k u^{k+1}\right] \tag{2.19}$$

$$\begin{aligned}\nabla^2\psi &= \frac{d^2}{dr^2}\left[\exp T_2(r)\right] \\ &= \exp T_2(r)\left[-\frac{b}{r^2} + a(\alpha-1)\alpha r^{\alpha-2} - \sum_{k=0}^{5} k.a_k u^{k+1}\right] + \nabla\psi\left[\frac{b}{r} + a\alpha r^{\alpha-1} - \sum_{k=0}^{5} k.a_k u^{k+1}\right]\end{aligned} \tag{2.20}$$

2.4. Development Environment

In the following sections, we survey the RC platforms and graphics processing units for general-purpose computing. These systems allow us to partition the applications such that critical components can be mapped onto hardware and the rest of the application retained in software providing significant performance gains over an entirely software implementation running on a processor. We also discuss the programming environments for the above platforms.

2.4.1. Reconfigurable Computing and Graphics Processors

Reconfigurable computing (RC) devices in the form of Field-Programmable Gate Arrays (FPGAs) consist of an array of logic blocks configured to achieve a desired functionality using a set of programmable routing resources. As discussed in Chapter 1, high performance reconfigurable computing (HPRC) systems, which incorporate RC elements into the computing nodes, provide a significant performance advantage over software-only solutions. Present FPGAs have increased gate densities, providing the capability to use a floating-point or customized precision for the calculations, depending on the

application requirements. A number of FPGA based solutions are offered by vendors such as Cray (*XD1* [43] and *XT5$_h$/XR1* [44]), SRC Computers (*MAPstations* [45]), SGI (*Reconfigurable Application Specific Computing – RASC* [46]), DRC Computers (*Reconfigurable Processor Unit – RPU* [47]), XtremeData [48] and Nallatech [49]. These systems that couple FPGAs with conventional microprocessors using high-bandwidth and low latency interconnects, are of increasing interest to the computational science and engineering community. The RPU from DRC Computers is equipped with FPGAs that directly fit in the microprocessor sockets, and provide direct access to the memory at Hypertransport speed and latencies. The product from Nallatech called the FSB FPGA accelerator module [50] features Xilinx Virtex-5 generation FPGAs which interface directly into an Intel Xeon processor socket and the 64-bit front side bus. The FPGA supercomputer project undertaken by the FPGA High Performance Computing Alliance (FHPCA) [51] at the University of Edinburgh has resulted in Maxwell [52], a 64-FPGA supercomputer, built from commodity parts and plug-in FPGA cards.

Hardware-description languages such as VHDL and Verilog are commonly used for FPGA programming. However, with the significant learning curve of these languages, there is a need for the scientists to work closely with hardware designers to develop accelerated scientific codes. Another alternative is Viva by Starbridge systems [53], which is a graphical object-oriented programming language that can be used to create a design directly compiled to a FPGA configuration by a non-VHDL expert. High-level languages (HLLs) have been developed in an attempt to break the barrier of the tedious FPGA programming process. Handel-C, a superset of ANSI-C can be used to compile high-level algorithms directly into gate level hardware [54]. MitrionC is a HLL provided by Mitrionics [55] and is a C-like programming language that allows the development of massively parallel programs for the Mitrion Virtual Processor, a massively parallel soft-core processor adapted to the implemented algorithm. The processor is then instantiated on the FPGA. Nallatech's DIME-C [49] based on a subset of ANSI-C is a parallelizing C-to-VHDL compiler and integrated with Nallatech's DIMETalk tool which enables the user to create the

FPGA hardware components and generate the bitstream for the final FPGA design.

Graphics processors have been traditionally used for rendering 3D graphics. The demand for more realistic graphics rendering from applications in the multibillion-dollar video game industry have led to a number of architectural innovations and improvements in the graphics hardware. The fixed pipelines on GPUs are now replaced with programmable shaders. These capabilities extend the use of graphics processors for a number of non-graphics general-purpose computing applications. Presently GPUs for general-purpose computing are available from both NVIDIA [9] and AMD/ATI [10]. A number of research efforts where computationally intensive problems have been mapped on GPUs, also known as general-purpose GPU (GP-GPU) computing [22]. Present GPUs support IEEE compliant single- and double- precision making them attractive for scientific computing. The Intel Larrabee project [56] is aimed at a GPU chip that will compete with the NVIDIA GeForce and AMD/ATI Radeon video cards. It is based on the x86 instruction set and intended for general-purpose or stream processing tasks such as in ray tracing or physics processing.

Prior to the advent of general-purpose programming languages for GPU, GPU implementations used 3D-rendering APIs such as OpenGL [57] and DirectX [58]. With these languages, there is a need to pose problems in terms of graphics primitives (textures, triangles) using 3D graphics API functions. However, these APIs also hide architectural details that are significant while programming GPUs for general-purpose computing applications. For example, graphics drivers make decisions such as where the data resides in the memory and when it is copied [59]. Efforts from various vendors to create general-purpose languages without having to invoke graphics API calls have led to languages such as Compute Unified Device Architecture [60], Brook [61], Sh [62] and its successor, RapidMind [63], and Microsoft's Accelerator [64]. Through added driver and hardware support, the need to proceed through the entire graphics pipeline (transformation of vertices, rasterization) and the use of graphics API is avoided.

NVIDIA's CUDA language provides extensions to C and uses a Single Program Multiple Data (SPMD) programming model with the concept of a computational grid consisting of many blocks with a number of threads (currently up to 512) in each block. With CUDA, the data to be processed by the GPU needs to be copied from the host to the device memory. Once the data is resident on the GPU, the kernel on the GPU is executed by all blocks in a SPMD fashion. Within each block, the threads operate using a Single Instruction Multiple Data (SIMD) model. AMD provides two API's that use the streaming model: Brook+ and Compute Abstraction Layer (CAL). Brook+ is a high-level stream computing language based on the Brook project at Stanford University. Brook+ is a C-like programming language using kernels running on the GPU in conjunction with the host side code written in C. The GPU is viewed as a streaming coprocessor which executes a kernel over all elements of an input stream, and places the results into an output stream [61]. CAL provides low-level access to the GPU for development and performance tuning. CAL also supports application development on multi-GPUs and multi-core processors [59]. With CAL, the need for copying data from the host memory to the local GPU memory is eliminated and the GPU can directly read from or write to the host memory [59]. OpenCL is a cross-platform framework that allows us to write portable code for heterogeneous systems with a diverse mix of multi-core CPUs, GPUs, and Cell architectures [65].

2.5. Survey of Architectures for MD and QMC

A number of science and engineering applications such as computational fluid dynamics [66, 67], molecular modeling [14, 68], and linear algebra [69, 70] have demonstrated significant performance advantages using the afore-mentioned FPGA and GPU-based platforms. We present the related work for the MD and MC simulation techniques covering the following topics: architectures for MD and widely used MD software packages, architectures for MC in general, and for QMC, and popular QMC software packages.

2.5.1. Architectures and Software Packages for Molecular Dynamics

Molecular Dynamics is a widely used tool to study chemical and biomolecular systems, including proteins, cell membranes, and DNA. Due to the enormous computational requirements, MD simulations have been targeted to HPC systems. To accelerate the force calculations within MD, special-purpose computers, reconfigurable computers, and GPU-based platforms have also been targeted. The Blue Gene project was originally started to address the grand challenge problem of protein folding [71]. In [3], the authors use MD for simulating proteins, along with Blue Matter, a software framework for scaling these simulations on systems with thousands of nodes [72]. In addition to harnessing the computational power from these massively parallel systems, special-purpose machines have been developed solely to speed up the kernels of MD simulation. Special-purpose computers, such as GRAPE (Gravity Pipe) systems [73], have been developed to accelerate gravitational *N*-body and MD simulations. These special-purpose engines are used for computationally intensive long-range force calculations, such as gravitational, Coulombic, and van der Waals forces. A number of GRAPE machines that have succeeded such as GRAPE-2A [74] and MD-GRAPE [75] are designed for MD simulations. The Protein Explorer with the MDGRAPE-3 chip [74] is a petaflop special-purpose computer designed for non-bonded force calculations in MD simulations with potential target applications including drug design, protein analysis, and material sciences. The MDGRAPE-3 chip calculates the non-bonded forces, such as Coulombic and van der Waals forces, and the remaining calculations are performed on the host computer. Anton [76] is yet another special-purpose massively parallel machine being designed to overcome the timescale limitations of systems like MD-GRAPE, providing the capability to execute millisecond-scale classical MD simulations of biomolecular systems.

When we accelerate MD simulations using FPGA/GPU co-processors, we begin with an optimized, preferably production-level MD code and port its kernels to the co-processors. AMBER [77], NAMD [78], CHARMM [79], LAAMPS [80], and GROMACS [81] are some currently used software MD

packages. FPGA and GPU architectures for MD have used AMBER, NAMD, or an experimental software version as the baseline MD code. AMBER is a package of molecular simulation programs written in Fortran that uses Message Passing Interface (MPI) on parallel processors [77]. NAMD is a parallel MD code that is reported to scale to many thousands of processors. It has been developed by a research group at the University of Illinois, Urbana-Champaign, written using the Charm++ parallel programming model, and designed for high-performance simulation of large biomolecular systems [82].

Research efforts have utilized FPGAs and GPUs for MD simulations. The most relevant work to our research is described here. One reconfigurable MD simulator maps all the MD tasks onto the Transmogrifier 3 (TM3) FPGA platform that consists of multiple interconnected FPGAs [83]. A complete simulation, including pair-wise interactions and position updates, is implemented using VHDL on the TM3 platform. The particle co-ordinates are stored in SRAM banks on the TM3. For every timestep, the distance between a pair of particles is computed and used by the Lennard-Jones module to calculate the force between a pair of particles. The forces are accumulated by the acceleration update module. The total acceleration for the particle is used by the Verlet update module to update the positions and velocity of each particle. These calculations are repeated for all particle pairs and the process is repeated for the next timestep. The system simulates 8,192 particles in 37 seconds at 26 MHz and uses fixed-point representation to minimize hardware requirements. The authors extrapolate the current results to show that a speedup of over 20*x* can be achieved with modern FPGAs parts running at 100 MHz.

Another effort relevant to our QMC work implements the Lennard-Jones potential and forces in VHDL on a Xilinx Virtex-II Pro XC2VP125 FPGA [84]. A pipelined design with 119 stages is used, consisting of several adders, multipliers, dividers and a square root operation. The implementation uses IEEE 754 double-precision floating-point arithmetic and computes the 64-bit potential and forces. They report a performance of 3.9 GFLOPS throughput compared to 1.5 GFLOPS on an Itanium2 900 MHz system.

A few efforts have demonstrated the use of high-level languages to port production-level MD codes to reconfigurable systems. [85] presents the results of porting the NAMD code on the SRC reconfigurable platform using a high-level programming language, MAP C [45]. AMBER code is ported on the SRC platform in [86]. In both cases, the authors report a 3x speedup over the software implementation.

In [14], the authors demonstrate the use of GPUs for the calculation of long-range electrostatic and non-bonded forces for MD simulations and obtain a 10-100x speedup on NVIDIA GPUs using CUDA programming over an optimized CPU implementation.

Tables 2.4-2.6 present a survey of FPGA, GPU, and special-purpose architectures for MD simulations. In each case, we compare the baseline MD code, the development platform, development language, the numerical precision, number and types of particles, the forces accelerated using the architecture, methods of force calculation, and the speedup obtained over the serial version on the CPU. For MD architectures utilized in biomolecular simulations, we also list the simulation timescale.

Table 2.4: Survey of MD on reconfigurable hardware

	Baseline MD code	Development platform	Precision	Development language	Number and types of particles	Forces/ Method of calculation	Speedup
[86]	AMBER† (Fortran, MPI)	SRC-6E MAPstation – Dual 2.8 GHz Xeon with MAP processor – 2 XC2VP1000 FPGA seven 4 MB SRAM banks	Single-precision	Carte, MAPC	20,000 – 200,000	Ewald method, Only the direct part (80-90%) on FPGA	100 MHz 3x speedup over the MAPstation host
[85]	NAMD†	SRC-6C MAPstation – Xilinx Virtex-II XC2V6000 FPGA SRC-6E MAPstation Xilinx Virtex-II Pro XC2VP100 FPGA	Single-precision	Carte, MAPC	92,224 atoms	Smooth Particle Mesh Ewald method – Coulomb's force Cut-off method – LJ force	100 MHz 3x speedup
[84]	Reference MD code – Velocity Verlet algorithm	SRC-6E MAPstation Virtex-II XC2V6000-4 FPGA, 100 MHz	Single-precision	SRC development suite Carte	52,558 – 8 atom types 32932 – 17 atom types	HW/SW LJ and cutoff Coulomb method Direct evaluation	100 MHz 2x speedup over MAPstation host
[87]	Reference MD code	Xilinx Virtex-II Pro XC2VP125 FPGA	Double-precision	VHDL	10,000 molecule	Lennard-Jones Direct evaluation	122 MHz 3.9 GFlops, 1.5 GFlops on an Itanium2 900 MHz
[88]	ProtoMol	Annapolis Micro Systems Xilinx Virtex-II Pro XC2VP70	Fixed-point	VHDL	8000 particles, 26 particle types	Lennard-Jones and Coulombic force	5.5x speedup over a 2.8GHz PC
[83]	Not integrated into an MD code	TM3 platform	Fixed-point (between 22 and 76 bits)	VHDL	8192 particle model	Velocity Verlet algorithm Only LJ force – table lookup, interpolation	26 MHz, 37 sec per time step, 2.4 GHz P4 – 10.8 sec Extrapolated speedup of 40x to 100x

†Production-level MD codes

Table 2.5: Survey of MD on graphics processors

	Baseline code	Development platform	Precision	Development language	Number of particles	Forces / Method of calculation	Speedup
[24]	Reference MD code	NVIDIA GeForce 7900 GTX	Single-precision	OpenGL	<= 2048 atoms	Lennard-Jones	6x over 2.2GHz Opteron
[14]	NAMD[†]	NVIDIA GeForce 8800GTX (128 programmable processing units, 330 GFlops peak single)	Single-precision	CUDA	92,224	Cut-off based method, Lennard-Jones and Coulomb's electrostatic force	6 GPUs – 10.6x acceleration
[89]	--	NVIDIA Tesla C870	Single-precision	CUDA	96,603	Cut-off based method electrostatic force	12-20x

Table 2.6: Survey of MD on ASICs

	Hardware specifications	Precision	Number and types of particles	Forces / Method of calculation	Speedup
[76]	512 processing nodes (400 MHz) Compatible with CHARMM, AMBER	Fixed-point arithmetic	Up to 200,000 particles, Millisecond Timescale	Van der Waals – cut off method , Electrostatic – k-space Gaussian split Ewald method	Expected speedup of 100x over BlueGene/L
[74]	MDGRAPE-3 chip (165 GFlops 250 MHz)	Floating-point and Fixed-point	32,768	Lennard Jones and Coulomb's force	--

[†]Production-level MD codes

2.5.2. Architectures for Monte Carlo Simulations

Monte Carlo methods have been widely used to solve problems in science, engineering, and finance. Reconfigurable Computing has been used to accelerate Monte Carlo simulations for various applications. A reconfigurable Monte Carlo clustering processor (MCCP) is proposed in [90] where a number of MC algorithms used in statistical physics are implemented. A reconfigurable architecture is used to accelerate the computationally intensive parts of a Monte Carlo application that simulates radiative heat transfer simulation in a 2-D chamber using Virtex-II and Virtex-II Pro FPGAs [21]. This application simulates a large number of photon emissions and absorptions between the surfaces of a 2D enclosure and the information during the simulation is used to compute the heat transfer coefficient between the two surfaces. A 10.37*x* speedup is reported with three single-precision floating-point pipelines on the Virtex-II Pro FPGA over the software application running on a 3.0 GHz Xeon processor. A reconfigurable architecture is also used to accelerate a financial engineering application for the Brace, Gatarek, and Musiela (BGM) interest rate model for pricing derivatives using a Gaussian-distributed random number generator implemented in hardware [91]. The authors use customized low-precision floating-point formats and achieve a 25*x* speedup using a development platform that consists of a Xilinx Virtex-II Pro FPGA over the software running on a 1.5 GHz Intel Pentium 4 processor.

Random Number Generators for Monte Carlo Simulations

High-quality uncorrelated random numbers are indispensable for Monte Carlo simulations. The Scalable Pseudo Random Number Generator (SPRNG) library available from Florida State University [92] is a scalable package designed for parallel pseudo random number generation, especially for large-scale parallel Monte Carlo applications. This library provides different types of random number generators: Combined Multiple Recursive Generator, 48-bit and 64-bit Linear Congruential Generator, Modified Lagged Fibonacci Generator, and Multiplicative Lagged Fibonacci Generator. Research efforts have led to the hardware accelerated SPRNG (HASPRNG) library, in which the SPRNG suite of random number

generators is ported on FPGAs [93]. The Mersenne Twister is another well-known pseudorandom number generator for Monte Carlo simulations and provides good quality random numbers with the advantage of speed and portability [94]. In our Monte Carlo implementation, we generate the random numbers on the processor using the SPRNG library. Our choice is motivated by the desirable characteristics of SPRNG such as high-quality, speed, scalability, portability, and the ability to produce "independent" and "reproducible" streams of random numbers for parallel Monte Carlo application.

2.5.3. Architectures and Software Packages for Quantum Monte Carlo

Here, we report work pertaining to the use of computer architectures to accelerate QMC simulations and commonly used QMC software packages. There are a number of QMC codes such as CASINO, CHAMP, QMCBeaver, and ZORI that provide a user-friendly framework to perform QMC calculations on a variety of processing platforms. CASINO is code developed at Cambridge University for performing QMC electronic structure calculations (i.e., solving the Schrödinger equation that describes the behavior of interacting electrons and ions) on finite atoms and molecules and crystalline systems [95]. It is written entirely in Fortran90 and uses Message Passing Interface (MPI) for use on parallel machines. CHAMP is a QMC suite of programs for electronic structure calculations on a variety of atomic and molecular systems from Cornell University [96]. ZORI is an open source QMC code from University of California, Berkeley for electronic structure calculations [97]. QMCBeaver is a QMC code written in C++ developed by California Institute of Technology and performs VMC and DMC calculations and can be compiled on GPUs [98] .

QMC@home is a distributed computing project for Berkeley Open Infrastructure for Network Computing (BOINC) for the development of Quantum Monte Carlo for use in quantum chemistry [99]. BOINC was initially developed for the SETI@home project to help researchers utilize the processing power of computers around the world. The QMC@home project is aimed at calculating the properties of molecules

using the DMC method by utilizing the idle cycles of computers from volunteers around the world. The only work reported in literature that is closely related to our QMC research is the GPU implementation of QMCBeaver [100] that is used to compute electronic structure of a given molecule on the NVIDIA 7800 GTX GPU. This implementation achieves a 30*x* speedup for individual kernels and an overall speedup of 6x over the optimized software application running on a 3.0 GHz Intel Pentium 4 processor [100]. The challenge in this implementation is the lack of a fully compliant IEEE floating-point implementation. To achieve a better accuracy, the authors use the Kahan summation formula [101] in matrix multiplications to achieve an accuracy that matches CPU single-precision. Our work uses the QMCC code [27] to calculate the properties of Helium atoms, which is different from the specific QMC application in [100]. We obtain a 40*x* speedup (using fixed-point) and 225*x* speedup on the GPU (using mixed-precision), which deliver the accuracy required for our application. Table 2.7 summarizes the prior work that has targeted QMC on emerging architectures.

From our survey of related literature (Tables 2.4-2.7), we can see that a number of efforts have targeted reconfigurable computing and GPUs for force calculations in MD. Other than the QMC packages, the QMC@home project, and the QMCBeaver on GPU [100], there is no prior work that explores HPRC or GPU architectures for our specific QMC application that calculates the ground-state properties for inert gas clusters. We focus on exploring individual FPGA, and GPU platforms for the QMC, application so we can provide the capability to the scientific community to use these emerging architectures through user-friendly hardware-accelerated simulation frameworks. We also seek to understand through this work, the best ways of partitioning and mapping the QMC application on to hybrid computing platforms that combine CPUs, FPGAs, and GPUs.

Table 2.7: Survey of architectures for QMC

	Baseline QMC code	Platform	Precision	Application	System size	Speedup
[100]	QMCBeaver [98]	NVIDIA 7800 GTX GPU	Single-precision	Electronic structure calculations	78432 (no of electrons x no of basis functions	6*x* over 3.0 GHz Intel Pentium 4 processor
Our research	QMCC [27]	Cray XD1 HPRC platform and NVIDIA Tesla GPUs	Fixed-point (52-bit) on FPGA, Mixed-Precision on GPU	Ground-state properties of Helium	4096 atoms (on FPGA)	40*x*, 225*x* on GPU over dual-core dual-processor AMD Opteron 2.2 GHz

2.5.4. Performance Modeling

Performance analysis is an important tool in identifying the best ways of mapping applications to available computational resources. Three broad classes of performance evaluation techniques are measurement, simulation, and analytical modeling [102]. Measurement is an accurate approach, but it requires that the system under study is available. This method is often used for performance tuning to improve the performance using the measured results. However this method requires us to perturb the system through monitors or probes. Another drawback with this method is that it cannot be used for performance prediction or in the analysis of different system configurations. Simulation involves building a model for the system's behavior and driving it with trace data. In this technique, it is not necessary for the system to exist and hence it can be used for performance prediction. This method provides visibility, controllability, and flexibility [103]. However, large simulations can take a long time and validating the simulation model is difficult as it may not be practical to run all possible test cases. The third technique, analytical modeling, involves the construction of a mathematical model for the system at the desired level of detail. Analytical models allow us to explore the system performance prior to its construction and gain

insights into the performance variations for different system parameters. However, the analytical models are difficult to develop with sufficient detail and validate.

An analytical performance model for studying application performance in shared, heterogeneous high performance computing environments is proposed in [104]. This model focuses on Synchronous Iterative Algorithms (SIAs), a class of a set of fork-join algorithms and includes the effects of application and background load imbalance. This model is extended to study the mapping of applications onto shared heterogeneous workstations containing reconfigurable computing devices in [105]. This work concerns the modeling of system-level, multi-FPGA architectures with variable computational loading. Analytical models are then used to estimate the various components of the applications' execution time. A methodology called the RC Amenability Test (RAT) is proposed for analyzing an application's migration to a particular RC platform [106]. The RAT approach considers the amenability of an application to the RC hardware taking the factors such as throughput, numerical precision, and resource usage into account. In this work, we are interested in building performance models for individual CPU, FPGA, and GPU platforms for the QMC application and validating the model from actual results. This will help us understand how to best map the application to the above platforms as well as identify the best ways of mapping applications on to future hybrid architectures consisting of one or more of these platforms.

3. CPU IMPLEMENTATION

This chapter provides an overview of the software Quantum Monte Carlo implementation on the CPU, followed by the parallel implementation on multiple processors. We use the algorithm to simulate a model system of Helium atoms as described in Chapter 2.

3.1. Serial Implementation

The CPU implementation of the VMC algorithm consists of two phases: Initialization Phase and Iterative Phase. Figure 3.1 shows the flow chart of the serial QMC algorithm.

3.1.1. Initialization Phase

In this phase, we initialize the random number generator, in this case the Additive Lagged Fibonacci Generator (ALFG) from the Scalable Parallel Random Number Generator (SPRNG) library with an initial seed, so that invoking the *sprng* function yields uniformly distributed random numbers in [0,1). The ALFG generator is a recurrence-based generator that has two important properties for MC simulations: the maximum period of the generator is $2x10^{394}$; the generator is also attractive for parallel MC simulations because distinctly seeded random number generators can run in parallel with no correlation between the random numbers [92]. Next, we initialize the *reference configuration* (or walker). We invoke the function, *compute*( ) once to obtain the properties, the wave function, and total ground-state energy (sum of potential energy and kinetic energy) for the reference configuration.

3.1.2. Iterative Phase

In this phase, for each configuration (or walker), we move its atoms to yield a new configuration, called a *proposed configuration*. We add small displacements using the random numbers from SPRNG to the (*x*, *y*, *z*) positions of the atoms in the present configuration. The next step is to compute the properties for the atoms in this configuration. We perform the following O(N^2) calculations: distance calculation between pairs of atoms and pair-wise potential energy and wave function calculations. After obtaining the wave function, we also compute its first and second derivatives in order to compute the kinetic energy. The energies are summed to yield the total energy for this configuration. We accept or reject this configuration (and associated properties) by determining the fraction *p*. The value of *p* is obtained using the ratio of the square of the wave function values of the proposed to the reference configuration. If the value of *p* is greater than or equal to 1, we accept the proposed configuration. Otherwise, we compare *p* with a uniformly distributed random number from SPRNG to decide whether to accept or reject the configuration. If the proposed configuration is accepted, we retain its properties; otherwise, we keep the properties of the reference configuration. After completing the equilibration iterations (I_{EQ}), we start updating the properties by accumulating the energies during the steady-state iterations (I_{SS}). This is done for all configurations in the system and repeated for a total of ($I_{EQ} + I_{SS}$) iterations.

3.2. Parallel Implementation

Parallelizing the Monte Carlo simulations involves distributing the walker population among the processors and using independent random number streams on each processor. A master-slave strategy is employed to parallelize the application. Each processor uses its own streams for random number generation from SPRNG and runs the VMC algorithm for a specified number of iterations. Each processor initializes SPRNG with a unique seed. This is important because when the equilibration process is finished, the positions of the atoms should be uncorrelated so we can start accumulating the energies.

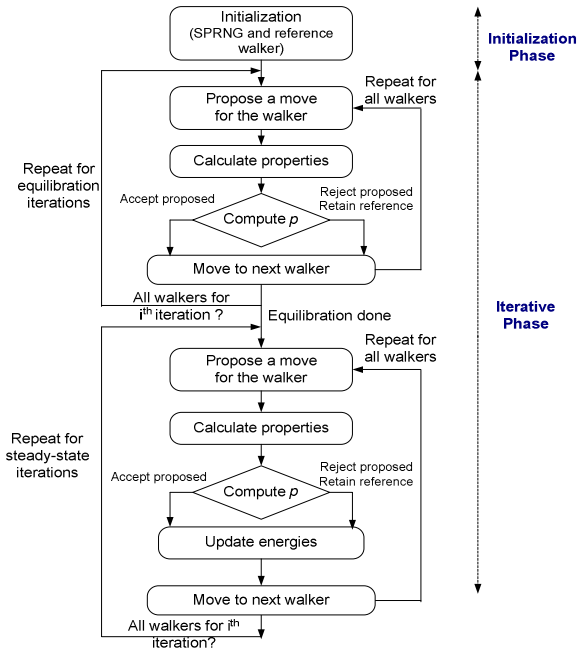

Figure 3.1: Flow chart of the serial VMC algorithm

The master process assigns an equal fraction of the total number of walkers to each processor including itself. It is also responsible for gathering the results from different processes. The slave processes execute their tasks and return the local estimates of the properties to the master process once every process has finished its work. The master process is responsible for receiving the results from the slave processes and accumulating the results including its results (i.e., the kinetic energy, potential energy and the total energy) to obtain global estimates. Each slave process works on the same number of configurations and for the same number of iterations. Also, no communication is required between the processes until we reach the end of the iterations. Figure 3.2 shows the flow chart of the parallel VMC algorithm. This method parallelizes the VMC algorithm by distributing the configurations among the processes with each process performing independent VMC calculations. We can also exploit functional parallelism by having each process compute different functions of the VMC algorithm. Analyzing the performance on a single computing node using performance models will allow us to understand the best ways of partitioning the application on multiple nodes for load balancing while parallelizing the algorithm.

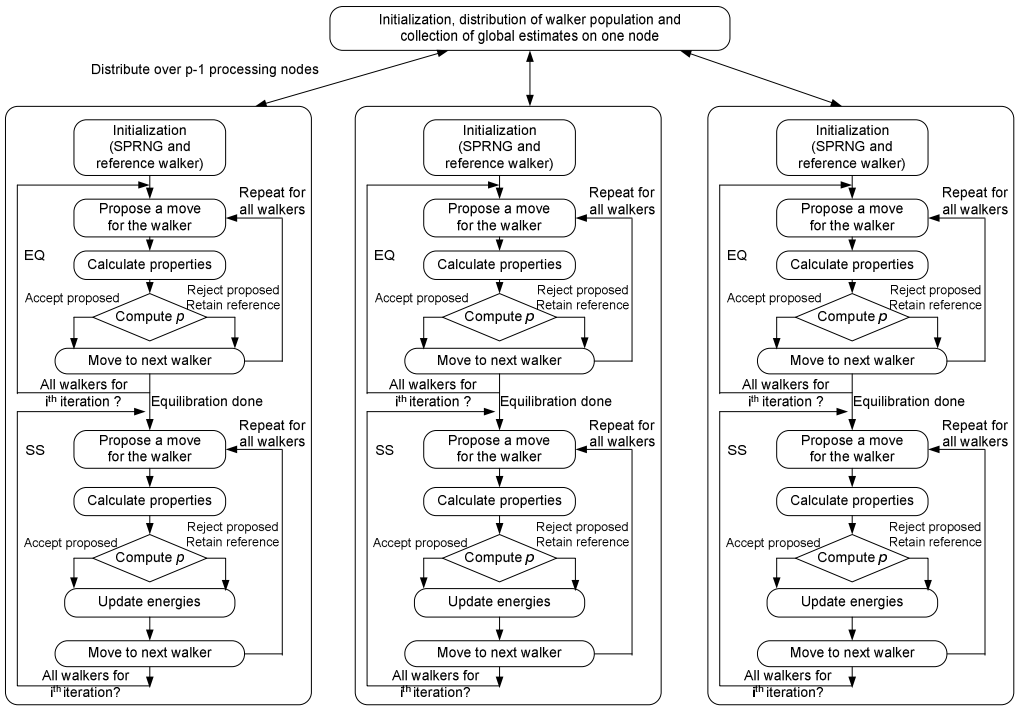

Figure 3.2: Flow chart of parallel VMC algorithm

4. RECONFIGURABLE ARCHITECTURE

Recent advances in field-programmable gate array (FPGA) technology have made reconfigurable computing using FPGAs an attractive platform, for accelerating scientific computing applications, providing hardware-like performance and flexibility possible with software. This chapter provides an overview of reconfigurable computing, followed by the implementation details of the reconfigurable architecture to accelerate the computationally intensive kernels of the Quantum Monte Carlo simulations. We also describe the target Cray XD1 high performance reconfigurable computing platform.

4.1. Reconfigurable Computing

Reconfigurable Computing (RC) [8] is the combination of reconfigurable logic with a general-purpose microprocessor aimed at providing more performance than possible with software-only solutions on general-purpose processors and increased flexibility compared to an application specific integrated circuit (ASIC). Reconfigurable logic devices such as field-programmable gate arrays (FPGAs) consist of a matrix of logic blocks that can be configured to implement the required logic functions and connecting them using programmable routing. FPGAs use clock rates an order of magnitude slower than a processor, but provide faster execution and lower power dissipation. Figure 4.1 shows the internal components of an FPGA. The FPGA and the microprocessor in a RC system are commonly connected using interfaces such as PCI, VME, or HyperTransport. Present FPGAs have increased gate densities that have allowed the mapping of complex designs onto these devices. In addition to the logic elements and programmable interconnect, current FPGA vendors have provided embedded resources that are used to achieve high

performance for commonly used functions. For example, on the Xilinx FPGAs, these resources include hardware multipliers, on-chip memories known as Block RAMs, digital signal processing blocks, and even PowerPC processors [107]. The other leading manufacturer of programmable logic devices, Altera also provides the Stratix and Cyclone series of FPGAs, which have embedded memories and multipliers [108]. FPGAs have shown a lot of potential to accelerate some computationally intensive applications. In [21], the authors provide a survey on reconfigurable architectures and a discussion of their representative applications including signal and image processing, bioinformatics, and supercomputing.

4.2. High Performance Reconfigurable Computing (HPRC)

High Performance Reconfigurable Computing is the combination of a high performance computing (HPC) system with RC elements to provide increased performance and flexibility. The computing nodes of HPRC systems are connected using a high performance, low latency interconnection network with some or all nodes equipped with one or more reconfigurable elements. Using these systems, we can take advantage of *polygranular* parallelism, i.e., the fine-grained parallelism offered by reconfigurable computing along with the coarse-grained parallelism typically available using parallel computing. Our target system for the hardware accelerated QMC simulation is the Cray XD1, a HPRC platform which integrates FPGA coprocessors in their supercomputer [43]. The architectural details and the development environment of the Cray XD1 are discussed later in this chapter.

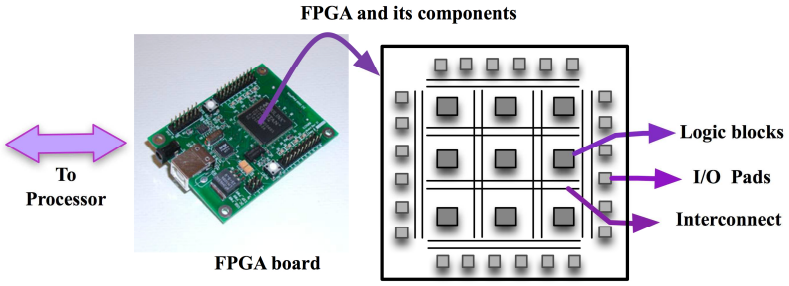

Figure 4.1: A reconfigurable computing system and FPGA components

4.3. Description of Kernels

The QMC application consists of the following functions which are computed between pairs of atoms in the system: distance calculation, potential energy, trial wave function, and derivatives of the wave function which are used to obtain the kinetic energy. The kinetic energy, potential energy, and wave function are functions of the squared distance. The numerical behavior of these functions requires us to use unique transformation schemes in order to restructure these kernel functions for efficiently mapping them onto the FPGA device. We describe the techniques used to transform the functions in the following subsections.

4.3.1. Potential Energy Calculation

The potential energy function of a cluster of *N* interacting atoms can be partitioned into a sum of two-, three- and many-body terms as in Equation 4.1.

$$V_{total} = \sum_{i<j} V_{i,j} + \sum_{i<j<k} V_{i,j,k} + \ldots \tag{4.1}$$

In Equation 4.1, the two-body potential, $V_{i,j}$, and the three-body potential, $V_{i,j,k}$ scale as O(N^2) and O(N^3) respectively. For large *N*, the number of higher terms becomes quickly unmanageable. Hence, in our work we ignore the contribution of the higher terms and assume a pair-wise model that includes only the two-body potential, which is often sufficient for modeling the desired chemical physics [109, 110]. Also, a fully pair-wise (two-body) model is a reasonable physical approximation in the study of weakly interacting atomic clusters. With this assumption, the total potential energy function in a cluster of *N* atoms, V_{total}, is approximated as a sum of $N(N-1)/2$ pair-wise contributions given by Equation 4.2.

$$V_{total} \approx \sum_{i<j} V(r_{ij}) \tag{4.2}$$

Figure 4.2 shows the interatomic potential as a function of the radial distance, r_{ij} between the atoms, *i* and *j*. The pair-wise potential energy function, $V(r_{ij})$, is characterized by an exponentially repulsive region at small values of r_{ij} and an attractive region at intermediate values of r_{ij} that reaches zero as $r_{ij} \to \infty$. We define the following parameters: ε, the depth of the well region, and σ, the cut-off value of r_{ij} where the potential function equals zero. The general shape of the potential energy function that Figure 4.2 depicts is universally applicable in describing non-Coulombic atomic or molecular interactions. However, slightly different potential energy functions are required depending on the exact chemical identities of the interacting atoms. Two problems posed by the potential energy function require us to use special techniques to transform the functions for the FPGA implementation.

First, the potential energy is a function of the distance, r_{ij}. At first glance, this calls for the instantiation of an expensive square root core on the FPGA device. However, if we rewrite each pair-wise potential $V(r_{ij})$ as a function of r_{ij}^2, we can eliminate the need for the square-root operation following the calculation of the squared distance values. This is largely a cosmetic distinction as all we have done is shift the burden of taking the square root of r_{ij}^2 inside the potential function. This is advantageous in conjunction with a spline-based evaluation method since we can effectively pre-compute the square root by building it into the supplied coefficients.

Second, we can observe the problematic range and domain of this functional shape. The potential has a finite domain and infinite range until it reaches a zero value, at the point $r_{ij}^2 = \sigma^2$, and an infinite domain and a finite range thereafter. We divide these regions with non-identical numerical behavior into regions I and II.

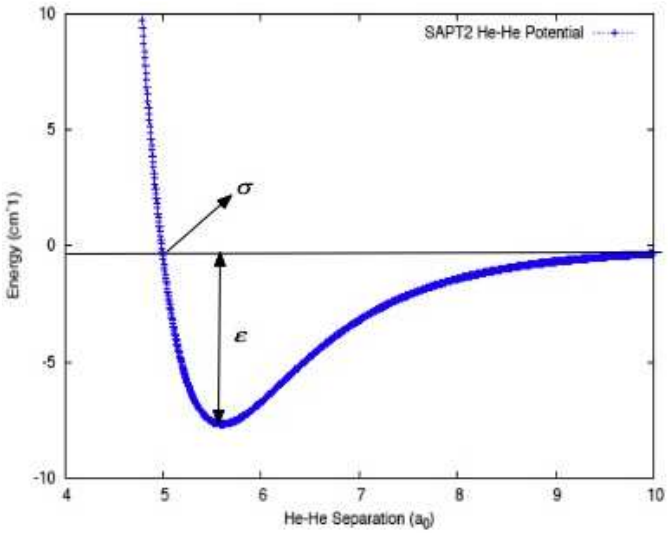

Figure 4.2: Potential energy as a function of distance

The original expression of Equation 4.2 for the total potential is now rewritten as a sum of two terms given by Equation 4.3. Region I is defined on the domain, $0 \le r_{ij}^2 < \sigma^2$, and within region I, $V_I(r_{ij}^2)$ is positive, taking on values from zero to positive infinity. This dynamic range is clearly undesirable as we implement all the operations using fixed-point due to space limitations on our current FPGA device. Region II of the potential is defined on the region $r_{ij}^2 \ge \sigma^2$ and ranges in value from $-\varepsilon$ to zero. The finite range of the function in region II is an attractive property as it bounds the sum in Equation 4.2, albeit the infinite domain remains a problem.

$$V_{total} = V_I + V_{II} = \sum_{(i<j)\in I} V_I(r_{ij}^2) + \sum_{(i<j)\in II} V_{II}(r_{ij}^2) \tag{4.3}$$

The exponential transform used in region I is given by Equation 4.4. This transformation provides several advantageous properties. The transformed region I potential, $V_I^{'}$ is restricted to take values between zero and one, inclusive. The sum of the pair-wise potentials for region I (the first term in Equation 4.3) can now be expressed as a product of the transformed pair-wise potentials (Equation 4.5). The relationship

between V_I and $V_I^{'}$ is now given by Equation 4.6. Another advantage here is that the expensive final transformation involving the natural logarithm performed once can now be delegated to the host processor.

$$V_I^{'}(r_{ij}^2) = e^{-V_I(r_{ij}^2)} \tag{4.4}$$

$$V_I^{'} = \prod_{(i<j)\in I} V_I^{'}(r_{ij}^2) \tag{4.5}$$

$$-\ln V_I^{'} = V_I = \sum_{(i<j)\in I} V_I(r_{ij}^2) \tag{4.6}$$

The transformed potential in region I is evaluated using polynomial interpolation. A lookup table contains the interpolation coefficients at uniformly spaced intervals from $r_{ij}^2 = 0$ to $r_{ij}^2 = \sigma^2$.

An approximate logarithmic binning scheme is used to cope with infinite domain in region II. We first introduce a cutoff at large distance that coincides with the largest value of r_{ij}^2 allowed by its fixed-precision format. Next, we partition the whole region into smaller regions such that the end points of each region correspond to consecutive powers of two. Thus, the size of each sub-region will increase stepwise with r_{ij}^2. This partitioning takes advantage of the fact that the curvature of the potential is largest at smaller values of r_{ij}^2 and asymptotically approaches zero or flattens out at large values of r_{ij}^2. Finally, we partition each sub-region into several intervals of equal size. Polynomial interpolation can be used to evaluate the potential within the intervals in region II. This scheme allows us to effectively take advantage of a logarithmic transformation of the coordinate r_{ij}^2 without the need to compute base two logarithms to determine the correct set of coefficients for interpolation. The exponential transform in region I automatically ensures that the potential takes values between zero and one, inclusive. We also rescale the region II potential by a factor of $-1/\varepsilon$ so that it always takes a value between zero and one.

Figure 4.3 shows the plot of *SAPT2* helium-helium potential energy versus distance, showing the two regions, region I, where $r < \sigma$, and region II, where $r > \sigma$. Applying the exponential transform in region I rescales the potential energy to lie between 0 and 1. The potential energy in region II is also rescaled to lie between 0 and 1. Figure 4.4 shows the rescaled and transformed helium-helium potential (using double-precision and fixed-point in software) as a function of the co-ordinate distance between the atoms.

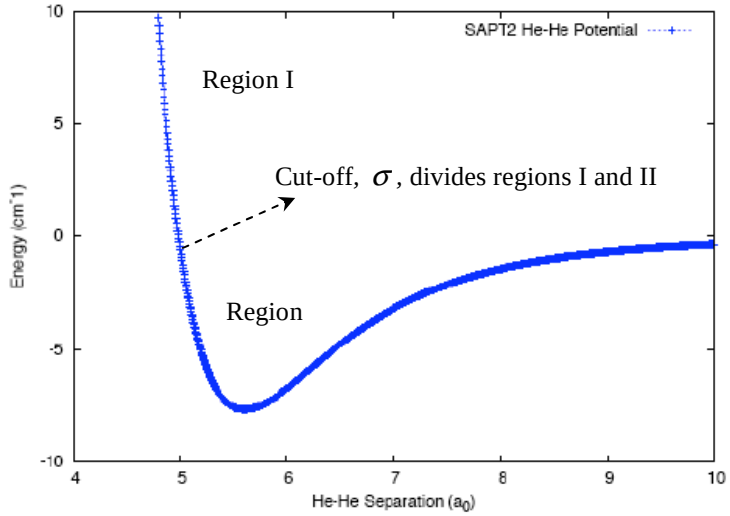

Figure 4.3: Plot of helium-helium potential versus distance

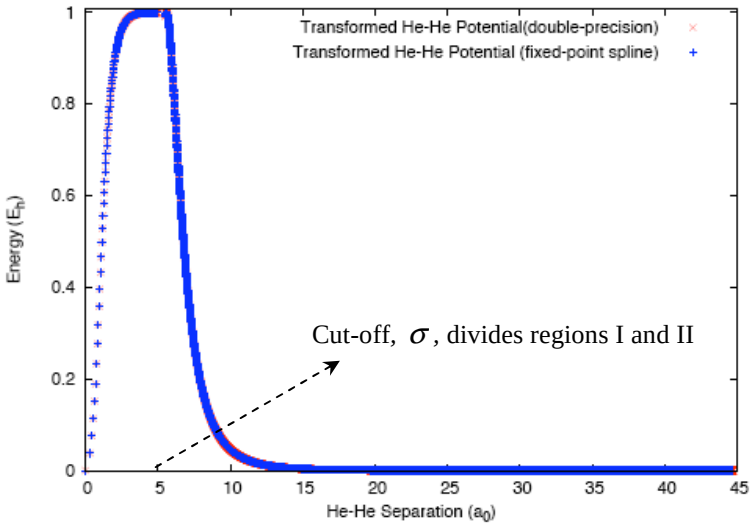

Figure 4.4: Plot of helium-helium potential versus distance (after rescaling and transformation)

4.3.2. Wave Function Calculation

The wave function is generally taken as the product of one-body (T_1), two-body (T_2), and three-body (T_3) terms as given in Equation 4.7. Figure 4.5 shows the general-shape of the wave function applicable for atomic and molecular clusters.

$$\psi = \prod_{i} T_1(r_i) \prod_{i<j} T_2(r_{ij}) \prod_{i<j<k} T_3(r_{ij}, r_{ik}, r_{jk}) \tag{4.7}$$

We can observe that the many-body effects also result in higher-order terms here as with the case of potential energy function. For our purposes, we ignore the one-body and three-body correlation functions and work with the two-body interactions to evaluate the wave function [41]. The one-body terms are unnecessary since they may be represented by the pair-wise terms. The wave function requires no transformation techniques. However, we rescale the wave function so that the maximum is less than one. We use a similar region classification approach for the wave function consisting of regions I and II. The cut-off value, σ, is set to the distance where the wave function is maximum. This value, obtained by setting its first derivative to zero, serves as a good dividing point so that we can use the same region classification approach used for the potential energy and yet accurately approximate the function. A quadratic polynomial interpolation is performed on this rescaled wave function. The advantage of this approach lies in the fact that we can re-use the same hardware developed for the potential energy calculations for the wave function. This significantly reduces our design time as we now only need to calculate new constants and a different set of interpolation coefficients. The two regions, region I, where $r < \sigma$, and region II, where $r \geq \sigma$ are also shown in Figure 4.5. Figure 4.6 shows the rescaled and wave function (using double-precision and fixed-point in software) as a function of the co-ordinate distance between the atoms.

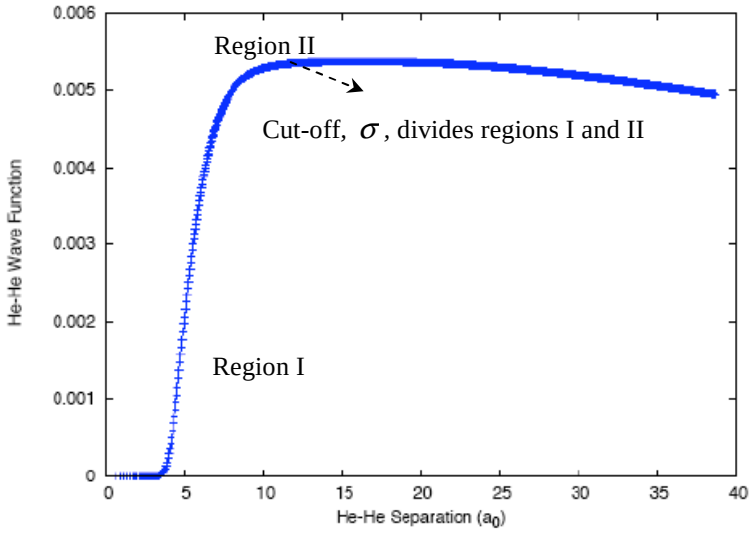

Figure 4.5: Plot of helium-helium wave function versus distance

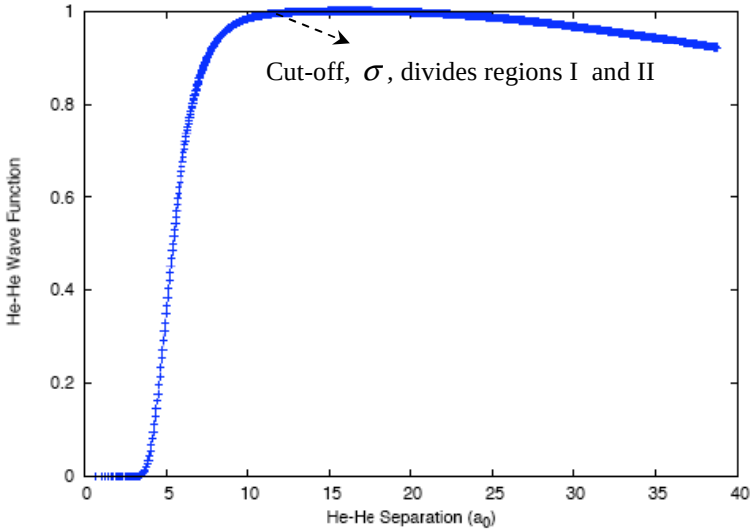

Figure 4.6: Plot of helium-helium wave function versus distance (after rescaling)

4.4. Top-Level Block Diagram

Figure 4.7 shows the top-level block diagram of the pipelined and parallel architecture developed to target a RC platform. The architecture consists of the following components: *CalcDist* is the distance calculation module that computes the squared distances between the pairs of atoms, and the *CalcFunc* module is a generic function evaluation module that uses an interpolation method to calculate the potential energy or wave function which are re-written as functions of squared distances as described earlier. We use two instances of the *CalcFunc* module, *CalcPE* to calculate the potential energy, and *CalcWF* to obtain the wave function. *AccFunc* is a generic module that accumulates the results from *CalcFunc*. We use two instances of this module, *AccPE* for potential energy, and *AccWF* to accumulate the pair-wise wave functions.

In addition to the above components, look-up tables to store the (*a*, *b*, *c*) interpolation coefficients for function evaluation and the (*x*, *y*, *z*) co-ordinate positions of atoms are implemented using on-chip Block RAMs. A state machine controller is used to generate the addresses to the Block RAMs. The *CalcBin* module is used to compute the bin address, which is used to fetch the interpolation coefficients. This module also produces a *delta* value, which is used along with the interpolation coefficients by the *CalcFunc* modules. The components are deeply pipelined and produce a result every clock cycle.

Figure 4.8 shows the data movement in the QMC application. Understanding the data movement and the computational complexity of the components of the application will help us partition the tasks between the processor and FPGA in an RC platform. We have O(*N*) co-ordinate positions to be moved from the processor to the FPGA for every iteration of the QMC algorithm. The distance calculation, potential energy, and wave function are O(N^2) in the number of atoms. Hence, these are chosen for the FPGA implementation. From the description of the kernel functions in this chapter, we might recall that the total

potential energy of a system of N atoms is the summation of the $N(N\text{-}1)/2$ pair-wise potential energy contributions. Similarly, the wave function is the product of the $N(N\text{-}1)/2$ pair-wise wave function contributions. If our RC platform consists of a high-speed interconnect between the processor and the FPGA so we can keep the cost of data movement low, our architecture is designed in such a way to provide significant speedups over the QMC application that runs entirely on the host processor. To accomplish this, we use novel techniques and a fixed-point representation to implement the deeply pipelined and parallel architecture.

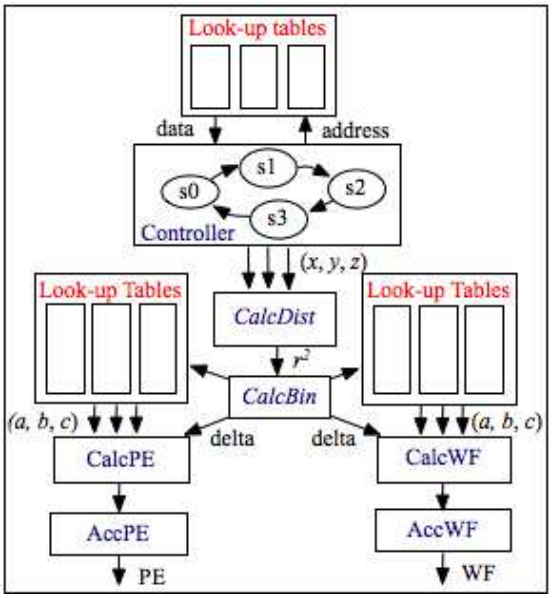

Figure 4.7: Top-Level block diagram of the pipelined architecture

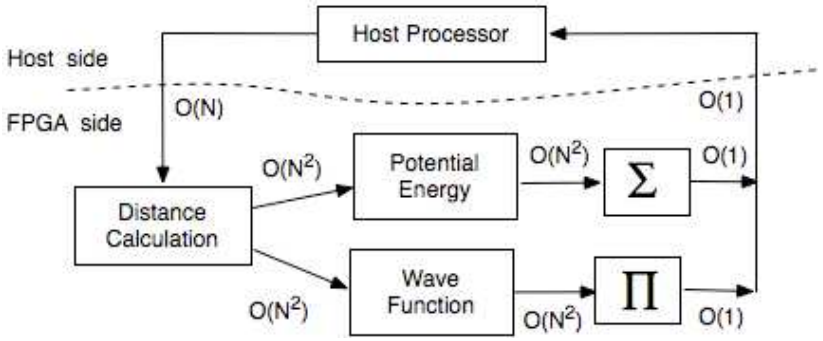

Figure 4.8: Data movement in the QMC application

4.4.1. Memory Platform

The 18-kbit embedded Block RAMs (BRAMs) available on the Xilinx Virtex-4 FPGA are used to store the various parameters needed by our system. We use the BRAMs to store the co-ordinate positions and the interpolation coefficients for the function evaluation. From our initial experiments varying the order of interpolation, we infer that the use of quadratic interpolation provides the required accuracy with modest use of BRAM resources. The BRAMs used to store the co-ordinate positions and interpolation coefficients are dual-ported and instantiated using the Xilinx Core Generator tools [111]. The BRAMs that store the 32-bit (x, y, z) positions are collectively referred to as *Position Memory*. The BRAMs that store the co-ordinate positions are presently designed to support clusters up to 4096 atoms. On larger FPGAs with increased BRAM resources, we would only need to regenerate the *Position Memory* using Xilinx Core Generator tools.

PE Coefficient Memory and *WF Coefficient Memory* store the (a, b, c) interpolation coefficients for regions I and II, for potential energy calculation, and wave function calculations, respectively. The lookup tables for each function store 256 (a, b, c) 52-bit coefficients for region I and 1344 (a, b, c) coefficients for region II. The BRAMs to store the region I and region II coefficients are configured to store 64-bit coefficients with a depth of 256 and 2048 respectively. Figure 4.9 shows the memory platform consisting

of the *Position Memory*, *PE Coefficient Memory*, and *WF Coefficient Memory*. (*a1*, *b1*, *c1*) and (*a2*, *b2*, *c2*) denote the coefficients for quadratic interpolation for regions I and II of the two functions. Ports A (*addr, din*) are used to write the co-ordinate positions and Ports B (*addr, dout*) are used to read the positions.

Ports A and B have a read pipeline latency of one clock cycle. In an RC platform, the host processor would load the co-ordinate positions to the *Position Memory* and the interpolation coefficients to the respective coefficient memories. The BRAMs that store the interpolation coefficients are initialized once prior to the beginning of pipeline operation and only receive read requests from the module. However, a change in co-ordinate positions that relates to the physical movement of the atoms during the simulation requires us to load new positions to the *Position Memory* every iteration. The state machine transitions from one state to another to generate the read addresses for the *Position Memory*. The order in which the *N* x *N* matrix is traversed and the addresses are generated to read the *Position Memory* is shown in Figure 4.10. The energies and wave functions are calculated due to the atom-atom interactions in the shaded portion (upper triangular part) of the matrix shown in Figure 4.10.

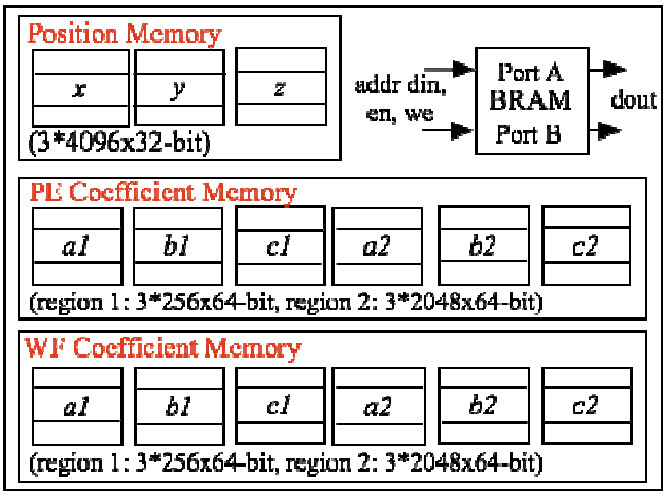

Figure 4.9: Memory platform

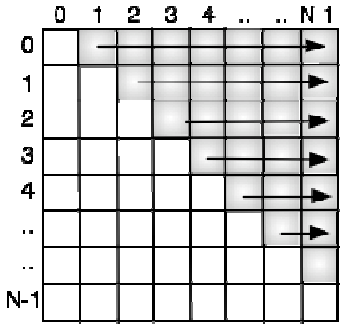

Figure 4.10: State machine generates addresses as shown to read *Position Memory*

4.4.2. Binning Scheme

We now discuss the binning schemes employed in the two regions of potential energy and wave function evaluation. The schemes to look up the interpolation coefficients are different for the two regions due to their non-identical numerical behavior. Region I is approximated using 256 bins. Determining the interpolation constants for region I is straightforward and a single stage lookup is sufficient to lookup the constants from a table of coefficients at uniformly spaced intervals ranging from $r_{ij}^2 = 0$ to $r_{ij}^2 = \sigma^2$. The width of each bin is used to choose from the 256 values corresponding to the squared distances. For the FPGA implementation, we store the inverse of the bin width in a register. The bin lookup scheme for region I is shown in Figure 4.11. The lower 8-bits of the integer portion of the product of the squared distance and inverse of the bin width form the address that is used to fetch the interpolation coefficients from the memory.

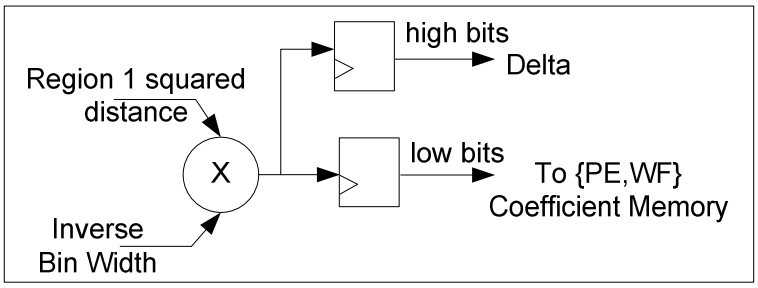

Figure 4.11: Bin lookup scheme for region I

We employ a logarithmic binning scheme in order to represent region II accurately. We divide region II into 21 regimes. Each regime is divided into 64 bins for a total of 1344 coefficients. Hence for regions I and II, we have a total of (256 + 21*64) * 3 coefficients for quadratic interpolation. A two-stage lookup procedure is used, first to determine the regime by performing a leading zero count and then determining the set of interpolation coefficients. The reciprocal of the bin widths is stored, eliminating the need for a division operation. Figure 4.12 shows the block diagram of the first stage of the lookup scheme for region II. The difference between the squared distances and the σ^2 value is used by the leading zero count detector (LZCD) to compute the index of the regime. The LZCD logic is implemented using a set of three priority encoders (Pr1, Pr2, Pr3). Figure 4.13 shows the second stage of the lookup scheme to obtain the actual set of interpolation coefficients after the regime lookup is complete. We compute the bin location for region II, using additional constants, which are obtained from memory addressed using the regime. We can use the computed address to retrieve the coefficients for region II.

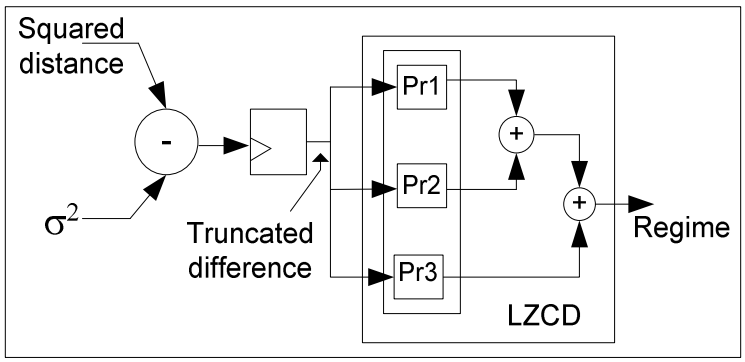

Figure 4.12: Lookup scheme for region II (1st stage)

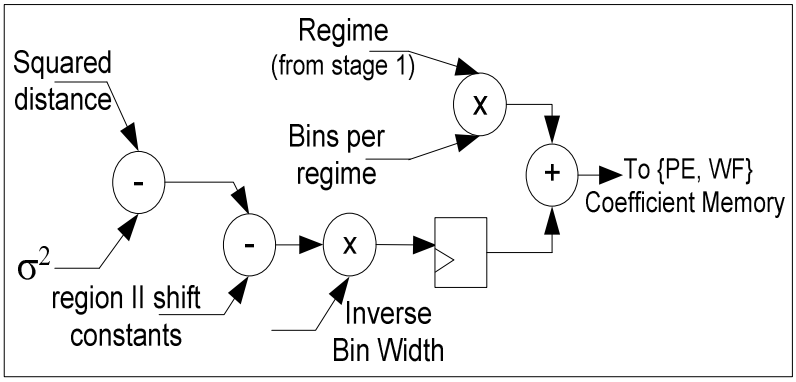

Figure 4.13: Lookup scheme for region II (2nd stage)

4.4.3. Calculation of Squared Distance

The data path of the *CalcDist* block is shown in Figure 4.14. The *CalcDist* block calculates the squared distances between pairs of atoms. In an RC platform, the O(*N*) co-ordinate positions are transferred from the host processor to the on-chip *Position Memory*. The state machine provides the read addresses to the *Position Memory* to read the pair positions, (x_i, y_i, z_i) and (x_j, y_j, z_j), every clock cycle and provide them to the *CalcDist* module. The co-ordinate distances between each atom and other atoms are obtained. Since $r_{ij} = r_{ji}$ and $r_{ij} = 0$ for $i = j$, we only need to calculate $N(N-1)/2$ squared distances (upper or lower triangular portion of the matrix in Figure 4.10). We use the Xilinx IP cores from Core Generator library to implement functions such as addition, subtraction, and multiplication. The cores are provided with a variety of pipelining options, variable input-output data-widths, and customized for the target Xilinx FPGA architecture. We use the multipliers with maximum pipelining, which corresponds to a latency of seven clock cycles. The initial latency of this module is ten clock cycles to fill the pipeline after which it produces a squared distance every clock cycle. Since we obtain a result every clock cycle, it takes $10 + N(N-1)/2$ clock cycles to obtain all the co-ordinate distances. However, since the entire architecture is pipelined, the calculation of the pair-wise distances is overlapped with the function evaluation.

Table 4.1 shows the data widths and latencies of the components of this module. The data widths are chosen after analyzing the errors with various fixed-point representations. This module uses a 32-bit fixed-point representation for the positions and produces a 53-bit squared distance value per clock cycle. The resulting squared distances are compared with σ^2 and classified as region I or region II values for the potential energy and wave function evaluations.

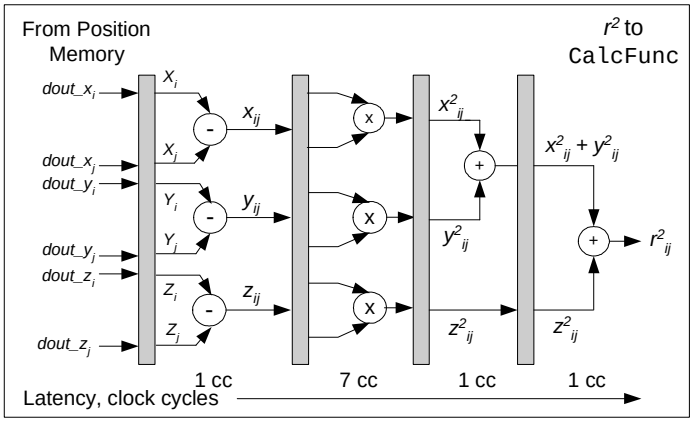

Figure 4.14: Data path of the distance calculation module (*CalcDist*)

Table 4.1: Data widths and latencies of *CalcDist*

Signal/Core	Data Widths	Latency (clock cycles)
Input	32-bit (signed 12.20)	--
Subtractor	32-bit i/p, 33-bit o/p	1
Multiplier	33-bit i/p, 66-bit o/p	7
Adder1	65-bit i/p, 66-bit o/p	1
Adder2	66-bit i/p, 67-bit o/p	1
Output	53-bit (unsigned 27.26)	--

4.4.4. Calculation of Potential Energy/Wave Function

Figure 4.15 shows the data path of the *CalcFunc* pipeline. This generic pipeline is built to compute either the potential energy or wave function using polynomial interpolation. In our implementation, we instantiate two copies of this module, one for potential energy called *CalcPE* and the other for wave function, namely *CalcWF*. The two blocks, *CalcPE* and *CalcWF*, concurrently process the squared distance values from the *CalcDist* module every clock cycle. Depending on whether the squared distance falls in region I or region II, the lookup schemes discussed earlier are used to fetch the interpolation coefficients for each function. A delta value pertinent to the squared distances is also used as an input to this block. These are processed by the *CalcFunc* module to produce a result every clock cycle once the pipeline is full. The adders and multipliers are instantiated from the IP cores provided by the Xilinx Core Generator library. The latencies of the multipliers and adders are shown in Figure 4.15. The multipliers use maximum pipelining with a latency of seven clock cycles. This module produces a 52-bit potential energy or wave function value every clock cycle. Table 4.2 shows the data widths and latencies of the components of this module.

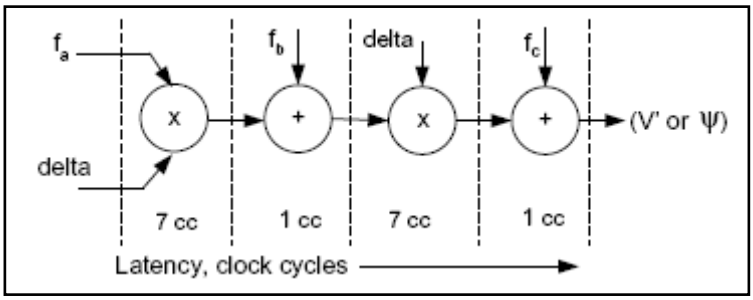

Figure 4.15: Data path of the function calculation module (*CalcFunc*)

Table 4.2: Data widths and latencies of *CalcFunc* module

Signal/ Core	*Data Widths*	*Latency* (clock cycles)
Inputs	Interpolation coefficients (*a*, *b*, *c*): signed 0.52, delta: signed 0.52	--
Multiplier1	52-bit i/p, 104 bit o/p	7
Adder1	52-bit i/p, 52-bit o/p	1
Multiplier2	52-bit i/p, 104-bit o/p	7
Adder2	52-bit i/p, 52-bit o/p	1
Output	Pair-wise Potential Energy /Wave Function: signed 0.52	--

4.4.5. Accumulation of Potential Energy/Wave Function

The *AccFunc* module is used to accumulate the energies and wave functions produced by the *CalcFunc* pipeline. We instantiate two copies of the module: *AccPE* for the potential energy and *AccWF* for the wave function. Figure 4.16 shows the *AccPE* module that accumulates the intermediate values of potential energy. Since we interpolate the transformed potential, we accumulate the potential energies resulting from region I distances as running products and the energies resulting from region II distances as a running sum. Figure 4.17 shows the *AccWF* module that accumulates the pair-wise wave functions. We may recall that the wave function did not undergo any transformation prior to interpolation. Due to the functional form of the wave function as described earlier, we form products of the pair-wise wave functions. We accumulate all the wave functions resulting from region I and region II distances as running products.

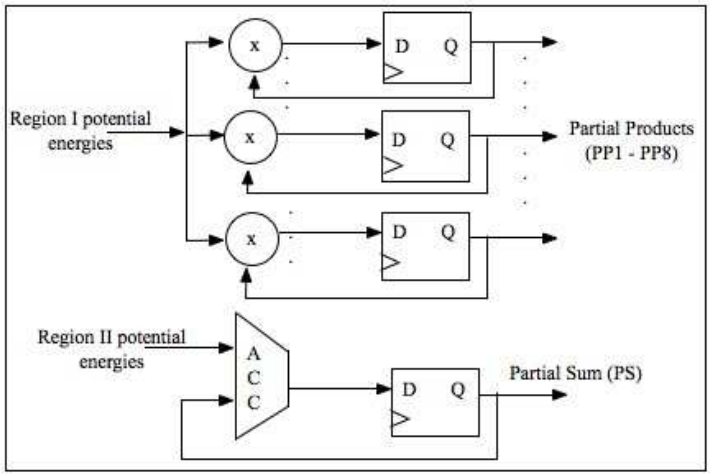

Figure 4.16: Potential energy accumulation (*AccPE*)

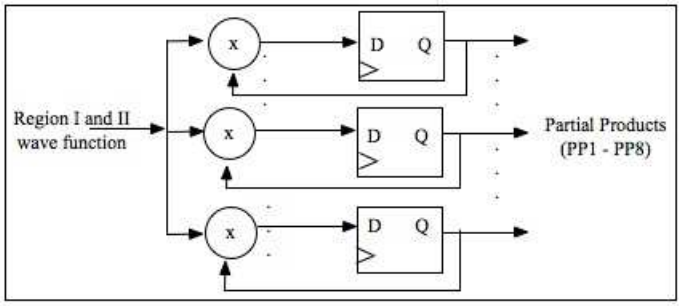

Figure 4.17: Wave function accumulation (*AccWF*)

The region II accumulator of *AccPE* is instantiated from the Xilinx Core Generator library, and with registered outputs, the latency of the accumulator is two clock cycles. The *CalcPE* module is pipelined and outputs a region I or region II potential energy result every clock cycle. Hence, the region II accumulator adds a region II potential if applicable or a zero if a region I potential is produced that clock cycle. The region I accumulator of *AccPE* and the *AccWF* are implemented using the multipliers from Xilinx Core Generator. For accumulating the region I potential energies, we multiply a region I potential if applicable or a one if a region II potential is produced in that clock cycle. With registered outputs, the multipliers have a latency of eight clock cycles. We use eight instances of the multipliers and switch between the multipliers to increase the data rate and in order to keep the pipeline busy. The *AccPE* module produces a single partial sum (PS) from region II and eight partial products (PP1 – PP8) from region I. The *AccWF* module produces eight partial products of the wave function (PP1 – PP8).

Some important concerns that are taken into account while designing these accumulators are discussed here. We perform successive multiplications of the region I potential energy values and the region I and II wave functions. The distances we sample in the QMC simulation are such that most of the potential energy and wave function values are close to one. Hence, repeated multiplication of these values during accumulation results in products that tend towards zero. In a fixed-precision register, the appearance of leading zeros results in a loss of precision. To guarantee that we do not lose precision during accumulation, we introduce a bit shift to the left after computing each product (if it is less than 2^{-1}) and incrementing an initially denormalized exponent. Each partial product (PP_n) is associated with a shift count. There are also overflow issues associated with the accumulator that accumulates the region II potential values. The region II potentials are accumulated in a register of fixed-precision large enough to hold *N* evaluations of the maximum value (1.0) of the re-scaled potential. The products along with the shift counts are delivered to the host processor, which removes the scaling and combines the results from region I and II to reconstruct the floating-point value of the total potential energy and wave function.

In the following sub sections, we provide an overview of the Cray XD1 architecture and describe the integration of the QMC design modules with the rest of the Cray XD1 platform.

4.5. Cray XD1 Architecture

The Cray XD1 supercomputer is an interesting platform for our design as it incorporates application acceleration processors made of FPGAs in its computing nodes [43]. It also provides a high-bandwidth, low-latency interconnect between the processor and the FPGAs. The architectural unit of the Cray XD1 is a chassis, which consists of up to six compute blades. The Cray XD1 chassis is shown in Figure 4.18. Each compute blade consists of two 64-bit AMD Opteron processors configured as six two-way symmetric multiprocessors (SMPs). A maximum of up to twelve RapidArray Processors (RAPs) are also present to process communications within the chassis. The chassis also contains the application acceleration system with six optional FPGAs, which are tightly coupled to the Opteron's address space and serve as coprocessors to the Opteron processors. A Cray XD1 system can contain one to hundreds of chassis. A compute blade also consists of 1 to 8 GB double data rate synchronous dynamic random access memory (DDR SDRAM) per compute processor.

The FPGA is present on the expansion module that optionally connects to each compute blade. The expansion module also contains an additional RapidArray processor providing two additional RapidArray links per compute blade. The RapidArray processor provides the interface for the FPGA to connect to the Opteron processors as well as to the high-speed RapidArray switch fabric. The RapidArray interconnect, which links the processors and memory within and between chassis, overcomes the PCI bus bottlenecks with some previous RC systems. In addition to the main memory and the FPGA on-chip memory, four quad data rate (QDR) static random access memory (SRAM) banks provide local high-speed storage for the FPGA. A programmable clock source provides the clock for the FPGA design. Figure 4.19 shows the logical view of the expansion module.

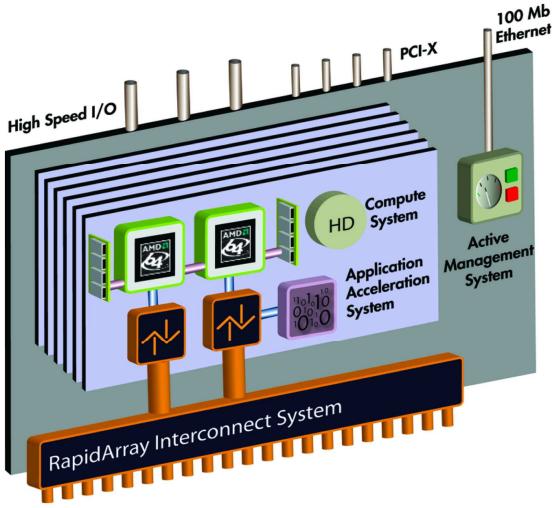

Figure 4.18: Cray XD1 system chassis [43]

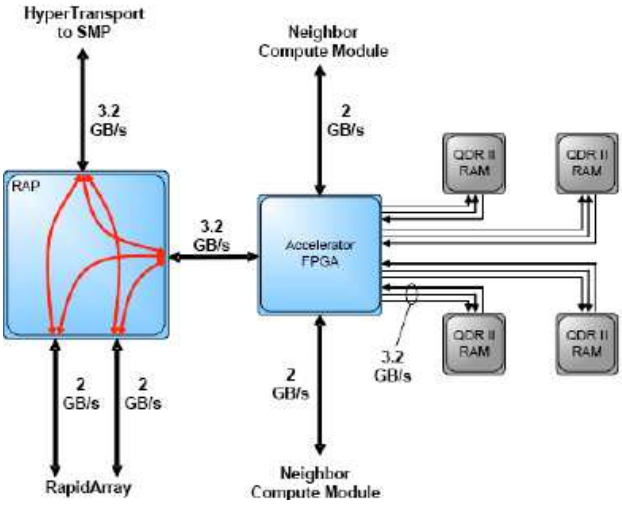

Figure 4.19: Components of the expansion module [43]

4.6. Development Environment

Cray provides IP cores that enable the user design modules to interface with the rest of the Cray system. The Cray QDR IP core is used to interface the FPGA design with the external QDR RAMs on the expansion module. The Cray RapidArray Transport (RT) core provides the interface between the RapidArray fabric and the FPGA. The RT core provides two interfaces: the fabric request interface that issues read and write requests to the user logic when it receives data from the node, and the user request interface which communicates the read and write requests originating from the user logic to the processor. The RT core is a 64-bit interface at a maximum speed of 200 MHz and yields a bandwidth of 1.6 Gbytes/sec for simultaneous transmit and receive operations.

The RapidArray processor connects the FPGA to the processor's Hypertransport link; hence the FPGA is accessible via a 128 Mbytes window of the Hypertransport I/O address space. Hypertransport read and write requests from the processor are passed to the user logic through the RT interface. Cray provides the application acceleration API functions, which allow an application running on the Opteron to communicate with the FPGA design. Invoking the API functions initiates the transactions on the RT fabric, which are delivered to the RT core to be processed by the user logic module that is connected to it. The FPGA control utility, which is part of the Cray development environment, also allows a user to interact with the FPGA and perform functions such as resetting or configuring the FPGA.

Figure 4.20 shows the design structure on the FPGA. A top-level VHDL wrapper code provided by Cray integrates the various logic components: the RT core, QDR core, programmable clock generator, and the user application. The user design logic is contained within the user application component. The standard design flow processes and tools are used to target the user design to the application acceleration processors on the Cray XD1. The design can be described using Verilog, VHDL, or a language that can

finally be converted to a Xilinx netlist. A design flow (one used in this work) is illustrated in Figure 4.21. The design modules are described in VHDL. In addition to the hand-written VHDL design modules, a number of parameterized logic blocks from Xilinx Core Generator are used. The process of synthesis converts the HDL code to a gate-level netlist. This is done using the Xilinx XST tools [111]. The implementation process consists of translating, mapping, placing and routing, and generating the binary file. An optional user constraint file (UCF) also allows us to specify timing and placement constraints for the implementation process. In addition to the above steps, simulation is also done using Modelsim on the design code or synthesized netlist. The last step performed using the FPGA control utility (shown in the dotted box) converts the binary file (.bin file) obtained at the end of the implementation process to a Cray-proprietary format containing the FPGA part information and clock frequency.

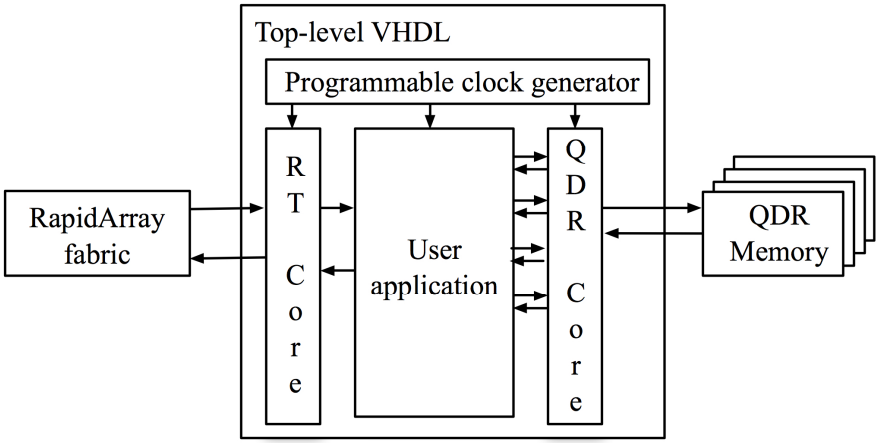

Figure 4.20: Cray XD1 design structure [43]

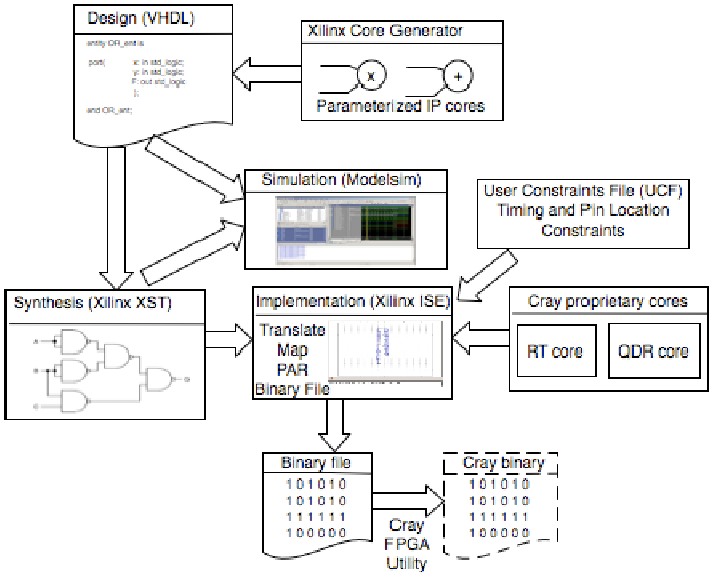

Figure 4.21: Design flow used in this work

4.7. Quantum Monte Carlo targeted to Cray XD1

Figure 4.22 shows the block diagram of the design implemented on the Cray XD1 platform. The architecture consists of the following components: RT Client, Block RAM Interface, Register Interface, and QMC Interface. The QDR II RAMs are not used in the current design and disabled to save power and resources. The RT Client block connects directly to the RT Core, which processes read and write requests to and from the Opteron processor. When an application on the Opteron processor opens and access the FPGA, the accesses are translated into read and write requests, which are forwarded through the RapidArray fabric to the RT Core block. The RT Client block, which is directly connected to RT Core, is comprised of a state machine that forwards the fabric requests to the Register Interface or the Block RAM Interface and accepts a response from these interfaces.

The Register Interface block provides a set of readable and writable registers, which can be used as configuration and status registers in any Accelerator FPGA design. It also contains the decoding and multiplexing logic to read and write the registers. In our design, the registers are used for the following

functions: (a) to communicate control signals from the FPGA to the processor and vice versa and (b) to send results from the FPGA to the processor. The Block RAM Interface provides read and write access for a processor to the Xilinx Block RAMs. The interface consists of RAM control logic, which uses the fabric request, and the control signals in the registers to generate the "write enable" signals for the respective Block RAMs. Using these interfaces, the user design can read the control signals and contents written to registers and Block RAMs by the processor and write results back to registers.

The block labeled QMC Interface connects the user design modules with the Cray platform-related interfaces described earlier. We use C for the host program on the Opteron and VHDL for the blocks implemented on the FPGA. The Potential Energy (PE) and Wave Function (WF) kernels are implemented on a single FPGA. The "write enable" signals from the Block RAM Interface are processed by the QMC Interface to initialize the *Position Memory* with co-ordinate positions from the processor and the PE and WF *Coefficient Memory* with interpolation coefficients. The Cray API functions, *fpga_wrt_appif_value*( ) and *fpga_rd_appif_value*( ), are used by the processor to access the FPGA's 128 Mbyte address region, blocks of which are allocated to the Block RAM and Register Interfaces. The *Position Memory* that stores the co-ordinate positions of the atoms and the *CalcDist* module are shared by the {PE/WF} engines. The results from the PE and WF calculation (*CalcFunc*, *AccFunc*) are written to the user-defined registers, which can be accessed by the Opteron processor through the RT Client block. The Opteron reads the register contents using the Cray API functions, rescales the fixed-point results back to their original values, and reconstructs the floating-point values of the functions. We discuss the results from the reconfigurable implementation in Chapter 6.

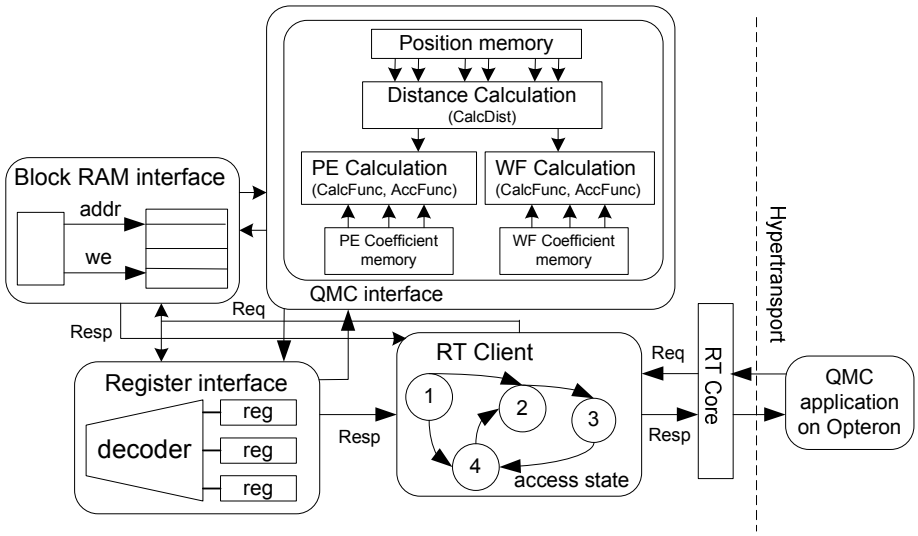

Figure 4.22: Cray XD1 design overview

5. GPU Implementation

The ever-growing demands for performance from the gaming industry have led to a tremendous growth and development of graphics processors. The huge amounts of memory bandwidth and the large number of processing cores on the graphics processors make them attractive for general-purpose tasks other than graphics processing. NVIDIA's GPU solutions presently provide hundreds of processing cores, tremendous on-chip memory bandwidth, and the CUDA programming paradigm that allows us to exploit the computational power of the GPU without the need to invoke graphics API functions. This chapter provides an overview of NVIDIA's Compute Unified Device Architecture (CUDA) paradigm and Tesla architecture. Following this, we discuss the implementation details of the Quantum Monte Carlo algorithm on the NVIDIA GPUs using CUDA.

5.1. Overview of CUDA

NVIDIA's CUDA paradigm allows programmers to use the C programming language with some extensions to develop general-purpose applications on their *Tesla*- architecture GPUs, including GeForce, Quadro, and Tesla products [60]. CUDA's scalable parallel programming model allows developers to transparently scale their applications to exploit parallelism on a large number of processor cores [112]. Recent work has demonstrated the viability of compiling CUDA programs for execution on multi-core CPUs [113]. There are three main abstractions in CUDA: a hierarchy of concurrent threads, software controlled scratchpad memory called shared memory, and barrier synchronization. These abstractions enable us to partition a problem into coarse-grained blocks that operate independently and into further fine-grained threads that work cooperatively in parallel. The serial program on the host processor invokes

a *kernel*, which is executed in parallel by a set of *threads*. A *thread block* refers to a group of concurrent threads that synchronize via barrier synchronization and co-operate and share data via a fast, shared memory space. A *computational grid* is a group of thread blocks that execute independently and in parallel. To invoke a CUDA kernel from the host program, we specify the name of the kernel followed by the *execution configuration* as shown below:

kernel<<<dimGrid, dimBlock>>>(parameters);

The execution configuration consists of the number of threads per block (**dimBlock**) and the number of blocks in the grid (**dimGrid)**. These block and grid dimensions are specified using the *dim3* three-element vector type which allows us to set the block and grid to be one-, two-, or three dimensions, using .x, .y, and .z fields. CUDA provides built-in variables to specify these parameters. **blockDim** and **gridDim** specify the number of threads per block and blocks per grid respectively. Within the CUDA kernel, block and thread ID variables, namely, **blockIdx** (labeled from 0 to **gridDim**-1) and **threadIdx** (labeled from 0 to **blockDim**-1) are used to specify the block number within a grid and the thread number within a block, respectively. Figure 5.1 shows the 1-D computational grid setup with 1-D thread blocks. In Figure 5.2, we show the computational grid set up for a 2-D grid consisting of 2-D thread blocks. Currently, CUDA supports a maximum of 512 threads in a thread block.

There are different levels of memory hierarchy on the GPU and threads may access data from these different levels during their execution. Each thread has a private local memory, used for variables that do not fit in the registers and for stack frames and register spilling [112]. Each thread block has a shared memory that is shared by all threads in the block and has the lifetime of the block. All threads have access to the global memory on the GPU. On the Tesla architecture, the shared and global memories are physically separate memories; the shared memory is a low-latency, on-chip RAM, the global memory is the DRAM on the GPU. The GPU has additional levels of memory called the texture memory and constant memory.

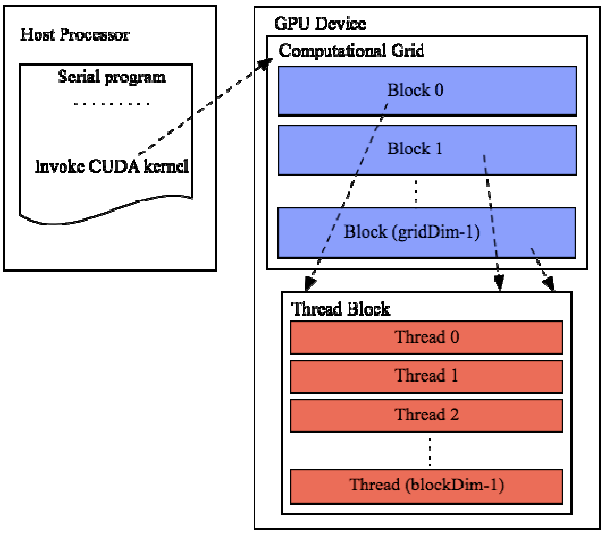

Figure 5.1: Computational grid (1-D grid and 1-D thread block) [60]

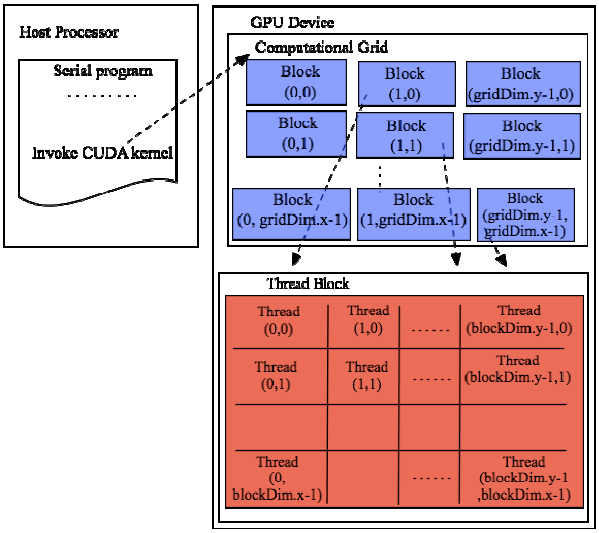

Figure 5.2: Computational grid (2-D grid and 2-D thread block) [60]

5.2. NVIDIA's Tesla Architecture

NVIDIA's Tesla architecture, first introduced in the GeForce 8800 GPU in November 2006, unifies the vertex and fragment processors present in graphics processors. The generality that came with this unified architecture, as vertex and fragment programs had to execute on this unified architecture, led to their entry into general-purpose computing. The Tesla architecture consists of an array of massively multithreaded streaming multiprocessors (SM) [114]. Each SM consists of eight scalar streaming processor (SP) cores, two special-function units (SFUs), which are used for transcendental functions, a multithreaded instruction unit, and 16KB shared memory. An instruction cache and a read-only constant cache are also present. The arrangement of the SP cores with the SMs in an independent processing unit is called a texture/processor cluster (TPC) and shown in Figure 5.3. The threads within a block execute concurrently on one SM. Each SP core contains a scalar multiply add (MAD) unit, for a total of 8 MAD units per SM. Each SFU also contains four floating-point multipliers. Tesla's multithreaded instruction unit called single-instruction, multiple-thread (SIMT) is responsible for creating, managing, scheduling, and executing threads in groups of 32 parallel threads called *warps*. The SM manages a pool of 24 warps, executing up to 768 (24*32) concurrent threads in hardware. A compute work distribution (CWD) unit is responsible for distributing the thread blocks to the SMs.

On the GeForce 8800 GPU, there are 16 SMs for a total of 128 processing cores. The SPs and the SFUs are clocked at 1.5 GHz, providing a peak performance of 36 GFlops per SM. The Tesla C870 GPU also contains 128 processing cores with 16 multiprocessors and 8 SPs per multiprocessor, clocked at 1.35 GHz. On the NVIDIA Tesla C1060 hardware, there are 30 streaming multiprocessors with eight scalar streaming processor cores and one double-precision core per SM, for a total of 240 single-precision processing cores and 30 double-precision cores on-chip.

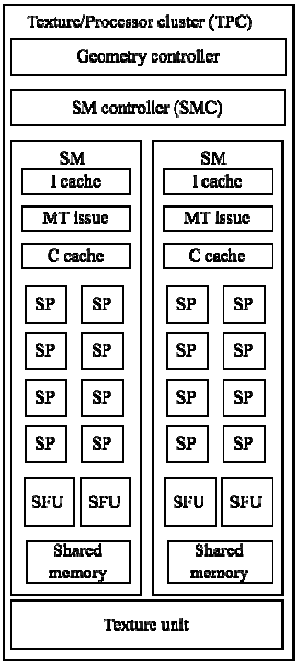

Figure 5.3: Texture/processor cluster architecture with SM and SP [114]

5.3. QMC Implementation

While porting the computationally intensive kernels of the QMC application to the GPU, we consider the following: (a) the partitioning approach, i.e., determining which functions are accelerated using the GPU and the parts of the QMC algorithm that will be retained on the CPU, (b) numerical precision that will be employed on the GPU for the function evaluations, and (c) the computational grid set up using CUDA, i.e. how to partition the computations so we keep the processors on the GPU busy.

Figure 5.4 shows the data movement for the GPU implementation. The O(N) co-ordinate positions are copied from the CPU main memory to the GPU main memory. The O(N) energies and wave function values are copied from the GPU memory to the CPU memory. On the host processor, we use the CUDA runtime functions to allocate memory and copy data from the host to the processor and vice versa. **cudaMalloc( )** and **cudaFree( )** are used to manage the global memory space visible to the kernels. The CUDA memory management function, **cudaMallocHost( )** allows us to allocate page-locked host memory,

which provides maximum PCI bandwidth for the memory transfers. The **cudaMemcpy( )** function is used to copy the data from the host memory to the device memory and vice versa. We also specify the execution configuration for CUDA, i.e., how the blocks and threads will be managed on the GPU. The computational grid set up for our implementation is the one shown in Figure 5.1, consisting of a 1D arrangement of blocks and threads. As mentioned earlier, the CUDA programming model is scalable and the program can be easily ported between different generations of GPUs with varying processor cores, by altering the number of blocks and threads while specifying the execution configuration.

We discuss two implementations: a naïve implementation and an optimized implementation depending on the number of computations performed in each case. Additionally, we also experiment with single- and double-precision as well as a combination of single- and double- precision for the evaluation of the kernel functions.

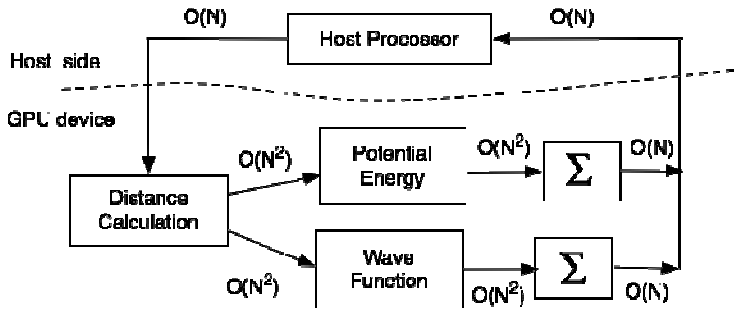

Figure 5.4: Data movement and data partitioning for the QMC application on the GPU

5.4. Approach

We consider a cluster of N interacting atoms, the interaction matrix for which is represented in Figure 5.5. The energies and wave function are functions of the co-ordinate distance, r_{ij} between pairs of atoms. Since $r_{ij} = r_{ji}$, we need to evaluate the lower (or upper) triangular portion of the interaction matrix, excluding the diagonal entries, which are zeros.

First, we discuss the naïve implementation in which we perform a brute-force evaluation of all N^2 interactions. We allocate a subset of the rows of the matrix, i.e., N/T to each block, b_i, where T is the number of threads in each block. Within each block, the T looping threads are used to obtain the interactions on an atom due to other atoms in the system. Thread t_i computes the interaction on atom 1 due to atoms 1 to N. This requires that each thread have access to the co-ordinate positions of N atoms. The GPU provides different levels of memory hierarchy: registers, shared memory, main memory, and texture memory. We investigate the use of different levels of memory. Initially we assume that the co-ordinate positions reside in the device memory. The threads access the positions from the device memory. With N looping threads, there are a total of $3N^2$ read accesses to the global memory to read the $3N$ co-ordinate positions and $2N$ write accesses to write the updated potential energy and wave function. The reading of co-ordinate positions from the GPU main memory by each thread is expensive (100-400 cycles latency). Hence, we modify the implementation to use the shared memory on the GPU. Simulation of a system with 4096 atoms would require 48 KB and 8192 atoms would require a 96 KB of shared memory. However, there is a limit of 16 KB shared memory per block. This means that we can store a maximum of 1364 atoms in the shared memory (for $3N$ 32-bit co-ordinate positions). In order to provide the capability to simulate larger clusters (greater than 1000 atoms), we would need to partition the accesses to the shared memory into a number of times steps, so we store only a subset of positions in a single time step. This is achieved by having each thread block load the co-ordinate positions of T atoms from the

main memory to the shared memory [115]. The accumulations of row-wise energies and wave function values are done by in-place reductions within the GPU kernel. The O(N) row-wise sums of energies and wave function values are transferred back to the host processor which performs the final summation of all O(N) to produce an O(1) result. Figure 5.6 shows the computational grid with N/T thread blocks. There are T looping threads in each block and N/T blocks for a total of N threads performing N calculations, for a total of N^2 interactions.

In the optimized implementation, we eliminate the need to evaluate the entire matrix. Figure 5.7 shows the optimized implementation in which only the lower triangular portion of the matrix (excluding the diagonal) is evaluated. This approach provides a significant speedup over the naïve implementation as shown by our results in Chapter 6 as it frees the GPU resources that are used for redundant computations. As in the naïve implementation, the O(N) partial sums of energies and wave function values are written to the device memory after each thread loops through the N atoms. The device memory is copied to the host memory using the CUDA runtime function. The host processor performs the final summation of all O(N) to produce an O(1) result. The impact of changing the block dimensions for different cluster sizes in both the naïve and optimized implementations are examined in Chapter 6. In the naïve and optimized implementation, we experiment with the numerical precisions used for the function evaluations and the accumulations. If we recall the Tesla architecture on the C1060 GPU, there are eight single-precision units and one double-precision unit. Hence, the single-precision performance is significantly greater than the double-precision peak performance. We analyze the numerical precisions while using entirely single-precision, double-precision, and a combination of single- and double- precision for our calculations. The speedup and the error performance of these implementations are summarized in Chapter 6.

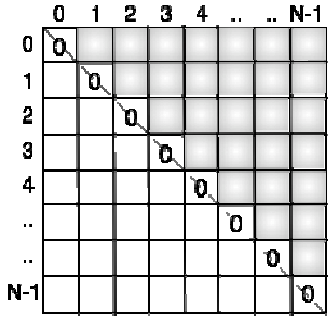

Figure 5.5: Interaction matrix

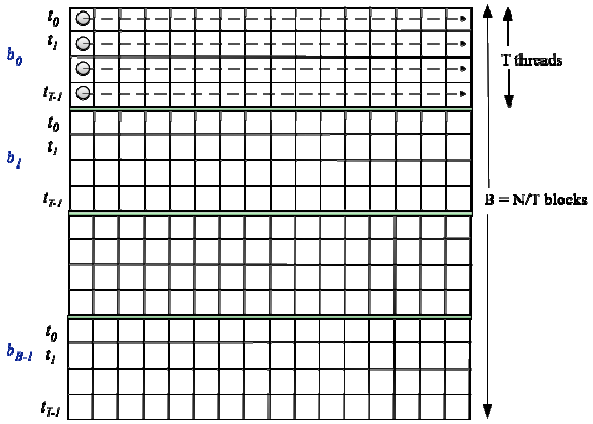

Figure 5.6: Computational grid for the QMC simulation in the naïve implementation

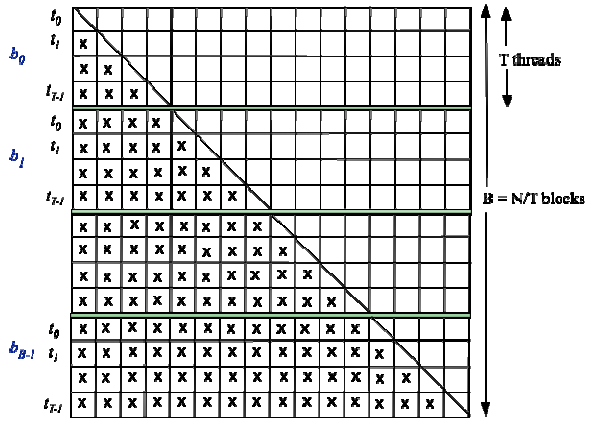

Figure 5.7: Computational grid for the QMC simulation in the optimized implementation

6. DISCUSSION OF RESULTS

In this chapter, we discuss the results from porting the Quantum Monte Carlo application onto the CPU, FPGA, and GPU platforms. We examine the speedup performance as the problem size is increased and the error performance for the numerical precision employed for the calculations while using each platform. We first provide an overview of the HPRC and GPU target platforms.

6.1. Target Platforms

6.1.1. HPRC Platform

We target the *Pacific* Cray XD1, a single chassis system consisting of six compute nodes, where each compute node consists of a dual-core dual-processor 2.2 GHz AMD Opteron with 8 GB local memory connected to Cray's proprietary RapidArray fabric. Each compute node consists of a Xilinx Virtex-4 VLX160 FPGA or a Virtex-II Pro VP50 FPGA. The compute node with the Virtex-4 FPGA is targeted in this work. An overview of the architectural details of the Cray XD1 has been presented in Chapter 4.

6.1.2. GPU Platforms

We target two generations of CUDA-compatible NVIDIA GPUs, C870 and C1060 Tesla. The C870 GPU contains 128 processing cores with 16 multiprocessors and 8 streaming processors per multiprocessor; clocked at 1.35 GHz. Three such GPUs are present on one of our target platforms (*Ed*) and connected via PCI-Express interface to a dual-core dual-processor AMD Opteron 1.8 GHz. On the NVIDIA Tesla C1060 hardware, there are 30 streaming multiprocessors with eight scalar streaming processor cores, for a

total of 240 single-precision processing cores. The processors are clocked at 1.3 GHz. Double-precision support is also added on the C1060. Two of these C1060 GPUs connect via PCI-Express 2.0 interfaces to a dual quad-core Intel i7 2.67 GHz that uses the Nehalem architecture (this platform is referred to as *Tesla* in this dissertation).

6.2. CPU Implementation

We consider the following scenarios for the CPU implementation. We implement the helium-helium potentials, *HFDB* and *SAPT2*, discussed in Chapter 2. We also experiment with double- and mixed-precision. First, we employ double-precision for the entire software implementation, for both function evaluations and accumulation of the results, and mixed-precision, where we employ single-precision for function evaluations and double-precision function accumulations. We repeat the experiments for various clusters from 256 to 8192 atoms (in powers of 2), to better understand how the application performs when increasing the problem sizes. Table 6.1 summarizes the configurations of the platforms, compiler settings, and optimizations used while benchmarking the application.

In Tables 6.2 (a), 6.3 (a), and 6.4 (a), we show the execution times from a single *compute*() function call that evaluates the *SAPT2* potential, *SAPT2* potential and trial wave function and the *SAPT2* potential, trial wave function, and kinetic energy, respectively, on a single core of the *Pacific*, *Tesla,* and *Ed* platforms. In this case, we employ double-precision. The slight differences in execution times between *Pacific* and *Ed* which are both AMD Opteron processors might be attributed to the compiler and architectural variations on these systems. *Tesla* uses the more recent Intel Nehalem architecture [116], which introduces various strategies to boost application performance. Of the three platforms, it exhibits a superior performance, running twice as fast as the other processors. In Tables 6.2 (b), 6.3 (b), and 6.4 (b) we show the execution times (in seconds) for a single *compute*() function call, when we repeat the experiments with the *HFDB* potential. The analytical functions for the wave function and hence, for the

kinetic energy are unchanged. The *SAPT2* potential is a more refined potential but also involves more floating point operations and a number of conditional branching operations. Comparing *Pacific* versus *Ed* for the *HFDB* versus the *SAPT2* potential on these machines, the slight differences in execution times could be due to the way the branches are handled. In this case also, *Tesla* has the lowest execution time of the three processors for the function evaluations for the various problem sizes.

In Tables 6.5, 6.6 and 6.7, we show the execution times of a single *compute*( ) function call when mixed-precision is employed for the following (i) *HFDB* potential, (ii) *HFDB* potential and trial wave function, and (iii) *HFDB* potential, trial wave function, and kinetic energy. We employ single-precision floating-point for the function evaluations and use double-precision for accumulating the pair-wise energies and trial wave functions. However, as we can infer, there is no real benefit in using mixed-precision on the CPU, as the execution times are only slightly better or sometimes worse than the double-precision implementation.

The *Tesla* CPU implementation was optimized using Intel's Math Kernel Library [117] , which yielded a speedup of 4*x* for the wave function, but with both potential energy and wave function calculations, only a marginal improvement in speedup was observed. Further optimizations of this implementation are left for future work. Since QMC is embarrassingly parallel, we were able to run four copies of the process on *Pacific* to achieve parallelism. As expected, this yielded near linear speedups while making explicit parallelism unnecessary. However, throughout this dissertation, we will compare our GPU and FPGA implementations to the CPU implementation on a single core of *Pacific*, but for fair comparisons against the multi-core implementations, the speedups should be suitably scaled.

Table 6.1: Description of the machines for CPU implementation

Parameters↓ Machines→	*Pacific*	*Ed*	*Tesla*
Machine configuration	Dual-core Dual-processor AMD Opteron 2.2 GHz **Accelerator:** Virtex-4 XC4VLX160 FPGA	Dual-core Dual-processor AMD Opteron 1.8 GHz **Accelerator:** C870 Tesla GPU	Dual quad-core Intel i7 2.67 GHz (Nehalem architecture) **Accelerator:** C1060 Tesla GPU
gcc, Operating system and libraries used	gcc 3.3.3, , -03, SuSE Linux, SPRNG library	gcc 4.1.2, -03, Red Hat, SPRNG library	gcc 4.3.2, -03, Ubuntu, SPRNG library

Table 6.2(a): Execution time (in seconds) for *compute*( ) (*SAPT2* PE, *double-precision*) †

CPUs ↓, Atoms→	64	128	256	512	1024	2048	4096	8192
Pacific	0.001253	0.005061	0.020353	0.082110	0.326270	1.30874	5.22374	21.5712
Tesla	0.001072	0.002956	0.011365	0.039422	0.165410	0.651453	2.62389	10.6599
Ed	0.001127	0.004525	0.018001	0.071840	0.287227	1.14778	4.62140	19.0146

Table 6.3(a): Execution time (in seconds) for *compute*( ) (*SAPT2* PE *and WF*, *double-precision*)

CPUs ↓, Atoms→	64	128	256	512	1024	2048	4096	8192
Pacific	0.001945	0.007814	0.032207	0.128770	0.508790	2.06939	8.29489	33.4688
Tesla	0.000929	0.003747	0.015050	0.060439	0.346481	0.986778	4.05593	15.8788
Ed	0.001728	0.006923	0.027730	0.111017	0.448845	1.81276	7.31264	29.7956

Table 6.4(a): Execution time (in seconds) for *compute*( ) (*SAPT2* PE, *WF, and KE, double-precision*)

CPUs ↓, Atoms→	64	128	256	512	1024	2048	4096	8192
Pacific	0.003103	0.012597	0.050545	0.203221	0.816351	3.27051	13.0645	52.8171
Tesla	0.002352	0.007585	0.021796	0.087492	0.351322	1.40949	5.62834	20.0417
Ed	0.002576	0.010328	0.041513	0.166202	0.667552	2.69638	11.7929	48.7159

† Run times averaged over 10 runs and six significant digits retained for consistency

Table 6.2(b): Execution time (in seconds) for *compute*( ) (*HFDB* PE, *double-precision*)

CPUs ↓, Atoms→	64	128	256	512	1024	2048	4096	8192
Pacific	0.000214	0.000806	0.003155	0.012555	0.049977	0.195755	0.761566	2.848174
Tesla	0.000204	0.000504	0.003598	0.009028	0.041873	0.134326	0.527274	1.914423
Ed	0.000265	0.001028	0.003921	0.015374	0.060539	0.235663	0.914291	3.397688

Table 6.3(b): Execution time (in seconds) for *compute*( ) (*HFDB* PE *and WF*, *double-precision*)

CPUs ↓, Atoms→	64	128	256	512	1024	2048	4096	8192
Pacific	0.000898	0.003516	0.013876	0.055529	0.221739	0.880652	3.50107	13.7741
Tesla	0.000774	0.003075	0.009264	0.029059	0.122849	0.456028	1.82976	7.07175
Ed	0.000909	0.003617	0.013782	0.055338	0.224379	0.887346	3.70142	14.3748

Table 6.4(b): Execution time (in seconds) for *compute*( ) (*HFDB* PE, *WF, and KE*, *double-precision*)

CPUs ↓, Atoms→	64	128	256	512	1024	2048	4096	8192
Pacific	0.002065	0.008152	0.032670	0.131136	0.526595	2.10771	8.41550	33.4486
Tesla	0.001524	0.004345	0.016179	0.056280	0.217704	0.969232	3.68709	14.4947
Ed	0.001710	0.006808	0.027141	0.108627	0.439554	1.77877	7.09371	30.8397

Table 6.5: Execution time (in seconds) for *compute*( ) (*HFDB* PE, *mixed-precision*)

CPUs ↓, Atoms→	64	128	256	512	1024	2048	4096	8192
Pacific	0.000200	0.000759	0.003068	0.012114	0.047487	0.184580	0.713787	2.64266
Tesla	0.000442	0.001684	0.003875	0.018083	0.056380	0.225428	0.785507	2.38867
Ed	0.000257	0.001000	0.003917	0.015484	0.060961	0.235434	0.909389	3.35794

Table 6.6: Execution time (in seconds) for *compute*( ) (*HFDB* PE *and WF*, *mixed-precision*)

CPUs ↓, Atoms→	64	128	256	512	1024	2048	4096	8192
Pacific	0.000870	0.003457	0.013937	0.055704	0.222534	0.884292	3.50169	13.76689
Tesla	0.000933	0.002147	0.010795	0.035660	0.135185	0.526483	2.00233	7.18308
Ed	0.000882	0.003499	0.013900	0.055614	0.223017	0.903003	3.61517	14.3442

Table 6.7: Execution time (in seconds) for *compute*( ) (*HFDB* PE, *WF, and KE*, *mixed-precision*)

CPUs ↓, Atoms→	64	128	256	512	1024	2048	4096	8192
Pacific	0.002034	0.008192	0.032843	0.132315	0.529312	2.12226	8.46132	27.4539
Tesla	0.001700	0.003880	0.018476	0.063733	0.246816	0.988119	3.82599	15.3541
Ed	0.001769	0.007110	0.027736	0.112401	0.448741	1.81056	7.21331	28.5052

Figure 6.1 (*Left*) shows the execution times (in seconds) for a single *compute*( ) function call for various problem sizes, when (i) only the *HFDB* potential is calculated, (ii) only the trial wave function is calculated, and (iii) only the kinetic energy computation is performed. In Figure 6.2 (*Left*), we repeat our experiments with the *SAPT2* potential energy. These times are measured on a single core of the dual-core dual-processor AMD Opteron *Pacific* (our baseline platform throughout this dissertation). We obtain the times for each of the functions to better understand which of these functions dominate the computations, and how partitioning the application to map these computations on FPGA or GPU co-processors would accelerate the overall application. For the above two cases, the analytical functions that we employ for the trial wave function and hence for the kinetic energy are unchanged. In the case of the *HFDB* potential, the calculation of all the terms contributing to the overall kinetic energy is the most dominant part of the computations, followed by the calculation of wave function and the potential energy. When the *SAPT2* potential is used, the potential energy forms the dominant part of the computation, with the execution times of kinetic energy closely following the potential energy. We summarize these results in Figures 6.1 and 6.2 (*Right*) for N = 1024 to 8192 to compare the execution times of the *HFDB* and *SAPT2* potential energy calculations, relative to the trial wave function and kinetic energy calculations.

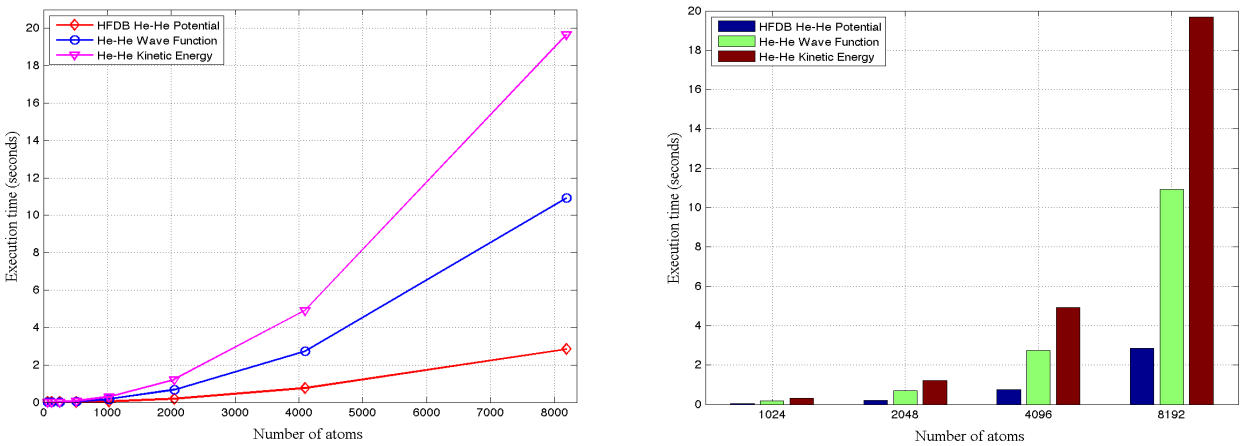

Figure 6.1: (*Left*) Execution times (in seconds) for *compute*( ) for *HFDB* PE, WF, and KE (*Right*) Execution times shown for N = 1024, 2048, 4096 and 8192 (*Pacific* Cray XD1 2.2 GHz Opteron)

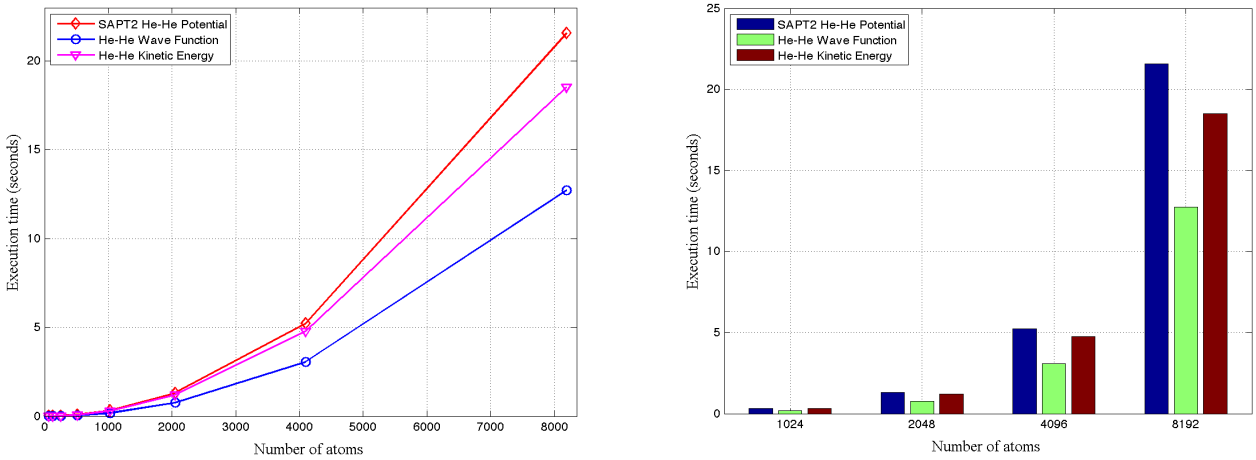

Figure 6.2: (*Left*) Execution times (in seconds) for *compute*( ) for *SAPT2* PE, WF and KE (*Right*) Execution times shown for N = 1024, 2048, 4096 and 8192 (*Pacific* Cray XD1 2.2 GHz Opteron)

In Table 6.8, we provide the execution times of the QMC algorithm on *Pacific*, where we sample 10 configurations computing 200 equilibration and 200 steady-state iterations (E = 10, I_{EQ} = 200, I_{SS} = 200). The last two rows of Table 6.8 provide the execution times of the QMC algorithm including calculations of both potential and kinetic energy, yielding the total ground-state energy for an atomic cluster. The experiments are repeated when *HFDB* and *SAPT2* potentials are employed. These results are summarized in Figure 6.3. An interesting observation from Table 6.8 is that the calculation of *HFDB* potential energy, trial wave function, and kinetic energy takes roughly the same amount of time as the *SAPT2* potential energy and WF calculations. From these results as well as those summarized in Figures 6.1 and 6.2, it is clear that the performance of the CPU implementation heavily depends on the analytical functions that we employ for the potential energy, kinetic energy, and trial wave function.

Parallelization of the QMC algorithm is accomplished by distributing the configurations among the processors in a multiprocessor environment. Each processor uses a uniquely seeded random number generator to eliminate any correlation between the random number streams. A Message Passing Interface (MPI) implementation is set up and we divide the E ensembles (or configurations) among *nproc* processors. In our experiments, we use a maximum of 4 processors on *Pacific*. We perform the experiments for the serial case (*nprocs*=1) and for *nprocs* = 2, 3 and 4. Tables 6.9 and 6.10 show the results from the MPI implementation of the QMC algorithm for *HFDB* potential and WF calculations, and for *SAPT2* potential and WF calculations. The simulation parameters are E = 120, I_{EQ} = 200, and I_{SS} = 200. Communication between the processes is needed only at the end of the algorithm when the slave processes send the results to a master process, which accumulates the results to calculate the average ground-state energy. Hence, as one would expect, we achieve a linear speedup in this case. These results are summarized in Figure 6.4 for the *HFDB* potential and WF calculations.

Table 6.8: Execution time (in seconds) of the serial QMC algorithm on *Pacific* Cray XD1 2.2 GHz Opteron (E = 10, I_{EQ} = 200, I_{SS} = 200 iterations)

Functions ↓, Atoms→	64	128	256	512	1024	2048	4096
HFDB Potential	0.813164	3.15606	12.4238	49.45375	196.869	773.960	3054.45
SAPT2 Potential	5.36394	21.5193	86.5714	354.708	1407.55	5567.32	22443.2
HFDB and WF	3.47493	13.9419	55.8101	223.177	890.905	3523.42	14015.4
SAPT2 and WF	9.02411	32.4207	130.0553	521.064	2148.11	8285.27	33185.4
HFDB, WF and KE	8.17277	32.8224	131.2376	525.549	2103.68	8423.31	33594.4
SAPT2, WF and KE	12.5659	50.3417	203.2685	812.239	3253.53	13071.2	52168.9

Table 6.9: Execution time (in seconds) of the MPI QMC algorithm on *Pacific* Cray XD1 2.2 GHz Opteron (with *HFDB* potential) [†] (E = 120, I_{EQ} = 200, I_{SS} = 200 iterations)

Function ↓, Atoms→		64	128	256	512	1024	2048
HFDB Potential and WF	*nprocs* = 1	41.02160	165.0960	657.34731	2667.7498	10491.489	42172.12
	nprocs = 2	20.55574	82.62442	329.93740	1318.73519	5242.8546	21119.15
	nprocs = 3	13.71532	55.01988	220.30582	883.53758	3519.4539	14083.77
	nprocs = 4	10.26351	41.32460	166.09703	673.46069	2640.3723	10867.63

[†] Simulated up to N = 2048 due to extremely long run times for large N

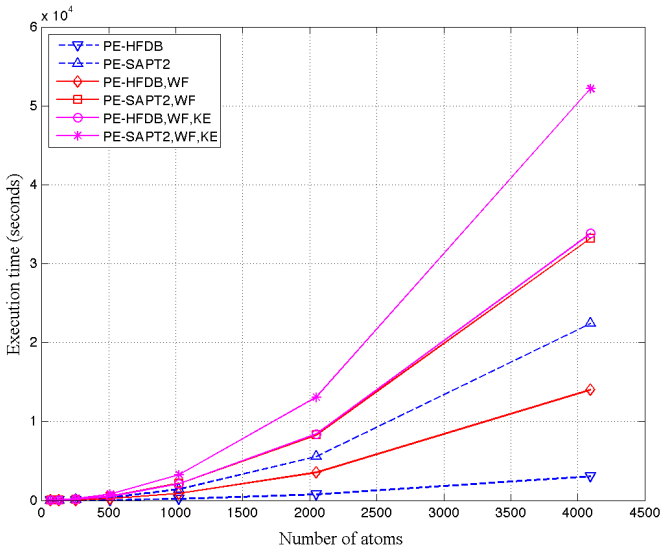

Figure 6.3: Comparison of execution times (in seconds) on *Pacific* Cray XD1 2.2 GHz Opteron for various functions (E = 10, I_{EQ} = 200, I_{ss} = 200)

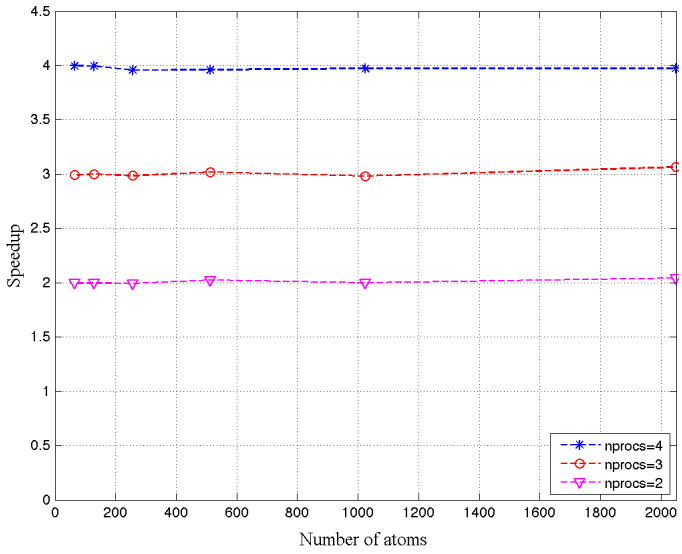

Figure 6.4: Speedup for MPI Implementation of the QMC algorithm (*HFDB*-PE and WF calculations) on *Pacific* Cray XD1 2.2 GHz Opteron (E = 120, I_{EQ} = 200, I_{ss} = 200)

6.3. Cray XD1 Hardware Accelerated QMC

The hardware accelerated QMC application is targeted to the Cray XD1 HPRC platform. The QMC application runs on a single core of a compute node on Cray XD1 and the potential energy and wave function calculations are implemented on the Virtex-4 LX160 FPGA. The use of a higher gate density FPGA such as the Virtex-4 allows us to map the potential and wave function calculations on a single FPGA. Cray also provides the Application Acceleration API [43], which are a set of functions to access the FPGA within an application on the Opteron using the Linux operating system. The command line FPGA control utility provided by Cray can also be used to control the FPGA, such as resetting the FPGA or loading the bitstream onto the FPGA [43]. The programming interface provided by Cray for the XD1 system greatly simplifies interfacing an application running on the Opteron with the design implemented on the FPGA.

As outlined in Chapter 4, the kernels of the chemistry application are coded using the hardware description language, VHDL, and the design is synthesized using Xilinx XST tools, which is part of the Xilinx 8.1 ISE/EDK toolkit [111]. The place and route is also performed with the Xilinx ISE tools. A hardware-software partitioning method is used; the host processor initializes the configurations and the FPGA calculates the potential energy and trial wave function of each configuration. The results are then transferred back to the host processor, which decides whether to accept or reject the configuration. Hence, we only need to replace the *compute*( ) function, where we calculate the trial wave function, and potential energy in the original software application with the FPGA function call. The interpolation coefficients and parameters required by the FPGA are directly written onto the Block RAM from within the QMC application running on the Opteron. A set of $O(N)$ partial accumulated results are written to registers and read within the host application. On the Virtex-4, we are unable to fit all the functions contributing to the

overall kinetic energy. Hence, the results from porting the potential energy and wave function are presented in this discussion.

Our design operates at 100 MHz and is deeply pipelined (58 pipeline stages) and we use a combination of the dedicated DSP48s and slices to implement multipliers. Table 6.10 shows the summary of resources of the PE and WF ported to the Cray XD1. The potential energy calculation uses roughly 25 percent of all the FPGA resources and hence we are able to fit a WF pipeline on the same FPGA. Placing multiple pipelines on the same FPGA allows independent units to share resources. For instance, in our current design, the distance calculation module along with the Block RAMs that store the configurations can be shared by PE and WF pipelines. With both PE and WF kernels ported to the FPGA, the design consumes roughly 50 percent of the FPGA. The remaining resources can be used to simulate larger clusters or fit additional functions. We could also use the available resources to calculate related ground-state properties, which are often of utmost importance for studying *N*-body systems. Table 6.11 compares the FPGA execution times for a *compute*( ) function call (including the time for data transfers and computation), that calculates only the potential energy, or both potential energy and wave function. Unlike the CPU implementation, the FPGA implementation has similar run times for the potential energy functions (*HFBD* and *SAPT2*), as shown in Figure 6.5, irrespective of their complexities, due to the generic interpolation approach used to evaluate the functions. We compare the FPGA run times for PE and WF calculations to the CPU execution times on *Pacific* in Figure 6.6.

Table 6.10: FPGA resource usage on *Pacific* Cray XD1 Virtex-4 VLX160 FPGA

Kernel /Resource type	PE	PE and WF
SLICEs (67,584)	17, 984 (26%)	30,432 (45%)
BRAMs (288)	79 (27%)	147 (51%)
DSP48s (96)	25 (26%)	50 (52%)

Table 6.11: Execution time (in seconds) for FPGA-accelerated *compute*( ) per iteration on *Pacific* Cray XD1 2.2 GHz Opteron (*fixed-point*)

Function ↓, Atoms→	64	128	256	512	1024	2048	4096
HFDB/*SAPT2* Potential Only	0.000075	0.000131	0.000373	0.001367	0.005322	0.02108	0.08398
HFDB/*SAPT2* Potential and Wave Function	0.000137	0.000191	0.000450	0.001450	0.005381	0.021169	0.084141

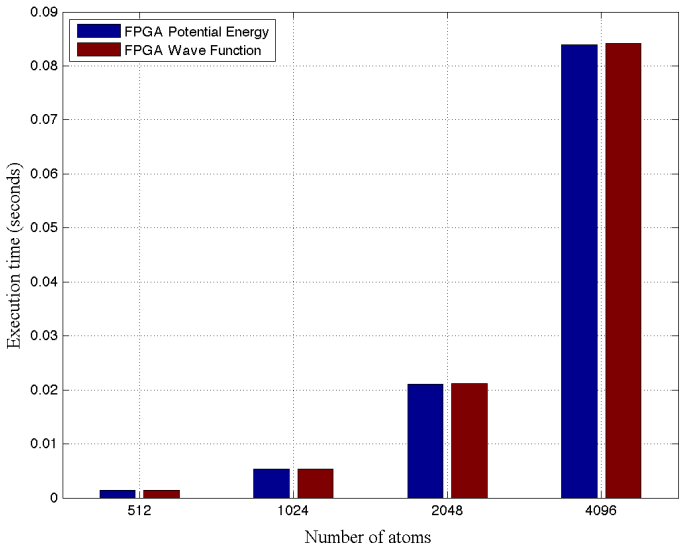

Figure 6.5: Comparison of execution times (seconds) for FPGA-accelerated *compute*( ) for PE and WF on *Pacific* Cray XD1 2.2 GHz Opteron (shown for N = 512, 1024, 2048 and 4096)

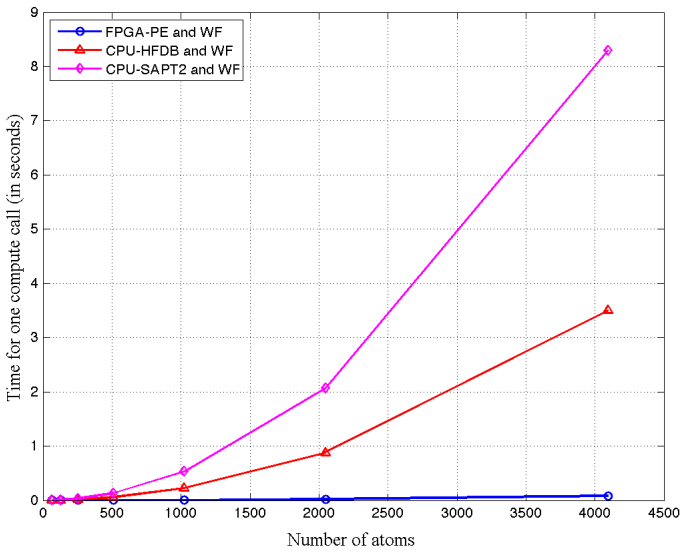

Figure 6.6: Comparison of CPU and FPGA execution times (in seconds) for *compute*( ) for PE and WF calculations on *Pacific* Cray XD1 2.2 GHz Opteron with Virtex-4 FPGA

Table 6.12 shows the execution time (in seconds) for the FPGA-accelerated fixed-point QMC application when simulated for 400 iterations with 10 ensembles. The times are shown for the two cases, when only potential energy, and when both potential energy and wave function calculations, are included. In Figure 6.7, we plot the execution times for the FPGA-accelerated QMC application along with the double-precision CPU implementation on *Pacific* for the same simulation parameters, as shown in Table 6.8. Figure 6.8 shows the speedup of the FPGA implementation over the CPU implementation. For the *HFDB* potential and trial wave function calculations, we observe a speedup of 40*x* over the software implementation as the number of atoms is increased to 4096 (the maximum number of atoms the Block RAMs are currently designed to store in the present implementation). With the *SAPT2* potential and trial wave function calculations, the speedup approaches 100*x* over the CPU implementation as we increase the number of atoms to 4096. While using accelerators to speed up computations, it is important to ensure that the ratio of computation to communication is high. As the problem size increases, the computations on the FPGA far exceed the cost of data transfers between the FPGA and the processor, and we observe a significant improvement in the execution times and a steady increase in speedup. For system sizes

between 100 and 1000 atoms, we still observe a speedup, however the speedup is limited due to the communication overheads between the FPGA and the processor. For smaller systems of atoms (less than 50), any performance gained from the FPGA acceleration of the kernels is offset by the transfer overheads and hence there is no practical benefit from using hardware acceleration. Increasing the number of atoms only requires trivial changes in the design to increase the depth of the position Block RAMs, providing the capability to researchers to perform large-scale simulations of atomic and molecular clusters, which are not feasible to realize using laboratory experiments.

Table 6.13 shows the components of the FPGA execution time for the QMC simulation of a cluster with 4096 atoms. We also show the corresponding CPU execution times with the two potential energies. The overheads in the FPGA implementation takes into account the time for configuring the FPGA with the bitstream and the time to load the interpolation coefficients onto the on-chip Block RAM. With the high performance RapidArray interconnect that couples the FPGA directly to the AMD Opteron, the communication overhead is negligible compared to the time spent on the actual kernel. The rest of the application consists of generating the reference configuration and making a decision whether to accept or reject the present configuration during each time step of the simulation.

Table 6.12: Execution time (in seconds) of the FPGA accelerated QMC algorithm on *Pacific* Cray XD1 2.2 GHz Opteron (E = 10, I_{EQ} = 200, I_{SS} = 200, *fixed-point*)

Function ↓, Atoms→	64	128	256	512	1024	2048	4096
HFDB/*SAPT*2 Potential	4.070662	4.237045	5.397105	9.425678	25.469545	88.987942	341.898050
HFDB/*SAPT*2 Potential and WF	4.239053	4.534727	5.562009	9.641202	25.665263	89.210814	342.388818

Table 6.13: Comparison of CPU and FPGA execution times (in seconds) of the QMC algorithm on *Pacific* Cray XD1 2.2 GHz Opteron (E = 10, I_{EQ} = 200, I_{SS} = 200)

Implementation ↓, Components →	PE+WF	Rest of the application	Overhead	Total
FPGA accelerated QMC (*fixed-point*)	337.141	1.4161	3.83192	342.3889
Software QMC (*double-precision*) *HFDB* PE	14013.97	1.4161	--	14015.386
Software QMC (*double-precision*) *SAPT2* PE	33184.025	1.4161	--	33185.441

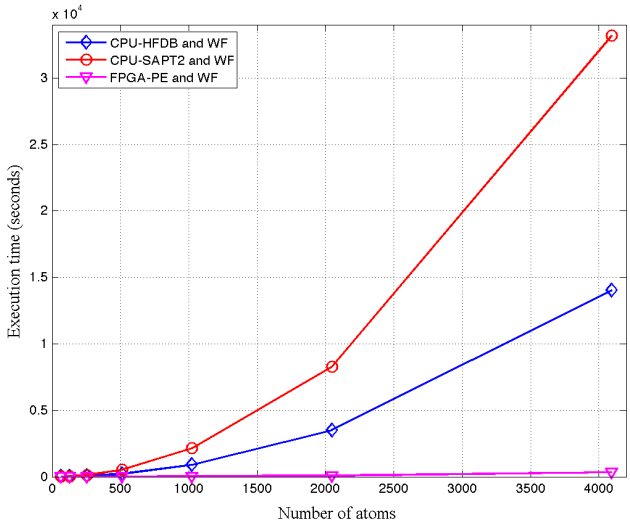

Figure 6.7: Comparison of CPU and FPGA execution times (in seconds) for PE and WF calculations on *Pacific* Cray XD1 2.2 GHz Opteron with Virtex-4 FPGA

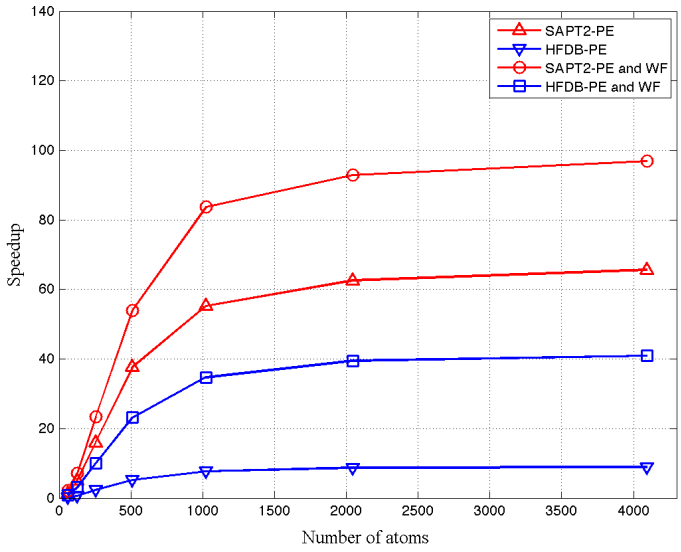

Figure 6.8: Speedup of FPGA-accelerated kernels (*fixed-point*) versus CPU implementation on *Pacific* Cray XD1 2.2 GHz Opteron with Virtex-4 FPGA (*double-precision*)

As the next step in the FPGA implementation, we map the kernels to multiple FPGA nodes on the Cray XD1. Parallelization is achieved in a manner similar to the CPU implementation. We use an MPI setup on 4 computing nodes, each node with a Virtex-4 FPGA. We perform a QMC simulation with 120 ensembles for a total of 400 iterations. The ensembles are equally divided among the four nodes. Each process in a computing node is responsible for configuring its FPGA, loading the interpolation coefficients to its FPGA, and processing its set of configurations using the pipelined FPGA architecture to produce the potential energy and wave function. There is communication only at the end of the algorithm, where a master process collects the results from the slave processes. Tables 6.14 and 6.15 show the results of the MPI multi-FPGA implementation on *Pacific* with the PE kernel, and both PE and WF kernels, respectively. In Figures 6.9 and 6.10, we plot the execution time (in seconds) for multiple RC nodes, and the speedup while using multiple FPGAs compared to a single FPGA equipped node. As we would expect, we obtain a speedup linear in the number of FPGA equipped nodes. We reinforce our results thus far in Figure 6.11, where we illustrate the performance benefits of using a parallel implementation that makes use of the FPGAs, as opposed to a purely software-based parallel implementation. With four computing nodes, equipped with reconfigurable devices on the *Pacific* Cray XD1, we demonstrate a speedup of 160*x* over the software implementation, in contrast to a speedup of 4*x* with four computing nodes (or) processors without reconfigurable logic elements.

Table 6.14: Execution times (in seconds) for the MPI implementation of hardware-accelerated QMC (PE only) on *Pacific* Cray XD1 2.2 GHz Opteron (E = 120, I_{EQ} = 200, I_{SS} = 200)

Function ↓, Atoms→		64	128	256	512	1024	2048	4096
PE only	*nprocs* = 1	6.78035	10.00056	22.19953	70.94682	263.8102	1025.717	4034. 041
	nprocs = 2	5.22259	6.61824	13.17679	37.57461	133.8511	514.8199	2031.912
	nprocs = 3	4.81228	6.08553	9.77974	26.27093	90.49975	344.4642	1355.945
	nprocs = 4	4.55843	5.39152	8.54499	20.65115	68.89143	259.3319	1018.139

Table 6.15: Execution times (in seconds) for the MPI implementation of hardware-accelerated QMC (PE and WF) on *Pacific* Cray XD1 2.2 GHz Opteron (E = 120, I_{EQ} = 200, I_{SS} = 200)

Function ↓, Atoms→		64	128	256	512	1024	2048	4096
PE and WF	*nprocs* = 1	8.40904	12.09592	24.12624	73.64530	266.07701	1027.975	4036.189
	nprocs = 2	6.29558	7.53265	14.27281	38.73011	134.7092	515.7861	2032.961
	nprocs = 3	5.38157	6.59500	10.79812	27.02720	91.27919	345.2168	1356.698
	nprocs = 4	5.022011	5.87889	9.16111	21.18694	69.46543	259.9759	1018.675

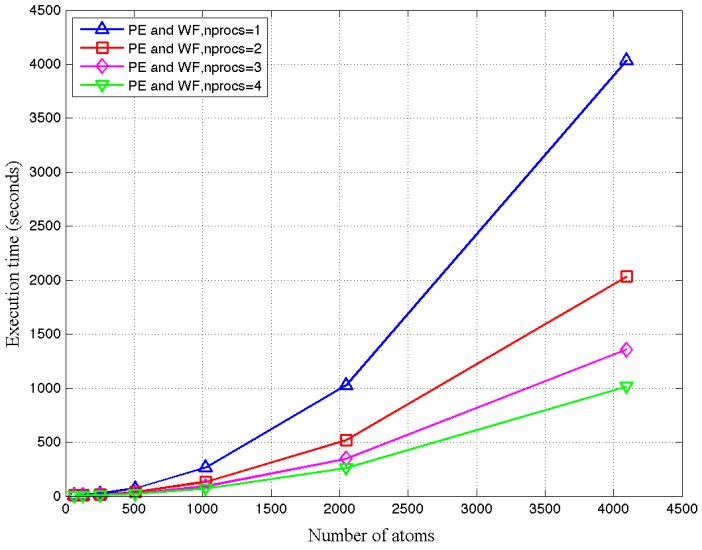

Figure 6.9: Execution time (seconds) for QMC using the MPI multi-FPGA implementation on *Pacific* Cray XD1

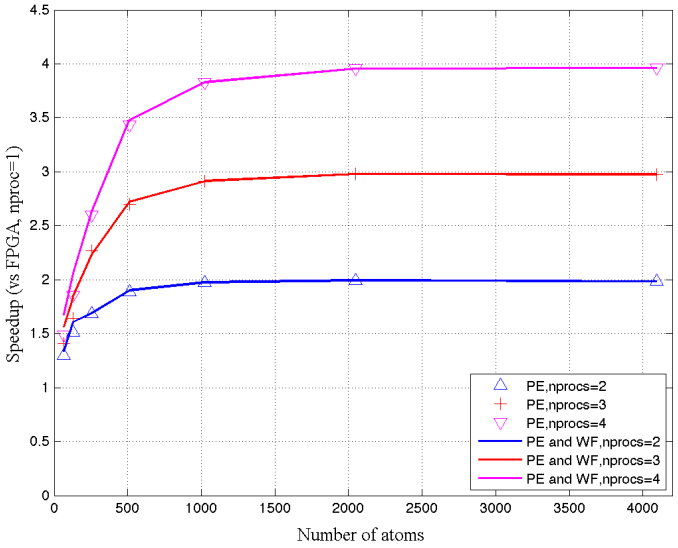

Figure 6.10: Speedup using the MPI multi-FPGA on *Pacific* Cray XD1 compared to a single FPGA

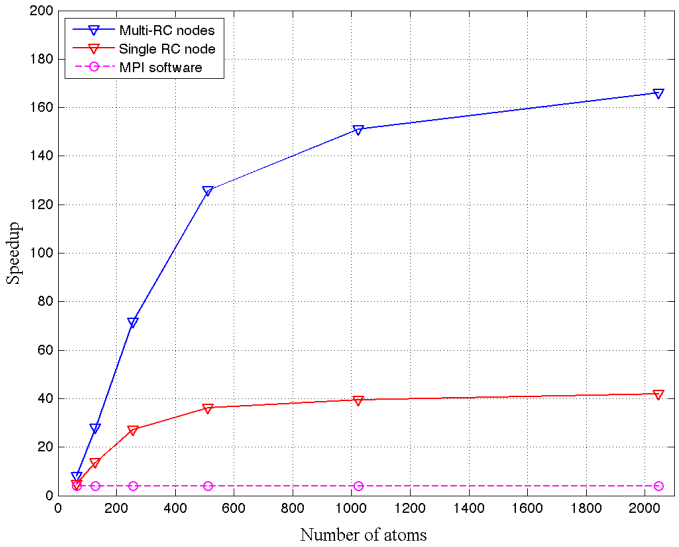

Figure 6.11: Comparison of speedups for QMC using software MPI, single RC node, and multiple RC nodes on *Pacific* Cray XD1

6.3.1. Error Analysis

Scientific applications commonly use floating-point representation as they allow for high precision arithmetic and higher dynamic range compared to fixed-point representation. The QMC software application employs 64-bit double-precision floating-point representation. One of the important concerns while using the FPGA accelerators for scientific computing is whether they can offer a performance improvement without any loss in numerical accuracy. The use of floating-point representation on the hardware is more expensive and slower than using an integer representation. We overcome this problem using a fixed-point representation, which is essentially an integer representation. The fixed-point representation used for the squared distances fed to the pipelines and potential and wave function results (prior to accumulation) is shown in Table 6.16. Using a fixed-point representation allows us to save area and increase the speed of operation. Since the potential and wave function values are rescaled to be less than or equal to 1, their fixed-point representations use 52 bits after the radix point. These representations deliver the required accuracy for our chemistry application.

Figures 6.12 and 6.13 show the absolute error of potential energy and wave function for the 52-bit hardware versus the same function expressed in 64-bit double-precision (52-bit mantissa) floating point representation. The plots are shown on a log (base 2) scale so that we can equate the error with the bits of precision. The plots show that the error for fixed-point spline levels out nicely at our 52-bit maximum fixed-point limit, although the floating-point representation has smaller error and is more precise. The fixed-point version for the potential fails for large values of r^2. However, the squared distances in our application are confined within 2^{16} and the fixed-point potential energy and trial wave function values compare well with the floating-point values within this region. Hence, we can safely ignore the degradation of accuracy in this poor region.

Table 6.16: Fixed-point representations for FPGA implementation of QMC kernels

Parameter	Fixed-point formats (s-signed, u-unsigned)
(x, y, z) positions	s12.20
Squared distances	u27.26
Potential energy and wave function	s0.51
Interpolation coefficients	s0.51

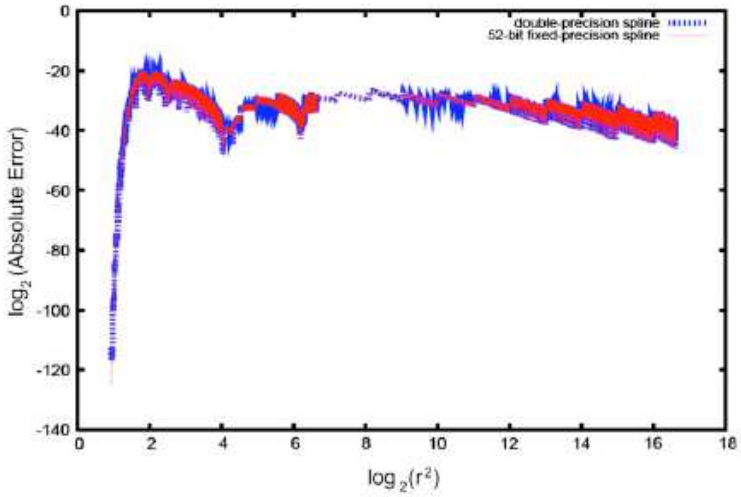

Figure 6.12: Absolute error of potential energy

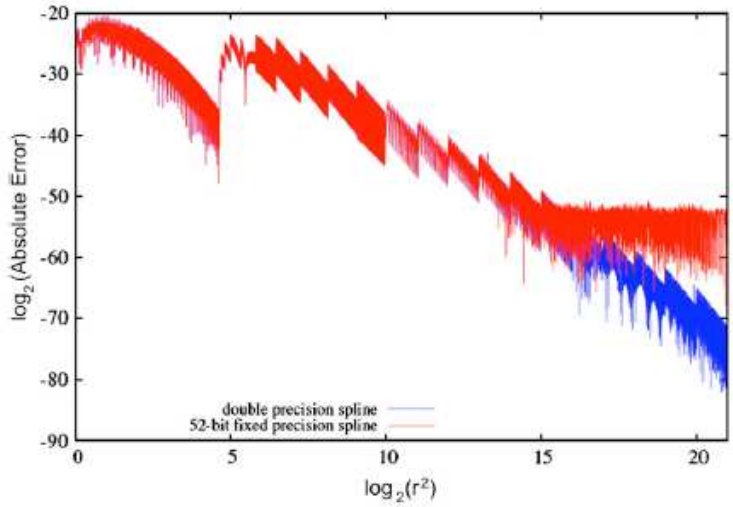

Figure 6.13: Absolute error of wave function

6.4. GPU Implementation

The results from the naïve and optimized GPU implementations of the *HFDB* PE kernel are summarized in Tables 6.17(a), (b)-6.20(a), (b). We plot the results from Tables 6.17-6.20 for both the naïve and optimized implementations in Figures 6.14 – 6.17. In Tables 6.17(a) and 6.18(a), we show the execution times (in seconds) for a *compute*( ) call (times for data transfers and the GPU kernel execution) that evaluates the *HFDB* PE kernel on the C870 and C1060 Tesla GPUs using single-precision. Tables 6.19(a) and 6.20(a) show the execution times (in seconds) for a *compute*( ) call that evaluates the *HFDB* PE kernel on the C1060 using double- and mixed- precision, respectively. It is interesting to observe the transitioning of the "optimal" number of threads and blocks for various problem sizes. For best performance, it is important to have the right number of threads, which keeps a multiprocessor busy to maximize thread-level parallelism, as well the right number of blocks to maximize the use of multiprocessors to take advantage of task-level parallelism. For each time slice, the GPU kernel fetches N^2/T memory elements, where T is the number of threads in the block and the size of the matrix is N x N. Hence increasing the number of threads, T, decreases the accesses to the device memory. There are 16 multiprocessors on the C870 and 30 multiprocessors on the C1060. The number of multiprocessors places a lower limit on the "optimal" number of threads, T, which is chosen so that the number of thread blocks, N/T, is greater than 16 and 30 on the C870 and C1060 respectively, so the multiprocessors are not idle. Inspecting the results in Tables 6.17(a), 6.18(a), 6.19(a), and 6.20(a), for small problem sizes, N = 256 and 512, the use of T = 32 is optimal as this results in a larger number of thread blocks, keeping the multiprocessors busy. (For N = 256, it is true that the use of T < 32 yields even small execution times). For larger problem sizes, there is a balance between the number of threads and the number of thread blocks scheduled on the multiprocessors. This leads to the performance variations between the C870 and C1060 with 128 and 240 processing cores. For instance, referring to Table 6.17(a) on the C870 GPU, when N = 4096, the use of T = 32 (so that number of thread blocks, N/T = 128, which is the maximum

number of concurrent thread blocks on all SMs) is less efficient than the use of $T = 64$ which results in 64 thread blocks, each with 64 threads. Comparing this to the C1060 results in Table 6.18(a), using $T = 32$ still yields the lowest run time as the C1060 with 30 SMs can have a total of 240 blocks. However, when $N = 8192$ using $T = 32$ results in 256 thread blocks which is greater than 240, yielding a higher run time than the use of $T = 64$, which results in 128 thread blocks.

For the double-precision naïve implementation on Tesla C1060 shown in Table 6.19(a), using 32 threads yields the best performance for all problem sizes up to $N = 4096$ for reasons discussed earlier. Using $T = 64$ gives the lowest run time as in the case of single-precision. A smaller number of threads results in a larger number of thread blocks which are assigned per multiprocessor to hide the latencies of pipelined instruction execution, which in double-precision, can take a large number of clock cycles. The mixed-precision implementation results in Table 6.20(a) exhibits a similar behavior to the results discussed earlier, but yielding a performance substantially greater than the double-precision implementation and only slightly lower than the single-precision performance. Tables 6.17(b)-6.20(b) present the results from the optimized implementation. We observe a similar trend in the performance as we vary the number of threads, as with the naïve implementation. However, in this case, we only compute the lower triangular matrix, which results in a reduction in the execution time per *compute*( ) function call in all the cases. We choose the number of threads as 64 for our discussion. In Figure 6.18, we compare the speedups (per *compute*( ) iteration) of the naïve and optimized implementations in single-precision for the PE calculation on C870 and C1060, double-precision on C1060 and mixed-precision on C1060. The single-precision optimized implementation on C1060 yields a speedup approaching 300*x* (for 8192 atoms) over the baseline CPU implementation on *Pacific*. The optimized versions using mixed-precision on C1060, and single-precision on C870 offer a speedup of 250*x*. The naïve and the optimized double-precision implementation on C1060 exhibit the least performance, providing speedups approaching 20*x* and 40*x* respectively.

Table 6.17(a): Execution time (in seconds) for *compute*( ) (*PE kernel*, *single-precision*, *naïve*) on C870

Threads ↓, Atoms→	256	512	1024	2048	4096	8192
32	0.000353	0.001020	0.001888	0.002838	0.005522	0.018643
64	0.000360	0.001023	0.001869	0.002792	0.005314	0.016083
128	0.000363	0.001036	0.001931	0.002795	0.005295	0.016052
256	0.000409	0.001139	0.002124	0.003185	0.005284	0.016058

Table 6.18(a): Execution time (in seconds) for *compute*( ) (*PE kernel*, *single-precision*, *naïve*) on C1060

Threads ↓, Atoms→	256	512	1024	2048	4096	8192
32	0.000329	0.000718	0.001558	0.002610	0.005053	0.018933
64	0.000350	0.000729	0.001403	0.002565	0.005342	0.013302
128	0.000379	0.000803	0.001681	0.002580	0.005934	0.015687
256	0.000452	0.000971	0.001876	0.003221	0.005969	0.020745

Table 6.19(a): Execution time (in seconds) for *compute*( ) (*PE kernel*, *double-precision*, *naïve*) on C1060

Threads ↓, Atoms→	256	512	1024	2048	4096	8192
32	0.002006	0.003915	0.007061	0.015742	0.042822	0.163610
64	0.002385	0.004702	0.007616	0.018572	0.049347	0.156795
128	0.003261	0.006275	0.010691	0.020281	0.063013	0.186714
256	0.005133	0.010071	0.019014	0.035930	0.069569	0.252813

Table 6.20(a): Execution time (in seconds) for *compute*( ) (*PE kernel*, *mixed-precision*, *naïve*) on C1060

Threads ↓, Atoms→	256	512	1024	2048	4096	8192
32	0.000325	0.000897	0.001543	0.002913	0.005915	0.021981
64	0.000336	0.001039	0.001526	0.002930	0.006372	0.016701
128	0.000386	0.001157	0.001748	0.002938	0.007282	0.019688
256	0.000515	0.001252	0.002254	0.003933	0.007292	0.025953

Table 6.17(b): Execution time (in seconds) for *compute*( ) (*PE, single-precision, optimized*) on C870

Threads ↓, Atoms→	256	512	1024	2048	4096	8192
32	0.000345	0.000959	0.001779	0.002649	0.005487	0.014921
64	0.000358	0.000964	0.001771	0.002644	0.004657	0.011857
128	0.000365	0.000997	0.001817	0.002656	0.004776	0.012112
256	0.000372	0.001061	0.001976	0.003024	0.005141	0.013488

Table 6.18(b): Execution time (in seconds) for *compute*( ) (*PE, single-precision, optimized*) on C1060

Threads ↓, Atoms→	256	512	1024	2048	4096	8192
32	0.000321	0.000775	0.001310	0.002362	0.004341	0.010287
64	0.000336	0.000783	0.001322	0.002374	0.004370	0.009730
128	0.000349	0.000854	0.001574	0.002603	0.004606	0.010367
256	0.000372	0.000890	0.001851	0.003072	0.005755	0.011733

Table 6.19(b): Execution time (in seconds) for *compute*( ) (*PE, double-precision, optimized*) on C1060

Threads ↓, Atoms→	256	512	1024	2048	4096	8192
32	0.001678	0.003518	0.006214	0.012705	0.030495	0.092039
64	0.002027	0.004131	0.007141	0.013835	0.033445	0.090882
128	0.002608	0.005509	0.009928	0.018795	0.039108	0.103483
256	0.003355	0.008222	0.016553	0.032537	0.065010	0.129098

Table 6.20(b): Execution time (in seconds) for *compute*( ) (*PE, mixed-precision, optimized*) on C1060

Threads ↓, Atoms→	256	512	1024	2048	4096	8192
32	0.000324	0.000884	0.001373	0.002476	0.004738	0.011630
64	0.000357	0.001037	0.001405	0.002552	0.004920	0.011208
128	0.000377	0.001124	0.001573	0.002732	0.005299	0.012081
256	0.000408	0.001280	0.002104	0.003680	0.006979	0.014100

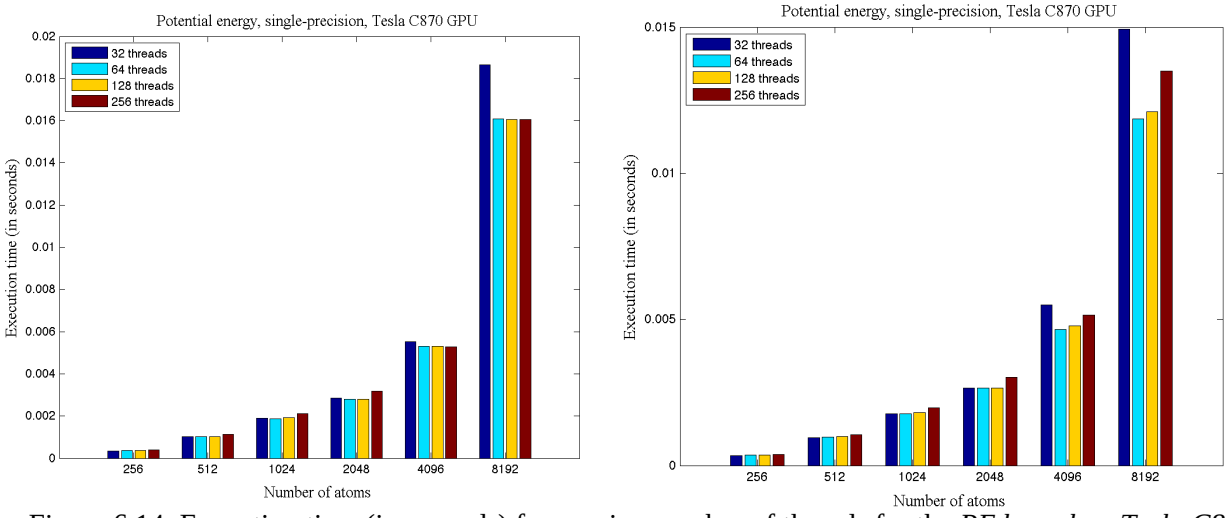

Figure 6.14: Execution time (in seconds) for varying number of threads for the *PE kernel* on *Tesla C870 GPU* (*single-precision*) (*Left*) Naïve implementation (*Right*) Optimized implementation

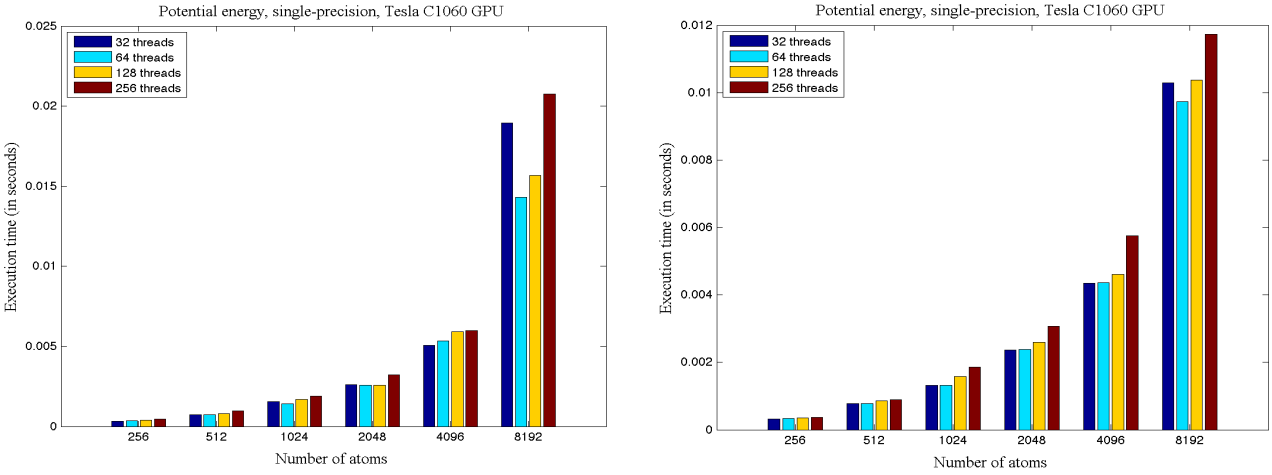

Figure 6.15: Execution time (in seconds) for varying number of threads for the *PE kernel* on *Tesla C1060 GPU* (*single-precision*) (*Left*) Naïve implementation (*Right*) Optimized implementation

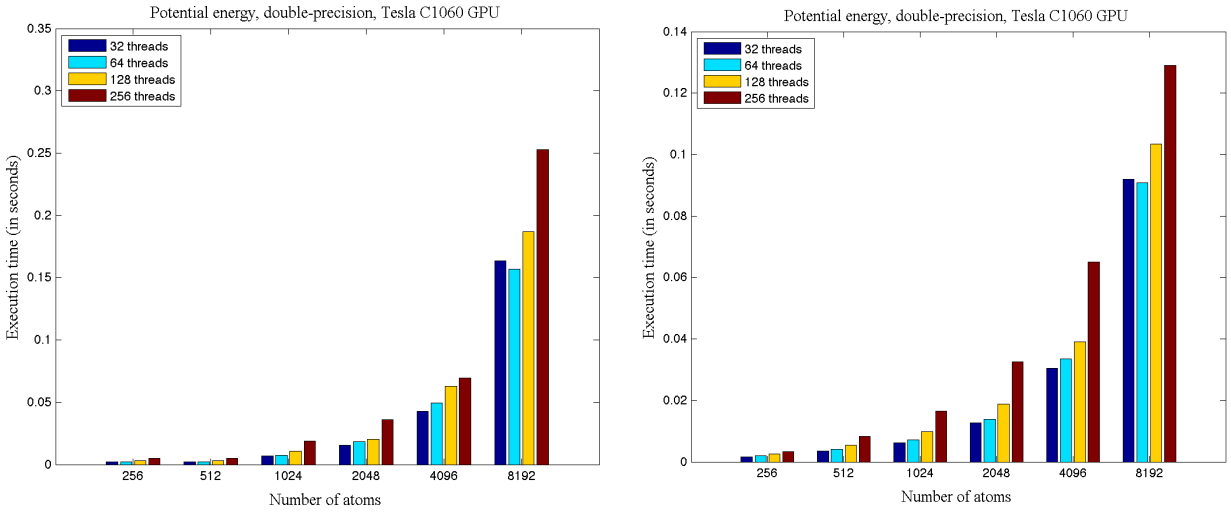

Figure 6.16: Execution time (in seconds) for varying number of threads for the *PE kernel* on *Tesla C1060 GPU* (*double-precision*) (*Left*) Naïve implementation (*Right*) Optimized implementation

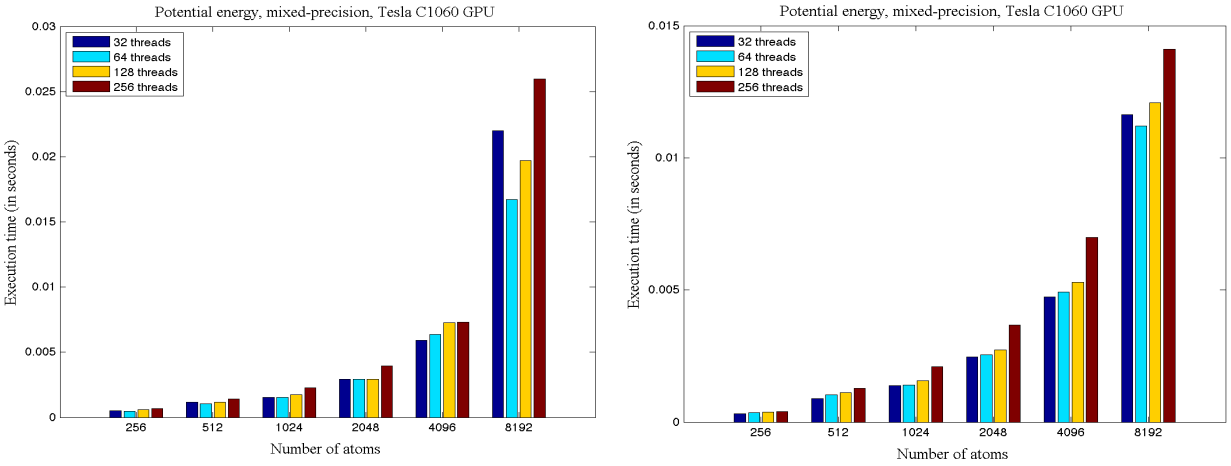

Figure 6.17: Execution time (in seconds) for varying number of threads for the *PE kernel* on *Tesla C1060 GPU* (*mixed-precision*) (*Left*) Naïve implementation (*Right*) Optimized implementation

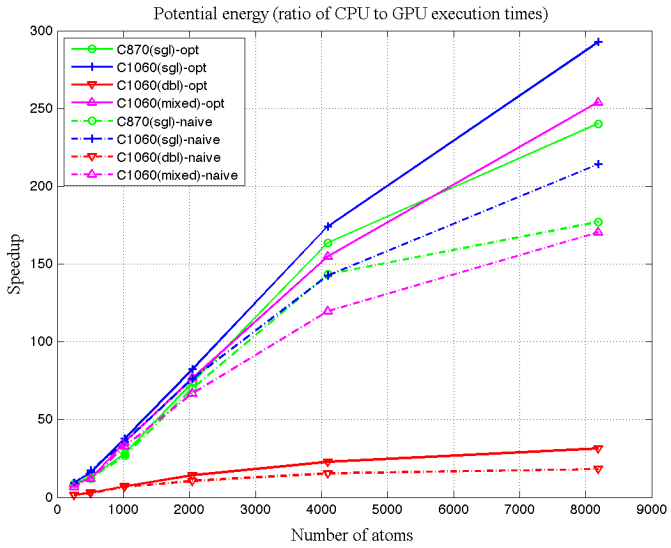

Figure 6.18: Speedup per iteration for the *CUDA PE kernel (Naïve and optimized*) with *64 threads* on the GPU and QMC on *Pacific* Cray XD1 2.2 GHz Opteron (*double-precision*) as the baseline

The execution times per iteration for the *HFDB* PE and WF kernels for naïve and optimized GPU implementations are summarized in Tables 6.21(a), (b)-6.24(a),(b). We plot the results for the naïve and optimized implementations in Figures 6.19-6.22. We observe similar trends in the execution times as seen with the PE kernel. In Figure 6.23, we compare the speedup performance of a single *compute*( ) function call on the C870 and C1060 GPUs (with 64 threads) versus a single iteration of the *compute*( ) function using double-precision on the Pacific. The optimized implementations on C1060 using single- and mixed-precision respectively, exhibit the best overall speedups of about 425*x* and approaching 400*x* (for N=8192) respectively. The naïve implementation using mixed-precision on the C1060 and the optimized single-precision version on C870 offer a speedup slightly greater than 250*x*. The decrease in performance of the naïve single-precision implementation using 64 threads on C870 is due to the fact that T = 256 actually provides the lowest execution time in this case as shown in Table 6.21(a), probably due to the fact there are more active threads per processor leading to a number of active warps, which are useful to hide global memory latencies, especially with large N. The double-precision performance shows the lowest speedup, approaching 50*x* and 25*x* respectively, for the optimized and naïve versions.

Table 6.21(a): Execution time (in seconds) for *compute*( ) (*PE and WF, single-precision, naïve*) on C870

Threads ↓, Atoms→	256	512	1024	2048	4096	8192
32	0.001090	0.002451	0.005169	0.009157	0.033754	0.099104
64	0.001120	0.002485	0.005154	0.009099	0.021901	0.086023
128	0.001168	0.002594	0.005371	0.009105	0.021814	0.087381
256	0.001466	0.003183	0.006587	0.011354	0.021899	0.079305

Table 6.22(a): Execution time (in seconds) for *compute*( ) (*PE and WF, single-precision, naïve*) on C1060

Threads ↓, Atoms→	256	512	1024	2048	4096	8192
32	0.000952	0.001941	0.003972	0.007575	0.015925	0.066030
64	0.000981	0.002036	0.004045	0.007954	0.016816	0.043525
128	0.001054	0.002101	0.004189	0.007815	0.019049	0.051380
256	0.001265	0.002618	0.005172	0.009728	0.019507	0.069278

Table 6.23(a): Execution time (in seconds) for *compute*( ) (*PE and WF, double-precision, naive*) on C1060

Threads ↓, Atoms→	256	512	1024	2048	4096	8192
32	0.004972	0.009899	0.021607	0.051502	0.146496	0.533078
64	0.005947	0.011704	0.022267	0.060912	0.168887	0.533464
128	0.008547	0.016736	0.032442	0.062551	0.217189	0.636170
256	0.015025	0.029582	0.057144	0.111947	0.222376	0.847532

Table 6.24(a): Execution time (in seconds) for *compute*( ) (*PE and WF, mixed-precision, naive*) on C1060

Threads ↓, Atoms→	256	512	1024	2048	4096	8192
32	0.001039	0.002314	0.004106	0.007949	0.017325	0.069671
64	0.001061	0.002382	0.004097	0.008516	0.018564	0.050180
128	0.001191	0.002589	0.004677	0.008285	0.021340	0.059456
256	0.001487	0.003369	0.005862	0.011241	0.021370	0.078658

Table 6.21(b): Execution time (in seconds) for *compute*( ) (*PE and WF, single-precision, opt*) on C870

Threads ↓, Atoms→	256	512	1024	2048	4096	8192
32	0.001048	0.002368	0.004842	0.008340	0.022370	0.066528
64	0.001081	0.002393	0.004819	0.008468	0.018053	0.051345
128	0.001124	0.002496	0.005027	0.008607	0.019002	0.054212
256	0.001177	0.002865	0.005935	0.010495	0.021105	0.058099

Table 6.22(b): Execution time (in seconds) for *compute*( ) (*PE and WF, single-precision, opt*) on C1060

Threads ↓, Atoms→	256	512	1024	2048	4096	8192
32	0.000935	0.001980	0.003849	0.007193	0.014295	0.033531
64	0.000937	0.002008	0.004092	0.007226	0.014458	0.032098
128	0.001003	0.002100	0.004161	0.007637	0.015001	0.033896
256	0.001072	0.002403	0.005092	0.009486	0.018551	0.038702

Table 6.23(b): Execution time (in seconds) for *compute*( ) (*PE and WF, double-precision, opt*) on C1060

Threads ↓, Atoms→	256	512	1024	2048	4096	8192
32	0.004710	0.009655	0.018376	0.039585	0.098108	0.312483
64	0.005523	0.011343	0.021461	0.042927	0.106487	0.306886
128	0.007307	0.015654	0.030889	0.060608	0.127064	0.354782
256	0.009292	0.023907	0.051325	0.105503	0.214339	0.446943

Table 6.24(b): Execution time (in seconds) for *compute*( ) (*PE and WF, mixed-precision, opt*) on C1060

Threads ↓, Atoms→	256	512	1024	2048	4096	8192
32	0.000956	0.002470	0.004150	0.007636	0.015276	0.036752
64	0.000996	0.002338	0.004105	0.007637	0.015530	0.035640
128	0.001056	0.002629	0.004534	0.008238	0.016280	0.037930
256	0.001167	0.003142	0.005535	0.010750	0.021085	0.044163

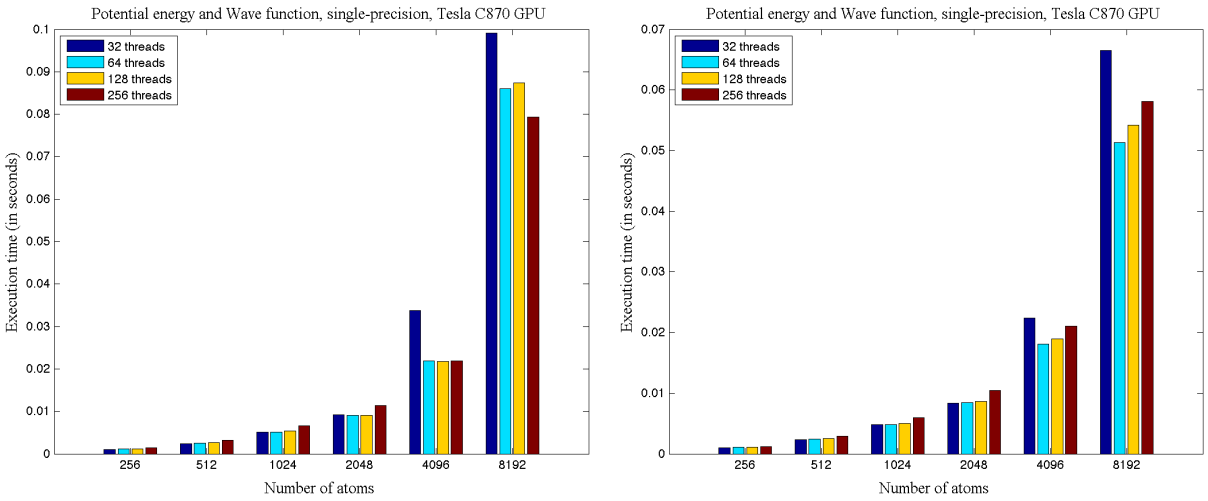

Figure 6.19: Execution time (in seconds) for varying number of threads for the *PE and WF kernel*s on *Tesla C870 GPU* (*single-precision*) (*Left*) Naïve implementation (*Right*) Optimized implementation

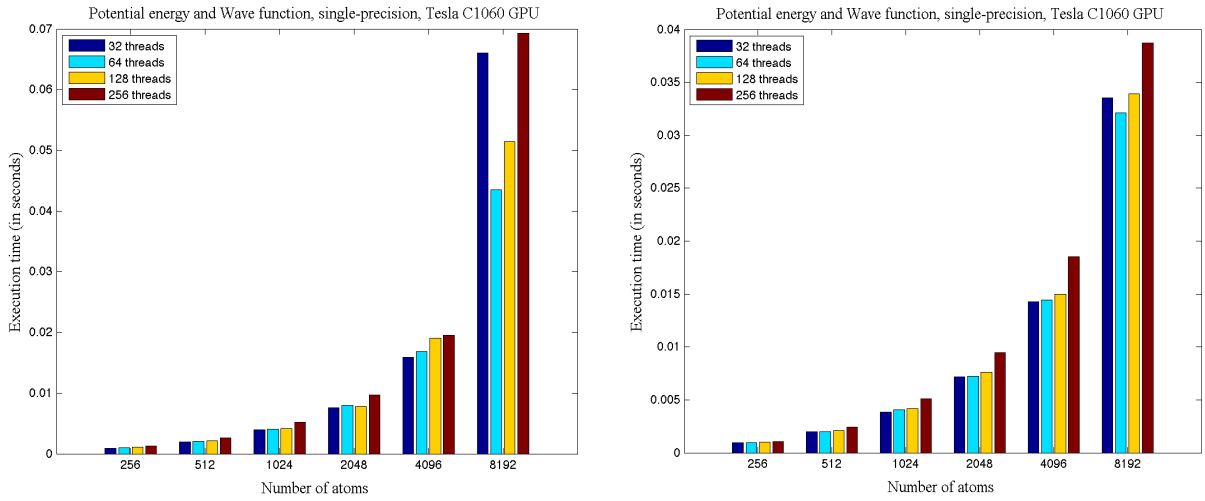

Figure 6.20: Execution time (in seconds) for varying number of threads for the *PE and WF kernels* on *Tesla C1060 GPU* (*single-precision*) (*Left*) Naïve implementation (*Right*) Optimized implementation

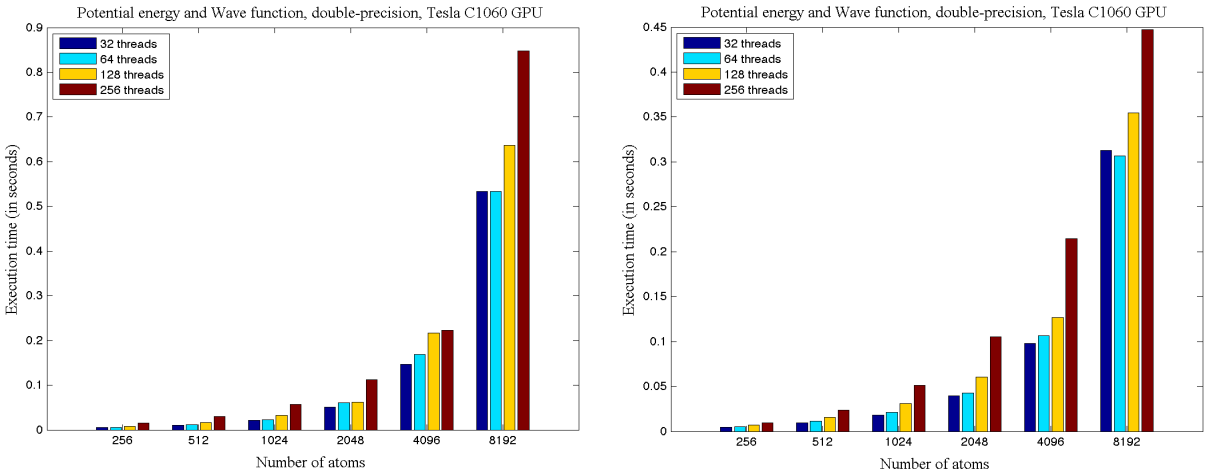

Figure 6.21: Execution time (in seconds) for varying number of threads for the *PE and WF kernels* on *Tesla C1060 GPU* (*double-precision*) (*Left*) Naïve implementation (*Right*) Optimized implementation

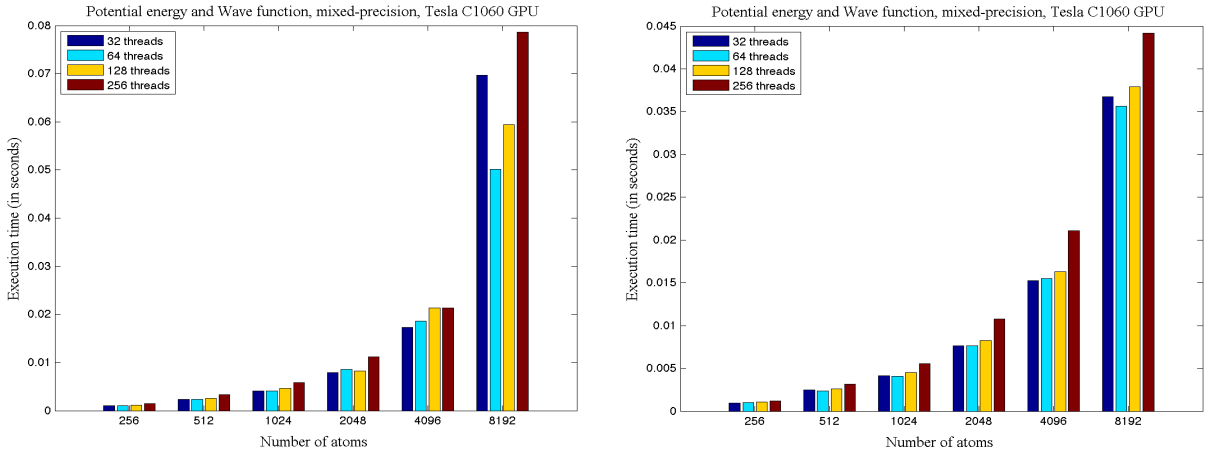

Figure 6.22: Execution time (in seconds) for varying number of threads for the PE and WF kernels on Tesla C1060 GPU (mixed-precision) (Left) Naïve implementation (Right) Optimized implementation

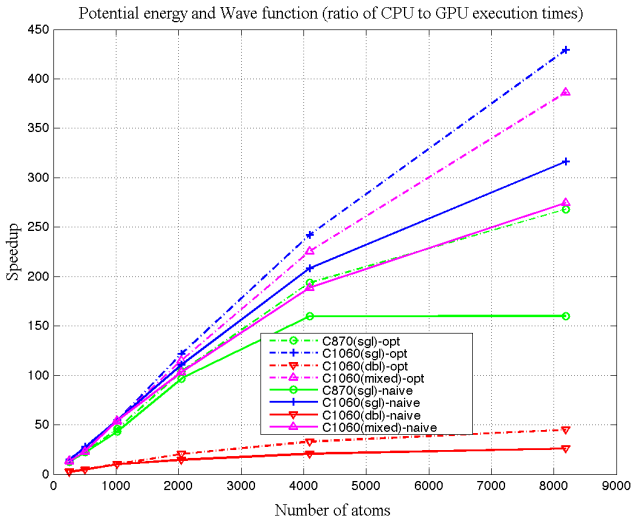

Figure 6.23: Speedup for the *CUDA PE and WF kernels* with *64 threads* (*optimized* and *naïve*) on the GPU and QMC on *Pacific* Cray XD1 2.2 GHz Opteron (*double-precision*) as the baseline

We present the results from a QMC simulation with 10 ensembles and a total of 400 iterations in Table 6.25. The results while using 32 and 64 threads in the naïve and optimized implementations with mixed-precision on the C1060 GPU are summarized in Table 6.25. Each of the optimized implementations has smaller run times over the corresponding naïve implementations. For small *N*, however, there is not much benefit. However, as *N* increases beyond 4096, the execution time of the optimized implementation is significantly reduced, with half the execution time in the case of *N* = 8192 with 64 threads. In Table 6.26, we present the results from the MPI implementation of the QMC algorithm where we use 120 ensembles and divide them across two C1060 GPUs on *Tesla*. Each GPU works on its set of ensembles and the results are sent to the master process from the other GPU at the end of the algorithm. As seen from Table 6.26, using two GPUs, we obtain a speedup of 1.8*x* over a single GPU for our application. We will assess the numerical accuracy delivered by the floating-point representations on the GPU (single-, double-, and mixed- precision versus the double-precision on the CPU) in the following section. For the present discussion, we assume that the mixed-precision representation, which provides the best speedup performance next to single-precision, as shown in Figure 6.23, also delivers the required numerical accuracy for our application. In Figure 6.24, we compare the speedup of the optimized implementations

using mixed-precision, with 32 and 64 threads, respectively on the C1060 GPU (data presented in Table 6.25) versus the software QMC on *Pacific* using double-precision. We also plot the speedup of the fixed-point FPGA implementation for the same simulation parameters. We show our results up to $N = 4096$ (the maximum number of atoms that can be presently simulated using the FPGA implementation). The optimized GPU implementations using mixed-precision provides the best speedup exceeding 225*x* for $N = 4096$ over the *Pacific* CPU implementation. The FPGA fixed-point implementation shows a speedup of 40*x* over the CPU implementation. The GPU implementation outperforms the FPGA implementation by a factor of 5*x*. As seen in Section 7.2, the factors that dominate the hardware execution time on a single RC node are the FPGA clock frequency, as well as the capacity, i.e., the number of pipelines or processing elements that can concurrently operate on the FPGA. Hence, with a larger FPGA device (hence more pipelines) and an increased clock frequency, we should expect speedup improvement of the FPGA implementation. Following our discussion on the error performance on the GPUs, we will compare these platforms for our application, and propose ways of combining them effectively to map different portions of our application onto these platforms.

Table 6.25: Execution time (seconds) the QMC algorithm with ($E = 10$, $I_{EQ} = 200$, $I_{SS} = 200$) (*mixed-precision*) on Tesla C1060 for the naïve and optimized implementations

Implementation ↓, Atoms→	64	128	256	512	1024	2048	4096	8192
Naïve: 32 threads	1.27294	2.51037	4.76282	12.3717	16.8471	32.7499	71.0266	297.686
Naïve: 64 threads	1.30285	2.22170	4.04569	9.5716	16.8394	35.6336	79.9690	300.521
Opt: 32 threads	1.21485	2.05461	3.77917	9.06793	15.9223	30.6861	61.1407	155.903
Opt: 64 threads	1.23098	2.11839	3.92078	9.30856	16.3868	34.4427	62.2520	143.350

Table 6.26: Execution time (seconds) of the QMC MPI Implementation on *Tesla* machine with two C1060 GPUs (optimized, 64 threads, *mixed-precision*) ($E = 120$, $I_{EQ} = 200$, $I_{SS} = 200$)

Function ↓, Atoms→		64	128	256	512	1024	2048	4096
PE and WF	*nprocs* = 1	14.6556	25.3745	46.9944	111.5926	198.1403	367.9782	746.9485
	nprocs = 2	8.32221	13.7645	24.9424	66.3183	108.9617	197.4356	398.3458

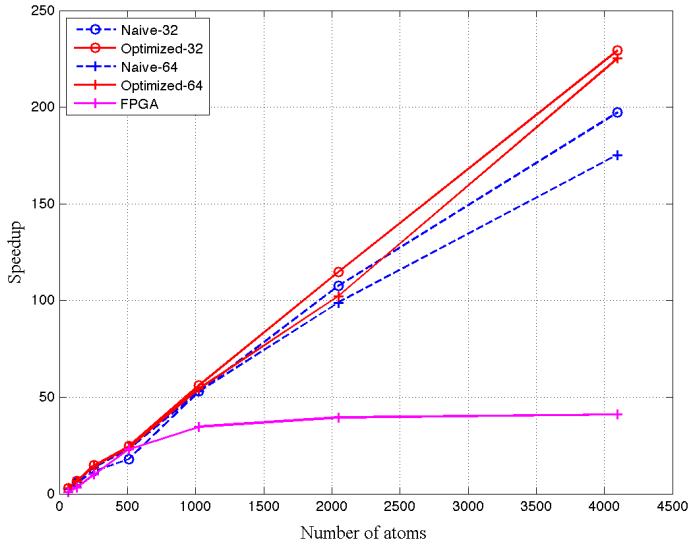

Figure 6.24: Comparison of FPGA (*fixed-point*) and GPU (32- and 64- threads, *mixed-precision*) versus the *double-precision* CPU implementation on *Pacific* Cray XD1 2.2 GHz Opteron

We observe from Figure 6.23 that the single-precision GPU implementation on Tesla C1060 provides the best speedup, followed by the mixed-precision implementation. While a platform and the various implementations on the platform might exhibit a good speedup performance, inspecting the accuracy it delivers to produce results that are scientifically meaningful, acceptable, and interesting, will help us better assess its suitability for an application. We plot the relative errors of the single-precision and mixed-precision GPU implementation of the QMC application in Figure 6.25. We observe that the single-precision pair-wise potentials have a relative error performance of 1.4×10^{-5} (for N = 8192) to the double-precision CPU implementation. The pair-wise potentials in mixed-precision show a relative error performance of 0.85×10^{-7} (for N = 8192). In this case, the function evaluations are done in single-precision and the in-place reductions are done using double-precision on the GPU. The final reductions of the O(N) potential energies and wave function values on the host are done in double-precision. In Figure 6.26, we plot the relative error performance of the double-precision GPU implementation versus the double-precision CPU implementation. The double-precision GPU implementation has relative errors of 1.4×10^{-11} for a cluster with 8192 atoms. But this also costs us 4x more execution time to get this level of

error performance. For large atomic clusters, the ground-state energies are not known experimentally and hence the mixed-precision error performance is reasonably acceptable and within the tolerance limit to compute the energies. Another advantage with this implementation is that the parameters for the wave function that yield the ground-state energies are not known and one typically optimizes the parameters during the course of the simulation. Hence, we could use a mixed-precision approach to experiment with the parameters and subsequently fine-tune the parameters using a higher numerical precision.

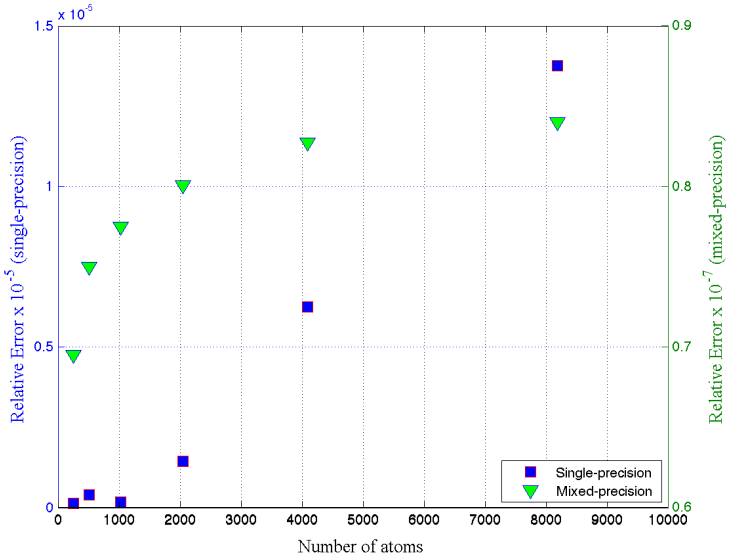

Figure 6.25: Relative Error (pair-wise potential) *single-precision* and *mixed-precision* on C1060 GPU compared to *double-precision* on *Pacific* Cray XD1 2.2 GHz Opteron

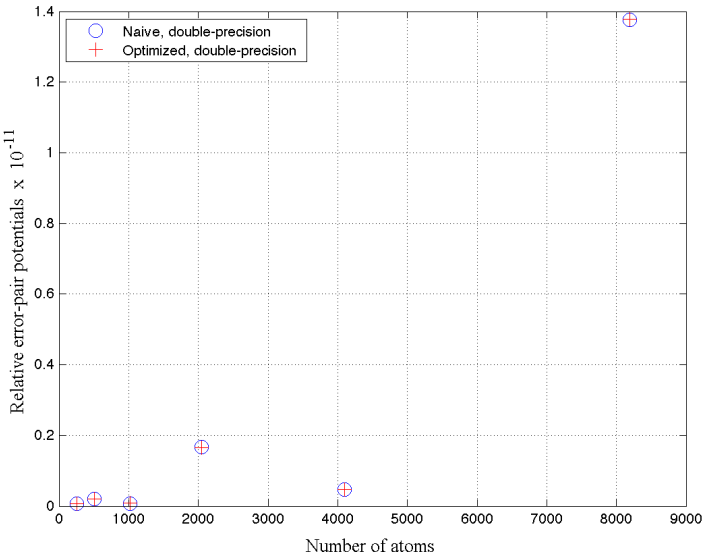

Figure 6.26: Relative Error (pair-wise potential) *double-precision* on C1060 GPU compared to *double-precision* on the *Pacific* Cray XD1 2.2 GHz Opteron

In this chapter, we compare individual CPU, and FPGA, and GPU platforms (FPGAs and GPUs are used as hardware accelerators), for the QMC application. We demonstrate the effectiveness of each of the platforms, by providing the speedup and error performance. Of the implementations, QMC on CPU required the least design effort. The GPU implementation required a modest amount of effort, with an initial learning curve for CUDA programming. The FPGA design took the highest amount of development time and effort, both for programming FPGAs using a hardware-description language as well as for placing-and-routing the design to generate the design bitstream that is downloaded to the FPGAs. Our approach significantly reduces the FPGA design effort by developing a general-purpose framework that allows us to reuse our design to calculate a variety of functions of interest [118]. Hence, while this involved a considerable amount of time and effort initially, implementing additional functions is trivial using the interpolation framework. On the GPUs, a non-trivial programming effort is involved to optimize the calculation of analytical functions and orchestrate device memory accesses appropriately. Comparing the costs of these platforms, present day multi-core processors cost a few hundred dollars; high-end GPUs cost a few thousand dollars and it costs tens of thousands of dollars for a HPRC FPGA

platform. Given the differences in cost and the programming effort associated with these platforms, the question arises if a platform is suitable for our application and is it justified in terms of the price-performance ratio and the development time and effort while using the platform. In the following chapter, we develop analytical performance models to understand how to best map the application to the computational resources available on these platforms. The models can help us predict and characterize the performance on various platforms, to better understand the suitability of a platform. If it is indeed justified in terms of performance, to invest in one or more of these platforms, and assuming a hybrid computing platform that combines different architectures exists, performance models are extremely useful to identify the ways of partitioning application, so we can exploit the different levels of parallelism available with these architectures.

7. PERFORMANCE MODELING

Performance models provide better insight into the application and help us understand, analyze and assess its performance (and bottlenecks) on various platforms. They can also be used for optimization by exploring the best ways of partitioning, mapping, and scheduling the code. Performance models can also be used for predicting and characterizing application performance with different architectures to assess their effectiveness.

In this chapter, we discuss our analytical performance models for individual CPU, FPGA, and GPU platforms. We develop the CPU performance model by expressing the execution time components in terms of the problem size and model parameters derived from profiling the application. We develop a clock cycle accurate model for the execution time on FPGAs. We also use this model to predict the execution time on related reconfigurable platforms. Developing an accurate performance model on the GPUs is difficult due to the hardware limitations as well as constraints imposed by the programming model. Unlike traditional microprocessors, the low-level architectural details are not available, which makes it harder to understand the resources that are available.

7.1. CPU Performance Model

The simulation parameters of the Quantum Monte Carlo (QMC) algorithm are given below:

- Problem size, which is the number of atoms = N
- Number of ensembles = E

- Total number of iterations, which is the sum of equilibration iterations (I_{EQ}) and the steady-state iterations (I_{SS}), given by $I = I_{EQ} + I_{SS}$

Figure 7.1 shows the execution steps of the QMC algorithm on CPU. The right side of the flow chart shows the computational complexity of each step of the algorithm. The left side of the flow chart shows the number of times (iterations*ensembles) each step is performed. As seen from the flow chart, the CPU execution time consists of the following components:

- The time for initializing the scalable parallel random number generator (SPRNG), which does not depend on problem size is denoted by t_{rng}. This time is set equal to K_{rng} as shown in Equation 7.1. The value of K_{rng} is derived from profiling the application.

$$t_{rng} = K_{rng} \tag{7.1}$$

- The time for initializing the reference configuration, which depends on *N*, is denoted by t_r. This is done 3**N* times corresponding to the (*x*, *y*, *z*) co-ordinate positions as shown in Equation 7.2. The value of K_r is derived from profiling.

$$t_r = 3 * N * K_r \tag{7.2}$$

- The time for obtaining the perturbed configuration, which depends *N*, is denoted by t_p. We invoke the *sprng*( ) function once for each of the co-ordinate positions of *N* atoms, for a total of 3**N* times. We obtain the value of K_p from profiling the application.

$$t_p = 3 * N * K_p \tag{7.3}$$

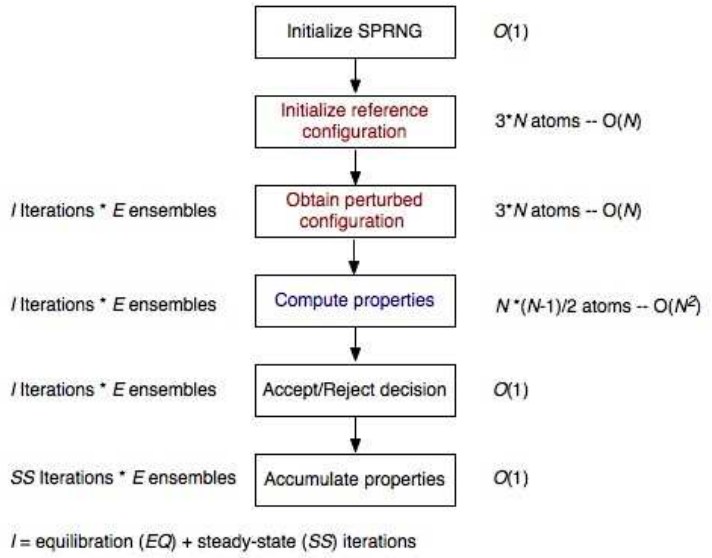

Figure 7.1: Execution steps of the QMC algorithm on CPU

- The time for computing the co-ordinate distances between the atoms and the ground-state properties (potential energy, kinetic energy, and trial wave function) depends on N and is O(N^2) in the number of atoms. This component forms the bulk of the execution time and is given by t_c. The constants K_{c1}, K_{c2}, K_{c3} and K_{c4} correspond to the contributions from distance calculation, potential energy, kinetic energy, and trial wave function calculations to t_c.

$$t_c = N*(N-1)/2*\left(K_{c1}+K_{c2}+K_{c3}+K_{c4}\right) \tag{7.4}$$

For our analysis, we ignore the trivial contributions from the times to decide whether to accept a configuration and the times for accumulating the energies for I_{ss} steady-state iterations, which do not depend on the problem size, N. Hence, the total time for the QMC algorithm on the CPU, t_{CPU}, is given by Equation 7.5.

$$t_{CPU} = t_{rng} + t_r + t_c + \sum_{i<I}\sum_{j<E}\left(t_p + t_c\right) \tag{7.5}$$

Table 7.1 describes the parameters used in the CPU model and lists their values derived from profiling. The measured run times for the various components in Equation 7.5 are averaged over 10 runs. The parameter, K_{rng}, is set to the average execution time for initializing the Scalable Parallel Random Number Generator (SPRNG), t_{rng}. Table 7.2 compares the actual execution times and the predicted results from the model (in seconds) for initializing the reference configuration, t_r, as *N* is varied from 256 to 8192 (in powers of 2). Table 7.3 compares the measured and predicted results for the times, t_p, to perturb the configuration to obtain the proposed configuration. Table 7.4 compares the measured and model results for the execution times for a single compute function call, t_c, for individual components such as *SAPT2-He* potential energy, trial wave function, and kinetic energy. We observe that the model error is below 3% for potential energy and below 1% for wave function and kinetic energy calculations. Figure 7.2 compares the measured and model execution times for a single compute function call with PE, WF, and KE calculations. In Table 7.5, we list the experimental results for a complete QMC simulation with potential energy and trial wave function calculations, when we have 10 ensembles and run the simulation for a total of 400 iterations, consisting of 200 equilibration and 200 steady-state iterations. We use the model parameters obtained from the profiling of various components to obtain the execution times for the overall simulation. Comparing the actual and model results, we observe a model error of less than 5% as we vary the number of atoms in the simulation. The kinetic energy calculations are also included in the QMC simulation and we show the measured and model results in Table 7.6 for the same simulation parameters used in Table 7.5. The model error between the average execution times and predicted times is less than 2%. These results are plotted in Figure 7.3. We also use the model to predict the execution times for a cluster with 8192 and 16384 atoms as shown in Tables 7.5, 7.6, and Figure 7.3.

Table 7.1: CPU model parameters learnt from profiling
(on *Pacific* Cray XD1 2.2 GHz AMD Opteron)

Constant	Constant description	Value from Profiling
K_{c1}	Distance calculation parameter	1.0990e-08
K_{c2}	Potential energy parameter	6.3127e-07
K_{c3}	Wave function parameter	3.3384e-07
K_{c4}	Kinetic energy parameter	6.0315e-07
K_{rng}	Initialization of SPRNG	0.00227
K_r	Initialization of reference configuration	5.7168e-09
K_p	Perturbation of the configuration	2.4202e-08

Table 7.2: Measured and predicted execution times (in seconds) to initialize a reference configuration, t_r (on *Pacific* Cray XD1 2.2 GHz AMD Opteron)

Execution time (seconds) ↓, Atoms→	256	512	1024	2048	4096	8192
Experimental results	3.09944e-06	1.01566e-05	1.6785e-05	3.21388e-05	7.06196e-05	0.000139570
Model results	4.3903e-06	8.7806e-06	1.7561e-05	3.5123e-05	7.025e-05	0.0001405
Model error (%)	41.6638	13.5441	4.6289	9.2882	0.5265	0.6632

Table 7.3: Measured and predicted execution times (in seconds) to obtain the perturbed configuration, t_p (on *Pacific* Cray XD1 2.2 GHz AMD Opteron)

Execution time (seconds) ↓, Atoms→	256	512	1024	2048	4096	8192
Experimental results	2.09808e-05	3.88622e-05	7.70092e-05	1.55926e-04	3.06845e-04	5.9891e-04
Model results	1.8587e-05	3.7174e-05	7.4348e-05	1.4870e-04	2.9739e-04	5.9479e-04
Model error (%)	11.4	4.34	3.45	4.64	3.08	0.688

Table 7.4: Measured and predicted execution times (in seconds) for a single compute function call, t_c (on *Pacific* Cray XD1 2.2 GHz AMD Opteron)

Execution time (seconds) ↓, Atoms→	256	512	1024	2048	4096	8192
Measured results (*Potential energy*)	0.020445	0.081886	0.327832	1.31519	5.26114	21.6459
Model results (*Potential energy*)	0.019059	0.076387	0.305847	1.22399	4.89715	19.5910
Model error (%)	2.5%	2.6%	2.5%	2.3%	2.6%	0.16%
Measured results (*Trial Wave Function*)	0.011187	0.044868	0.180072	0.720045	2.88660	11.5707
Model results (*Trial Wave Function*)	0.011255	0.045109	0.180614	0.722810	2.89195	11.5692
Model error (%)	0.6%	0.5%	0.3%	0.4%	0.2%	0.01%
Measured results (*Kinetic Energy*)	0.019963	0.079729	0.319957	1.28376	5.14606	20.6061
Model results (*Kinetic Energy*)	0.020046	0.080339	0.321673	1.28732	5.15055	20.6047
Model error (%)	0.4%	0.8%	0.6%	0.3%	0.09%	0.007%

Table 7.5: Measured and predicted results of total CPU time (in seconds), t_{CPU} (for *SAPT2* PE and WF calculations) $E = 10,\ I_{EQ} = I_{SS} = 200$ (on *Pacific* Cray XD1 2.2 GHz AMD Opteron)

Execution time (seconds) ↓, Atoms→	256	512	1024	2048	4096	8192	16384
Measured results	130.055	521.065	2148.11	8285.27	33185.4	--	--
Model results	127.549	511.039	2045.85	8186.78	32753.9	131029.2	524144.0
Model error (%)	1.97	1.92	4.8	1.2	1.3	--	--

Table 7.6: Measured and predicted results of total CPU time (in seconds), t_{CPU} (for *SAPT2* PE, KE, and WF calculations) $E = 10,\ I_{EQ} = I_{SS} = 200$ (on *Pacific* Cray XD1 2.2 GHz AMD Opteron)

Execution time (seconds) ↓, Atoms→	256	512	1024	2048	4096	8192	16384
Measured results	203.2685	812.2389	3253.528	13071.15	52168.93	--	--
Model results	206.3165	826.7262	3309.831	13245.18	52992.47	211993.33	848020.28
Model error (%)	1.5%	1.8%	1.7%	1.3%	1.6%	--	--

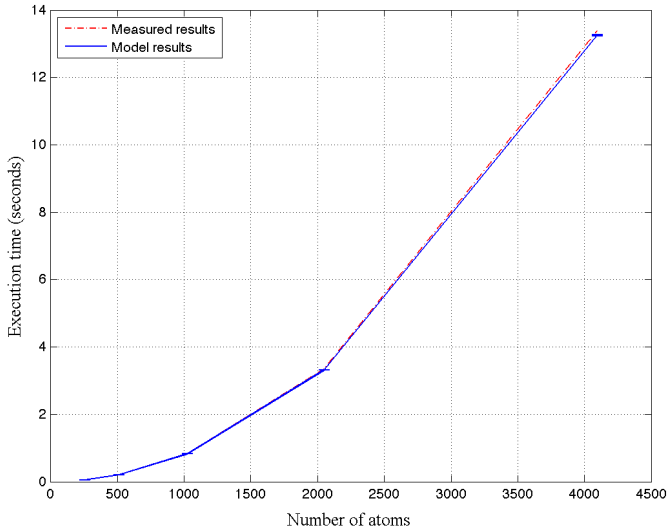

Figure 7.2: Measured and model execution times (in seconds) for a *compute*( ) call on *Pacific* XD1 2.2 GHz Opteron with *SAPT2* PE, WF, and KE calculations

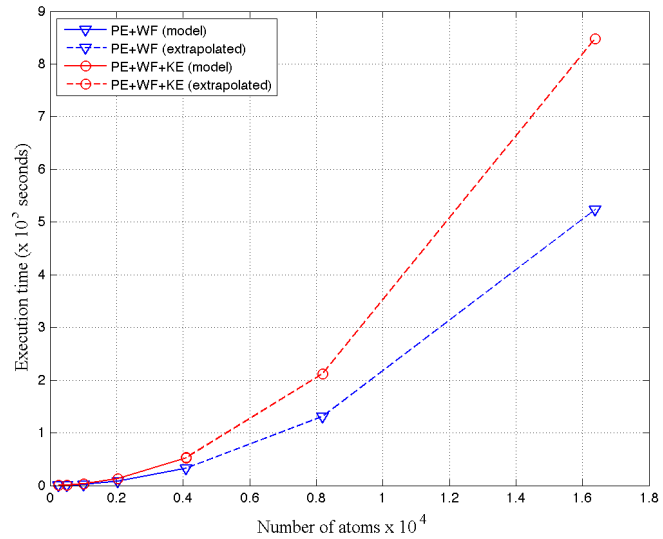

Figure 7.3: Execution times (in seconds) for a complete QMC simulation on *Pacific* XD1 2.2 GHz Opteron (E = 10, I_{EQ} = 200, I_{SS} = 200) (extrapolated for large clusters)

7.2. Performance Metrics

Commonly used performance metrics in high performance computing are speedup and efficiency. Amdahl's law [119] for "fixed-size" speedup and Gustafson's law [120] for "scaled-speedup" provide the theoretical performance limits of parallelization. We use the speedup metric to compare the performance of heterogeneous parallel systems investigated in this dissertation over a single microprocessor implementation. Speedup is given by the ratio of the execution time of the best serial algorithm on a single microprocessor, T_{serial} to the execution time on a parallel system, $T_{parallel}$, as shown in Equation 7.6 [121].

$$Speedup = \frac{T_{serial}}{T_{parallel}} \tag{7.6}$$

Efficiency is another useful parallel metric for parallel computing performance. It is given by the ratio of speedup to the number of processing elements, *p*, as shown in Equation 7.7 [121].

$$Efficiency = \frac{Speedup}{p} = \frac{T_{serial}}{T_{parallel} \cdot p} \tag{7.7}$$

7.3. RC Single Node Model

We begin our performance analysis for a single reconfigurable computing node on the Cray XD1 executing a synchronous iterative algorithm (SIA). This will allow us to investigate the interactions between the Opteron and the FPGA acceleration processors. We start with the performance model proposed for a class of fork-join algorithms, called synchronous iterative algorithms (SIA) [105, 122]. We assume that we have a program segment, which has *I* iterations, which are statistically identical with respect to run times. The RC node could have multiple FPGAs and processors. This is especially true for current FPGA-based supercomputers, which use multi-core processors and multiple FPGAs. For example, each computing node on the Cray XD1 consists of a single FPGA connected to a dual-core dual-Opteron.

Hence, the program segment can be partitioned into hardware and software tasks, which execute concurrently on the FPGA and the processor(s).

Equation 7.8 gives the total time for the SIA running on a single RC node.

$$t_{SIA} = \sum_{i=1}^{I} \left[t_{serial,i} + \max(\max_{1 \le j \le m} (t_{sw,j,i}), \max_{1 \le j \le n} (t_{hw,j,i})) + t_{comm,i} \right] \tag{7.8}$$

For an iteration i, $t_{serial,i}$ denotes the serial execution time that cannot be accelerated. The program segment is partitioned into tasks that run concurrently in software on m processors, given by $t_{sw,j,i}\,(1 \le j \le m)$, and tasks that run on hardware on n FPGAs, given by $t_{hw,j,i}\,(1 \le j \le n)$. The communication time is denoted as $t_{comm,i}$. The overheads such as configuration of the FPGA and initialization will be included in the final RC execution time.

To simplify the analysis, we assume that each iteration requires statistically the same amount of run time and hence we focus on a typical iteration. Each term is modeled as a random variable and hence replaced using its expected value. We also assume that each of the random variables is independent and identically distributed. These assumptions allow us to rewrite Equation 7.8 as,

$$t_{SIA} = I \left[E(t_{serial,i}) + E\left[\max(\max_{1 \le j \le m} (t_{sw,j,i}), \max_{1 \le j \le n} (t_{hw,j,i})) \right] + E\left[t_{comm,i} \right] \right] \tag{7.9}$$

We define t_{serial} as the expected value of $t_{serial,i}$ and t_{comm} as the expected value of $t_{comm,i}$. Hence Equation 7.9 can be now be written as,

$$t_{SIA} = I \left[t_{serial} + E\left[\max(\max_{1 \le j \le m} (t_{sw,j,i}), \max_{1 \le j \le n} (t_{hw,j,i})) \right] + t_{comm} \right] \tag{7.10}$$

The hardware tasks are deterministic, i.e., only depend on clock frequency and the problem size, *N*. Hence, we can obtain this term accurate to the number of clock cycles. We also assume that the *n* hardware tasks are identical with respect to run times, and that the software and hardware tasks do not overlap, allowing us to replace the time for hardware tasks with the mean value, t_{HW}. The software time depends on the speed of the microprocessor, and the computational load on the system. For our analysis, we assume a dedicated system and replace the software task completion time by the expected value, t_{SW}. Equation 7.11 gives the total time for the SIA on a single RC node.

$$t_{SIA} = I\left[t_{serial} + t_{SW} + t_{HW} + t_{comm}\right] \tag{7.11}$$

We introduce Equation 7.11 in our final analysis to obtain the total execution time on a single reconfigurable computing node, t_{RC}. We define $t_{overhead}$ as the time for configuring the FPGA with the bitstream, $t_{configure}$, and for initialization of parameters that are used for FPGA control, denoted as t_{init}, as shown in Equation 7.12. t_s is the time for the serial components of the program including the time to initialize the random number generator, t_{rng}, and the time to initialize the reference configuration, t_r. The serial time, t_{serial}, in each iteration is the mean time for perturbing a configuration, t_p. In our implementation, the software and hardware tasks do not overlap and for our model, we ignore the small amount of time taken to reconstruct the floating-point results from the FPGA fixed-point results. Hence we remove the term, t_{SW}, from our analysis. The execution time on a single RC node, t_{RC}, is given by Equation 7.13. Using our earlier definitions for terms in Equation 7.13, we write Equation 7.14.

$$t_{overhead} = t_{configure} + t_{init} \tag{7.12}$$

$$t_{RC} = t_{overhead} + t_s + t_{HW} + I * E\left(t_{serial} + t_{comm} + t_{HW}\right) \tag{7.13}$$

$$t_{RC} = t_{configure} + t_{init} + t_{rng} + t_r + t_{HW} + I * E\left(t_p + t_{comm} + t_{HW}\right) \quad (7.14)$$

We will describe the contributions to the execution time in Equation 7.14 when using an FPGA to accelerate the algorithm. The parameters are defined in Table 7.7. Equation 7.15 gives the time for initializations of the interpolation coefficients on the FPGA. Equation 7.16 gives the time for the hardware tasks (in our implementation, all hardware tasks finish in the same clock cycle). The time for loading the co-ordinate positions, t_{comm}, is given by Equation 7.17.

$$t_{init} \approx \frac{N_c * b}{BW} \quad (7.15)$$

$$t_{HW} = \frac{L + \left(N(N-1)/2\right)}{f} \quad (7.16)$$

$$t_{comm} \approx \frac{3 * N * b}{BW} \quad (7.17)$$

Table 7.7: Description of FPGA parameters

m	Number of hardware tasks
f	Clock frequency for the design implemented on the FPGA
N_c	$3 * m * (f_I + f_{II})$ f_I - Number of region I coefficients f_{II} - Number of region II coefficients
L	Latency of the pipeline (=58)
b	Number of bits (=64)
BW	Bandwidth of the CPU-FPGA interconnect in bits/second

Table 7.8 gives the time to transfer the co-ordinate positions, t_{comm}, from the host memory to the on-chip memory which are done using the Cray API functions within the software application. The first row gives the measured times to transfer the positions, when the design on the FPGA performing only data transfers operates at 199 MHz (which corresponds to a RapidArray bandwidth of 1.6 GBytes/second). We provide the results from the model for a clock frequency of 199 MHz and the transfer times for the actual frequency of 100 MHz for the complete FPGA design (corresponding to half the RapidArray bandwidth). This time forms a small component of the total time for the QMC algorithm as we can see from Tables 7.9 and 7.10, where we report the execution times of the QMC algorithm with PE calculations, followed by PE and WF calculations, and the model error is under 5%. The times for PE, and PE and WF calculations are similar with a small additional overhead incurred for reading the results and reconstructing the floating-point results when the wave function calculations are also included. Figure 7.4 compares the measured and model results for the PE and WF calculations for N = 512, 1024, 2048, and 4096, and as per our discussion, we can see that the measured execution times agree with the times predicted by the model.

Table 7.8: Communication overhead (in seconds), t_{comm}, on *Pacific* Cray XD1 2.2 GHz AMD Opteron

Execution time (seconds) ↓, Atoms→	64	128	256	512	1024	2048	4096
Measured Results (199 MHz)	0.954×10^{-6}	2.289×10^{-6}	4.648×10^{-6}	8.583×10^{-6}	1.788×10^{-5}	3.576×10^{-5}	6.864×10^{-5}
Model Prediction (199 MHz)	0.894×10^{-6}	1.788×10^{-6}	3.576×10^{-6}	7.680×10^{-6}	1.431×10^{-5}	2.861×10^{-5}	5.722×10^{-5}
Measured Results (100 MHz)	1.192×10^{-6}	2.718×10^{-6}	5.721×10^{-6}	9.297×10^{-6}	2.433×10^{-5}	6.867×10^{-5}	11.445×10^{-5}

Table 7.9: Execution times (in seconds) of the complete QMC algorithm (PE calculations) on *Pacific* Cray XD1 2.2 GHz AMD Opteron with Virtex-4 FPGA (E = 10, I = 400 iterations)

Execution time (seconds) ↓, Atoms→	64	128	256	512	1024	2048	4096
Measured Results	3.9556	4.2639	5.2011	9.3111	25.4630	88.8541	341.8406
Model Prediction	3.9077	4.1747	5.2003	9.2180	25.1198	88.3884	340.7863
Model Error (%)	1.21	2.09	0.02	0.11	1.35	0.524	0.308

Table 7.10: Execution times (in seconds) of the complete QMC algorithm (PE and WF calculations) on *Pacific* Cray XD1 2.2 GHz AMD Opteron with Virtex-4 FPGA (E = 10, I = 400 iterations)

Execution time (seconds) ↓, Atoms→	64	128	256	512	1024	2048	4096
Measured Results	4.1086	4.3337	5.4509	9.6142	25.5955	89.2015	342.0629
Model Prediction	3.9078	4.1747	5.2003	9.2180	25.1198	88.3885	340.7863
Model Error (%)	4.89%	3.67%	4.60%	4.12%	1.86%	0.911%	0.373%

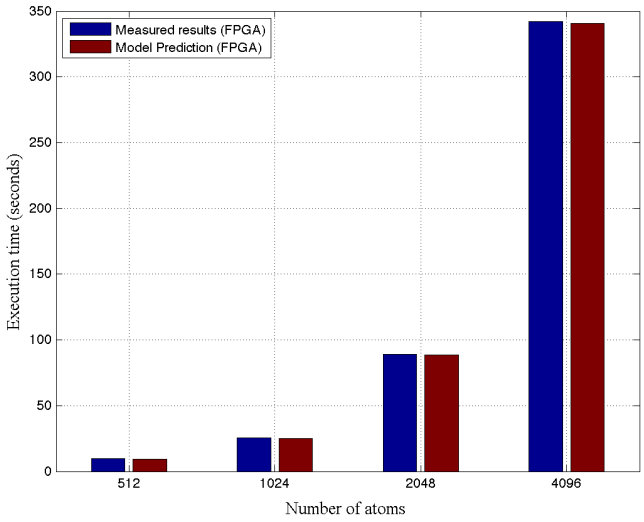

Figure 7.4: Comparison of execution times (in seconds) from measurement and model (PE and WF) on the Pacific Cray XD1 Virtex-4 VLX160 FPGA for N = 512, 1024, 2048 and 4096

We use our model to characterize performance on other FPGA platforms. The memory bandwidth and the I/O interface play an important role while using accelerators. The FPGA needs to be constantly fed with data to keep the pipeline busy, which is constrained by the host processor's memory bandwidth. In our implementation, the data from the host memory is copied to the on-chip memory prior to the pipeline operation using the Cray XD1 API functions. Along with system architectures, APIs are also critical to application performance [123]. In our initial analysis, we will predict the performance for the following platforms: (a) an Amirix development FPGA platform with a Xilinx Virtex-II V2P30 FPGA [124] and connected to the host process via a PCI interface (500 Mbytes/sec), and (b) a DRC platform with a Xilinx Virtex-4 VLX200 FPGA [47] connected to the processor via Hypertransport providing 3.2 GBytes/sec (duplex per connection). The Amirix platform previously targeted in our work for application development on FPGAs uses a maximum frequency of 66 MHz for our design. We assume the frequency estimated by the place-and-route process (excluding the platform) of 140 MHz on the FPGA present on the DRC platform. We consider the number of hardware tasks or pipelines, $m = 2$, which includes PE and WF. We will extend our analysis later to consider additional processing elements or pipelines on a larger FPGA and how this impacts performance. Figure 7.5 shows the execution time predicted by the model for the Amirix and DRC platforms. We also plot our results from the Cray XD1 implementation, which operates at 100 MHz, and the projected execution time if the design were to operate at 199 MHz, the maximum FPGA clock frequency. Our Cray FPGA implementation is 1.5x faster than the Amirix implementation, which is also 2x slower than the projected DRC implementation. Here, we assume that the Amirix and DRC platforms provide the same level of API interfaces as the Cray XD1. However, any overhead incurred without efficient APIs would further impact performance.

We also target the design to the Virtex-5 SX240 FPGA [17] present on the Nallatech FPGA systems. We may recall from our earlier results that on the Virtex-4 FPGA, roughly 50 percent of the resources were used for the PE and WF calculations. On the Virtex-5 FPGA, roughly 10 percent of the DSPs and

BRAMs, and 28 percent of slices were used. PAR also estimates a frequency of 166 MHz for our design. Optimizing the resource usage to achieve the right balance between the usage of DSPs and slices should allow us to fit multiple pipelines or include kinetic energy calculations on the FPGA. The large number of 36-kbit BRAMs available on the Virtex-5 SX FPGA will also allow us to simulate clusters containing more than 4096 atoms. If we can improve the clock frequency to 200 MHz, we should expect a speedup of 2*x* over our present Cray XD1 implementation at 100 MHz as shown in Figure 7.5 (corresponding to the Cray XD1-199 MHz design) and a further improvement in speedup with the inclusion of multiple identical engines or additional functions on the Virtex-5 FPGA.

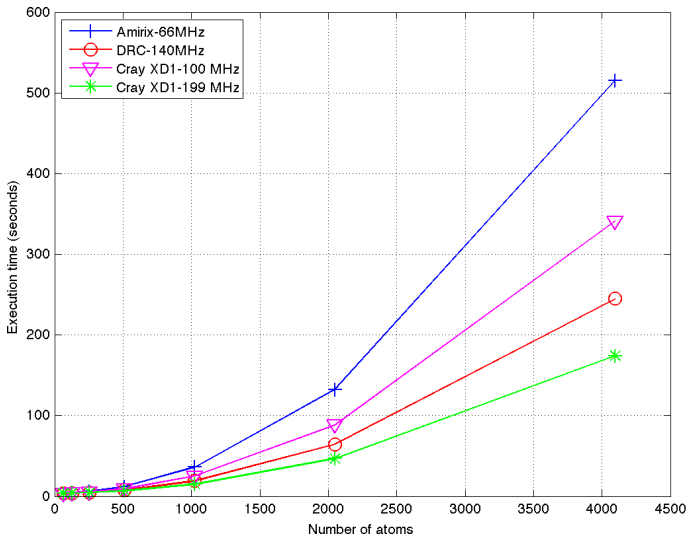

Figure 7.5: Comparison of projected execution times (in seconds) on various FPGA platforms

We extend the use of the performance model as a tool to predict performance, while exploiting additional functional parallelism, to map functions currently performed on the CPU to a larger FPGA device concurrently using our general interpolation framework. On a larger FPGA, we can fit the potential energy, trial wave function, and kinetic energy calculations on a single FPGA. In Figure 7.6, we compare the predicted results versus the CPU execution times, for a cluster of 4096 atoms. We refer to our CPU execution times on *Pacific* reported in Chapter 6 for 10 ensembles and simulated for 400 iterations. In our FPGA model, we account for the additional overhead incurred to load the interpolation coefficients for the kinetic energy evaluation. Hence, if we can fit the potential energy, trial wave function, and kinetic energy on a single FPGA using the general interpolation framework, we can expect the performance shown in Figure 7.6, which corresponds to a speedup of 100*x* over the CPU implementation. We can further extend the proposed model to predict the performance of systems with multiple RC nodes and take into account the contention for bandwidth by the multiple nodes.

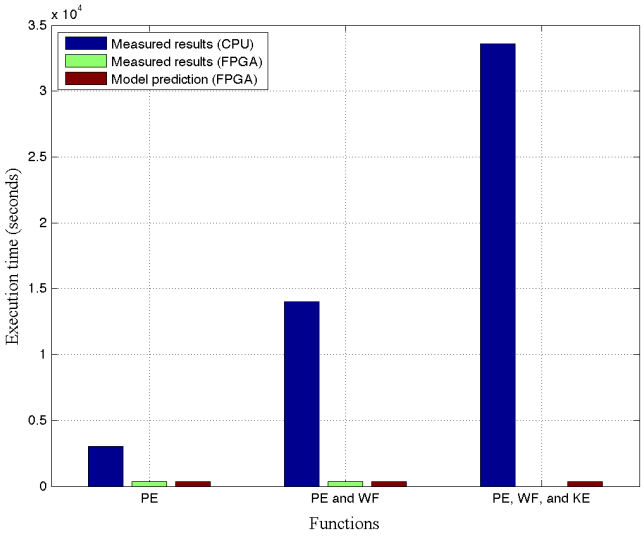

Figure 7.6: Comparison of measured CPU and FPGA execution times (on *Pacific* Cray XD1 Opteron 2.2 GHz with Virtex-4 FPGA) (in seconds) versus FPGA model results for a cluster of 4096 atoms

7.4. GPU Performance Model

We present a preliminary analytical performance model to predict the execution times of our application on NVIDIA GPUs while using the Compute Unified Device Architecture (CUDA) paradigm. Developing an accurate performance model is complicated due to the fact that one does not have access to all the low-level GPU architectural details. The optimization of memory accesses by the CUDA nvcc compiler makes it difficult to estimate parameters that could be used to model the execution time. Using the available hardware constraints, constraints from the programming model, and by benchmarking the application, we find the execution time of a block, t_p, with the number of threads set to the warp size of 32 threads. Here, we describe the model with the available hardware details of the C870 Tesla GPU. Within a warp, the threads are executed in parallel and the warp represents the smallest concurrent unit of execution in CUDA. For problem sizes, N = 64 to 512, the number of blocks with 32 threads are 2, 4, 8, and 16 respectively. There are a total of 16 multiprocessors (N_{SM} = 16) on the C870 Tesla. Multiple blocks (up to 8) can be scheduled on a streaming multiprocessor (SM), but only one warp is chosen for instruction fetch and execution at any instant of time. For our naïve QMC implementation on GPUs, we time slice the matrix using the number of threads. For example, when N = 64, we have two time slices along the y-dimension of the interaction matrix, with the x-dimension of the matrix divided into 2 blocks, each with 32 threads, which is equal to the warp size. To begin with, we consider the expression for the hardware time, t_{HW}, where we simply multiply the number of blocks, b, which represents the number of time slices in the y-dimension, with t_p as shown in Equation 7.18 (we assume the number of threads, T = 32, which is equal to the warp size).

$$t_{HW} = \frac{(T)*(b)*t_p}{(\text{Warp size})} \tag{7.18}$$

When N = 1024, the number of blocks is 32, which exceeds the number of multiprocessors. However, at most eight blocks can be scheduled per SM. Multiple blocks are scheduled per SM to have multiple active

warps thereby hiding the latencies of the pipelined instruction execution. When N = 4096, $b = N/T$ = 128. In this case, eight blocks are scheduled in each SM for a total of 128 blocks. Table 7.11 compares the measured results (averaged over 10 runs) and the results from the model for an iteration of potential energy calculations on the C870 GPU using single-precision. First, we focus on problem sizes up to N = 4096. We observe a model error under 15% for most problem sizes (N = 128 to 4096). When N = 8192, the model predicts an execution time that is roughly half of the actual execution time. The reason for this is as follows. For N = 8192 and T = 32, the number of thread blocks is 256. As described earlier, this would result in the hardware scheduling 128 blocks to the SMs, and once the resources are freed, the remaining 128 thread blocks are scheduled, which requires an additional time, t_p, for execution. The model in Equation 7.18 does not take into account this hardware constraint and hence we rewrite Equation 7.18 to include this additional constraint. Equation 7.19 now gives the hardware time, taking into account the hardware constraint, resulting in a model time (shown in bold) in Table 7.12 that is in good agreement (with less than 10% error) with the measured result for N = 8192.

$$t_{HW} = \left(1 + \left\lfloor \frac{b-1}{8 * N_{SM}} \right\rfloor\right) * t_p \tag{7.19}$$

where $t_{HW} = \lfloor \square \rfloor$ denotes the greatest integer function (*floor*)

Table 7.12 compares the measured and model results (using Equation 7.18) for one iteration of the potential energy and trial wave function calculations on the C870 GPU using single-precision as we vary the number of atoms. A model error of 12% and below is achieved for $N \leq 2048$. When N = 4096 and 8192, the model predicts execution times, which are half of the actual execution times with 50% model error as noted in Table 7.12. The reason this happens particularly with large N is most likely due to the memory latencies as N increases and more threads per block are needed to reduce the memory traffic. Using the information from the execution times by repeating our experiments with various sized blocks

on C870 and C1060, the proposed model will be improved to include parameters to account for the memory access behavior for large problem sizes. We will use the model parameter, t_p, to predict the time for the overall algorithm using Equation 7.13. In Equation 7.13, the expression for t_{comm} has the same form as in Equation 7.16. In this case, we include the time to copy the O(N) 32-bit potential energies and wave function values from device memory to host memory along with the time to copy the 3*N 32-bit positions from the host memory to the device memory. In the model, we set the PCI bandwidth to 2.5GBytes/sec (found by running the bandwidth tests for using page-locked memory in the CUDA SDK code samples [60]). For the CPU components, we use the CPU model parameters derived in Section 7.1. Table 7.13 compares the execution time of the complete QMC algorithm with PE and WF calculations for 10 ensembles and a total of 400 iterations. In this case, the model error is less than 15% for $N \leq 2048$. We observe similar model errors (50%) for N = 4096, and 8192. These results are obtained when 32 threads are used. When 64 threads or higher are used (i.e., the number of threads that yields the best performance for large N is used), we observe a model error of less than 20%.

Table 7.11: Measured and model execution times (in seconds) for one iteration of QMC (with potential energy) on *Ed* (with C870)

Execution time (seconds) ↓, Atoms→	64	128	256	512	1024	2048	4096	8192
Measured Results	0.000088	0.000143	0.000227	0.000446	0.000882	0.001812	0.004441	0.017472
Model Prediction (t_p=3.0875e-05)	0.000062	0.000124	0.000247	0.000494	0.000988	0.001976	0.003952	0.007904 / **0.015808**
Model Error (%)	29.8%	13.5%	8.81%	10.8%	12.0%	9.05%	11.0%	54.8% / **9.52%**

Table 7.12: Measured and model execution times (in seconds) for one iteration of QMC (with potential energy and wave function) on *Ed* (with C870)

Execution time (seconds) ↓, Atoms→	64	128	256	512	1024	2048	4096	8192
Measured Results	0.000275	0.000509	0.000990	0.001861	0.003742	0.007831	0.030901	0.122641
Model Prediction (t_p = 1.2063e-04)	0.000241	0.000483	0.000965	0.001930	0.003860	0.007720	0.0154	0.061762
Model Error (%)	12.4%	5.11%	2.53%	3.76%	3.21%	1.40%	50.16%	49.64%

Table 7.13: Measured and model execution times (in seconds) of QMC (with potential energy and wave function) on *Ed* (with C870), E = 10, I = 200+200 iterations

Execution time (seconds) ↓, Atoms→	64	128	256	512	1024	2048	4096	8192
Measured Results	1.5349	2.5289	4.4616	9.8364	20.9031	38.0294	137.118	500.221
Model Prediction	1.3254	2.3129	4.2878	9.1801	21.0296	36.8288	68.3815	249.243
Model Error (%)	13.65%	8.54%	3.90%	6.67%	0.61%	3.16%	50.12%	50.17%

8. CONCLUSIONS AND FUTURE WORK

Over the last few years, new architectures have emerged to keep up with the enormous performance demands in high performance computing and provide the computing power to tackle large complex scientific problems faster. Emerging architectures such as reconfigurable computing using field-programmable gate arrays (FPGAs), and graphics processing units (GPUs) show tremendous potential for scientific computing. These architectures promise the ability to explore exciting new questions and obtain previously unobservable properties in scientific computing. We investigate the use of FPGAs and GPUs to explore the viability of these platforms for scientific computing by considering the Quantum Monte Carlo (QMC) simulation method that is widely used in physics and physical chemistry. In this chapter, we summarize the contributions of this dissertation and highlight promising areas of future research.

8.1. Conclusions

Traditional reconfigurable computing (RC) platforms using field-programmable gate arrays (FPGAs) are shifting from PCI-based platforms to a variety of high performance reconfigurable computing (HPRC) platforms, which, with high-performance interconnect between the FPGA co-processor and the microprocessor, provide a tightly coupled system and friendly programming interfaces, to integrate FPGA acceleration into present scientific applications. Graphics Processing Units (GPUs) have evolved from fixed-function 3D graphics pipelines to flexible general-purpose computing engines, spawning a number of research efforts that apply GPUs in scientific computing. Numerical precision is an important concern in scientific computing. The latest FPGAs with increased gate density now provide the resources to

implement floating-point cores on the device, or the flexibility to use a customized precision according to application requirements. Present GPUs also offer the much-needed double-precision support and easy-to-use programming platforms, making them attractive platforms for scientific computing.

We develop a reconfigurable architecture for accelerating the computationally intensive functions of the QMC application such as potential energy and wave function. We target our architecture to a computing node on the Cray XD1 high performance reconfigurable computing platform, which consists of an AMD Opteron 2.2 GHz processor with a Xilinx Virtex-4 FPGA. Our implementation provides a speedup of 40*x* for the implementation of the *HFDB-He* potential energy and wave function, over the optimized double-precision CPU implementation (on a single core). We achieve a 100*x* speedup for the *SAPT2-He* potential energy and wave function over the CPU implementation. This speedup is attributed to the use of a pipelined architecture, and the use of fixed-point for all our calculations, which guarantees the accuracy required for our application. We also demonstrate that scaling our application to multiple nodes on the Cray XD1 provides a speedup linear in the number of FPGA equipped nodes compared to the use of a single FPGA node. We provide our implementation with the CPU implementation, and the hardware-accelerated version as an open-source framework that can be used both to study the energies of large clusters, and to experiment with the analytical functions of potential energies and wave functions to study a variety of clusters.

Next, we explore graphics processing units to accelerate the computationally intensive functions of the QMC application. We target two generations of NVIDIA GPUs, the Tesla C870 that provides only single-precision capability, and the Tesla C1060, which adds double-precision support. We show that our single- and double- precision implementations have a huge gap in performance and accuracy. However, we are able to combine the advantages of single-precision performance with double-precision accuracy, by exploiting a mixed-precision approach, in which we use single-precision for the function evaluations, and

double-precision for the accumulation of the pair-wise functions. Our optimized mixed-precision implementation delivers a speedup of 225*x* for the *HFDB-He* potential energy and wave function calculations. For the *HFDB-He* potential energy function, we observe a 5*x* difference in performance between the corresponding FPGA and GPU implementations. The conditional branching operations involved in evaluating the *SAPT-He* potential would likely impact its performance on the GPUs to some extent and we might need to look at ways of optimizing the kernel. However, we show that our FPGA framework is general-purpose and allows us to study a number of properties, using the interpolation framework, irrespective of the exact form of the analytical function.

We develop analytical performance models for the CPU, FPGA, and GPU platforms to help understand and optimize the code. With the existing analytical models for HPC and HPRC, we include clock cycle accurate execution times of our design on the FPGA. This is useful to understand the application performance on our target platform and predict the performance on other HPRC platforms. For example, on the Cray XD1 Virtex-4 FPGA, we predict a 100*x* speedup if the kinetic energy calculations are also mapped on the FPGA, compared to a present speedup of 40*x* (while using the *HFDB-He* potential). On the GPUs, the analytical models will help us better optimize the use of memory accesses, and offer us insights into the hardware architectural details.

8.2. Future Work

We have demonstrated the general-purpose nature of our open-source FPGA hardware accelerated framework, using which we calculate the energies and trial wave functions of a cluster of inert gas atoms. The present generation of FPGAs with larger gate densities will allow us to investigate both the ability to evaluate additional properties and also experiment with numerical precision. The mixed-precision approach on GPUs used in this dissertation can be investigated on the FPGAs for higher performance.

This will allow us to optimize the parameters for our functions using a lower precision, and then use the parameters to obtain exact energies using a higher precision.

Future directions in our research include optimizing and providing our GPU implementation as an open-source framework. GPUs have an attractive price-performance ratio and this framework will provide cost-effective capabilities to the scientific community. Another interesting effort is to scale our application on to a large number of computing nodes, so we can most effectively partition the task of sampling thousands of configurations for tens of thousands of iterations, which is typical in Monte Carlo simulations. Other useful extensions of this work will be in the area of performance modeling, to account for the memory bandwidth limitations for systems with multiple RC elements, and extend them to study application performance on heterogeneous architectures. Developing performance metrics which include power consumption and chip area will be important to assess the suitability of a platform for an application.

Based on our present results, we can identify the best ways of partitioning the application onto hybrid computing platforms, which could consist of multi-core processors, multiple FPGAs and GPUs. Next-generation supercomputers are likely to be made of one or more of these architectures. Given this roadmap of supercomputing, our framework combined with analytical modeling and experimental results will help us investigate related scientific applications onto individual and hybrid computing platforms.

Bibliography

Bibliography

[1] Cornell Theory Center (CTC),< http://www.cac.cornell.edu/ >.

[2] International Assessment of Simulation-Based Engineering and Science (NSF Report),< http://www.engin.umich.edu/newscenter/pressReleases/20090422111935nqq/ >.

[3] F. Suits, M. C. Pitman, J. W. Pitera, W. C. Swope and R. S. Germain, "Overview of molecular dynamics techniques and early scientific results from the Blue Gene project," *IBM Journal of Research and Development*, vol. 49, pp. 475-487, 2005.

[4] K. Barker, K. Davis, A. Hoisie, D. Kerbyson, M. Lang, et al., "Entering the petaflop era: the architecture and performance of Roadrunner," in *SC '08: Proceedings of the 2008 ACM/IEEE Supercomputing Conference*, Austin, Texas, 2008, pp. 1-11.

[5] ORNL Jaguar,< http://www.cray.com/Products/XT/Product/ORNLJaguar.aspx >.

[6] Top500 Supercomputers,< http://www.top500.org/ >.

[7] NICS Kraken,< http://www.nics.tennessee.edu/computing-resources/kraken >.

[8] A. DeHon and S. Hauck, *Reconfigurable Computing, The Theory and Practice of FPGA-Based Computation*: Morgan Kauffman, 2007.

[9] NVIDIA Tesla GPUs,< http://www.nvidia.com/object/tesla_computing_solutions.html >.

[10] AMD/ATI GPUs,< http://ati.amd.com/products/index.html >.

[11] J. A. Kahle, M. N. Day, H. P. Hofstee, C. R. Johns, T. R. Maeurer, et al., "Introduction to the cell multiprocessor," *IBM Journal of Research and Development*, vol. 49, pp. 589-604, July/September 2005.

[12] V. Sachdeva, M. Kistler, E. Speight and T. H. K. Tzeng, "Exploring the viability of the Cell broadband engine for bioinformatics applications," in *IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, 2007, pp. 1-8.

[13] Y. Liu, W. Huang, J. Johnson and S. Vaidya, "GPU Accelerated Smith Waterman," in *International Conference on Computational Science*, 2006, pp. 188-195.

[14] J. E. Stone, J. C. Phillips, P. L. Freddolino, D. J. Hardy, L. G. Trabuco, et al., "Accelerating molecular modeling applications with graphics processors," *Journal of Computational Chemistry*, vol. 28, pp. 2618-2640, 2007.

[15] J. L. Hennessy and D. A. Patterson, *Computer Architecture: A Quantitative Approach*: Morgan Kaufman, 1996.

[16] K. Compton and S. Hauck, "Reconfigurable computing: a survey of systems and software," *ACM Computing Surveys*, vol. 34, pp. 171-210, 2002.

[17] Virtex-4 FPGA,< http://www.xilinx.com/support/documentation/data_sheets/ds112.pdf >.

[18] K. Underwood, "FPGAs vs. CPUs: trends in peak floating-point performance," in *Proceedings of the 2004 ACM/SIGDA 12th international symposium on Field programmable gate arrays* Monterey, California, USA: ACM, 2004, pp. 171-180.

[19] F. Rodriguez-Henriquez, N. A. Saqib, A. Diaz-Perez and C. Koc, *Cryptography Algorithms on Reconfigurable Hardware*: Springer, 2007.

[20] S. Choi, R. Scrofano, V. K. Prasanna and J. Jang, "Energy-efficient signal processing using FPGAs," in *ACM/SIGDA Eleventh International Symposium on Field-Programmable Gate Arrays*, 2003, pp. 225-234.

[21] M. Gokhale and P. Graham, *Reconfigurable Computing: Accelerating Computation with Field-Programmable Gate Arrays*: Springer, 2005.

[22] J. D. Owens, D. Luebke, N. Govindaraju, M. Harris, J. Kr\"ger, et al., "A Survey of General-Purpose Computation on Graphics Hardware," in *Eurographics 2005, State of the Art Reports*, 2005, pp. 21-51.

[23] Y. Gu, "FPGA acceleration of molecular dynamics simulations," *Ph.D. Dissertation*, Boston University, 2008.

[24] J. S. Meredith, S. R. Alam and J. S. Vetter, "Analysis of a Computational Biology Simulation Technique on Emerging Processing Architectures," in *IEEE International Parallel and Distributed Processing Symposium, IPDPS*, 2007, pp. 1-8.

[25] Z. K. Baker, M. Gokhale and J. L. Tripp, "Matched filter computations on FPGA, Cell and GPU," in *International Symposium on Field-Programmable Custom Computing Machines*, 2007.

[26] M. P. Nightingale and C. J. Umrigar, *Quantum Monte Carlo methods in physics and chemistry*: Kluwer Academic Publishers, 1999.

[27] G. L. Warren, "Overcoming statistical error and bias in Quantum Monte Carlo: application to metal-doped helium clusters," *Ph.D. Dissertation*, University of Tennessee, 2005.

[28] R. D. Chamberlain, J. M. Lancaster and R. K. Cytron, "Visions for application development on hybrid-computing systems," *Parallel Computing*, vol. 34, pp. 201-216, March 2008.

[29] M. P. Allen and D. J. Tildesley, *Computer Simulation of Liquids*: Clarendon Press, 1987.

[30] A. R. Leach, *Molecular Modeling: Principles and Applications*: Prentice Hall 2001.

[31] D. C. Rappaport, *The Art of Molecular Dynamics Simulation*: Cambridge University Press, 2004.

[32] R. D. Skeel, I. Tezcan and D. J. Hardy, "Multiple grid methods for molecular dynamics," *Journal of Computational Chemistry*, vol. 23, pp. 673-684, 2002.

[33] T. A. Darden, D. M. York and L. G. Pedersen, "Particle mesh Ewald: An N log(N) method for Ewald sums in large systems," *Journal of Chemical Physics*, vol. 98, pp. 10089-10092, 1993.

[34] C. Sagui and T. A. Darden, "Molecular dynamics simulations of biomolecules: long-range electrostatic effects," *Annu Rev Biophys Biomol Struct*, vol. 28, pp. 155-179, 1999.

[35] N. Metropolis and S. Ulam, "The Monte Carlo Method," *Journal of the American Statistical Association*, vol. 44, pp. 335-341, 1949.

[36] J. M. Thijssen, *Computational Physics*: Cambridge University Press, 1999.

[37] J. Doll and D. L. Freeman, "Monte Carlo methods in chemistry," *IEEE Computational Science and Engineering*, vol. 1, pp. 22-32, 1994.

[38] C. L. Tang, *Fundamentals of Quantum Mechanics for Solid State Electronics and Optics*: Cambridge University Press, 2005.

[39] R. A. Aziz, F. R. W. McCourt and C. C. K. Wong, "A new determination of the ground state interatomic potential for He_2," *Molecular Physics*, vol. 61, pp. 1487 - 1511, 1987.

[40] A. R. Janzen and R. A. Aziz, "An accurate potential energy curve for helium based on ab initio calculations," *The Journal of Chemical Physics*, vol. 107, pp. 914-919, 1997.

[41] R. N. Barnett and K. B. Whaley, "Variational and diffusion Monte Carlo techniques for quantum clusters," *Physical Review A*, vol. 47, p. 4082, 1993.

[42] M. V. Rama Krishna and K. B. Whaley, "Wave functions of helium clusters," *The Journal of Chemical Physics*, vol. 93, pp. 6738-6751, 1990.

[43] Cray Inc.,< http://www.cray.com/products/xd1/index.html >.

[44] Cray Inc. XT5,< http://www.cray.com/Products/XT5.aspx >.

[45] SRC Computers Inc.,< http://www.srccomp.com/HardwareSpecs.htm >.

[46] SGI Inc.,< http://www.sgi.com/products/rasc/ >.

[47] DRC Computers Inc.,< http://www.drccomputer.com/ >.

[48] XtremeData Inc.,< http://www.xtremedatainc.com/ >.

[49] Nallatech Inc.,< http://www.nallatech.com/ >.

[50] Nallatech FSB FPGA Accelerator,
< http://www.nallatech.com/mediaLibrary/images/english/7633.pdf >.

[51] FPGA High Performance Computing Alliance (FHPCA),< http://www.fhpca.org/ >.

[52] R. Baxter, S. Booth, M. Bull, G. Cawood, J. Perry, et al., "Maxwell - a 64 FPGA Supercomputer," in *Second NASA/ESA Conference on Adaptive Hardware and Systems*, 2007, pp. 287-294.

[53] Starbridge Systems Viva,< http://www.starbridgesystems.com/viva-software/what-is-viva/ >.

[54] Handel-C,< http://www.agilityds.com/products/c_based_products/dk_design_suite/handel-c.aspx >.

[55] Mitrionics Inc.,< http://www.mitrionics.com/ >.

[56] L. Seiler, D. Carmean, E. Sprangle, T. Forsyth, M. Abrash, et al., "Larrabee: a many-core x86 architecture for visual computing," *ACM Transactions on Graphics*, vol. 27, pp. 1-15, 2008.

[57] J. Kessenich, D. Baldwin and R. Rost,< http://www.opengl.org/documentation/glsl >.

[58] Microsoft DirectX 10,
< http://www.gamesforwindows/en-US/AboutGFW/Pages/DirectX10.aspx >.

[59] J. Hensley, "Hardware and Compute Abstraction Layers for Accelerated Computing using Graphics Hardware and Conventional CPUs," in *High Performance Embedded Computing (HPEC)* 2007.

[60] NVIDIA Inc. CUDA,< http://www.nvidia.com >.

[61] I. Buck, T. Foley, D. Horn, J. Sugerman, K. Fatahalian, et al., "Brook for GPUs: a stream computing on graphics hardware," *ACM Transactions on Graphics*, vol. 23, pp. 777-786, 2004.

[62] M. D. McCool, *Metaprogramming GPUs with Sh*: AK Peters, 2004.

[63] RapidMind Inc.,< http://www.rapidmind.net/ >.

[64] D. Tarditi, S. Puri and J. Oglesby, "Accelerator: using data parallelism to program GPUs for general-purpose uses," in *Proceedings of the 12th International Conference on Architectural Support for Programming Languages and Operating Systems*, 2006.

[65] OpenCL,< http://www.khronos.org/registry/cl/ >.

[66] K. Sano, T. Iizuka and S. Yamamoto, "Systolic Architecture for Computational Fluid Dynamics on FPGAs," in *IEEE Field-Programmable Custom Computing Machines (FCCM)*, 2007, pp. 107-116.

[67] W. D. Smith and A. R. Schnore, "Towards an RCC-based accelerator for computational fluid dynamics applications," *Journal of Supercomputing*, vol. 30, pp. 239-261, 2004.

[68] Y. Gu, T. VanCourt and M. C. Herbordt, "Accelerating molecular dynamics simulations with configurable circuits," *IEEE Proceedings on Computers and Digital Techniques,* vol. 153, pp. 189-195, May 2006.

[69] J. Sun, G. D. Peterson and O. O. Storaasli, "High Performance Mixed-Precision Linear Solver for FPGAs," *IEEE Transactions on Computers,* vol. 57, pp. 1614-1623, 2008.

[70] N. Galoppo, N. K. Govindaraju, M. Henson and D. Manocha, "LU-GPU: Efficient Algorithms for Solving Dense Linear Systems on Graphics Hardware," in *Proceedings of the ACM/IEEE Supercomputing 2005 Conference*, 2005.

[71] F. Allen, G. Almasi, W. Andreoni and D. Beece, "Blue Gene: A vision for protein science using a petaflop supercomputer," *IBM Systems Journal,* vol. 40, pp. 310-327, 2001.

[72] R. S. Germain, Y. Zhestkov and M. Eleftheriou, "Early performance data on the Blue Matter molecular simulation framework," *IBM Journal of Research and Development,* vol. 49, pp. 447-455, 2005.

[73] J. Makino, "The GRAPE project," in *Computing in Science and Engineering*, 2006, pp. 30-40.

[74] M. Taiji, T. Narumi, Y. Ohno, N. Futatsugi, A. Suenaga, et al., "Protein Explorer: A Petaflops Special-Purpose Computer System for Molecular Dynamics Simulations," in *Supercomputing, 2003 ACM/IEEE Conference*, 2003, pp. 15-15.

[75] M. Taiji, J. Makino, A. Shimazu, R. Takada, T. Ebisuzaki, et al., "MD-GRAPE: a parallel special-purpose computer system for classical molecular dynamics simulations," in *Proceedings of International Conference on Physics Computing*, 1994, pp. 609-612.

[76] D. E. Shaw, M. M. Deneroff, R. O. Dror, J. S. Kuskin, R.H. Larson, et al., "Anton, a special-purpose machine for molecular dynamics simulation," in *Proceedings of the 34th Annual International Symposium on Computer Architecture* San Diego, California, USA: ACM, 2007.

[77] "AMBER MD package," http://amber.scripps.edu/.

[78] NAMD,< http://www.ks.uiuc.edu/Research/namd/ >.

[79] CHARMM,< http://www.charmm.org/ >.

[80] LAMMPS Molecular Dynamics Simulator,< http://lammps.sandia.gov/ >.

[81] GROMACS,< http://www.gromacs.org/ >.

[82] J. C. Phillips, R. Braun, W. Wang, J. Gumbart, E. Tajkhorshid, et al., "Scalable molecular dynamics with NAMD," *Journal of Computational Chemistry,* vol. 26, pp. 1781-1802, 2005.

[83] N. Azizi, I. Kuon, A. Egier, A. Darabiha and P. Chow, "Reconfigurable molecular dynamics simulator," in *IEEE International Symposium on Field-Programmable Custom Computing Machines (FCCM)*, 2004.

[84] R. Scrofano and V. K. Prasanna, "Computing Lennard-Jones potentials and forces with reconfigurable hardware," in *International Conference on Engineering of Reconfigurable Systems and Algorithms (ERSA)*, 2004.

[85] V. Kindratenko and D. Pointer, "A case study in porting a production scientific supercomputing application to a reconfigurable supercomputer," in *IEEE Symposium on Field-Programmable Custom Computing Machines (FCCM)*, 2006.

[86] S. R. Alam, J. S. Vetter and M. C. Smith, "An Application Specific Memory Characterization Technique for Co-processor Accelerators," in *IEEE International Conference on Application-specific Systems, Architectures and Processors (ASAP)*, 2007, pp. 353-358.

[87] R. Scrofano and V. K. Prasanna, "A hardware/software approach to molecular dynamics on reconfigurable computers," in *IEEE International Symposium on Field-Programmable Custom Computing Machines (FCCM)*, 2006.

[88] Y. Gu and M. C. Herbordt, "High Performance Molecular Dynamics Simulations with FPGA Coprocessors," in *Reconfigurable Summer Systems Institute*, 2007.

[89] C. I. Rodrigues, D. J. Hardy, J. E. Stone, K. Schulten and W. W. Hwu, "GPU acceleration of cutof pair potentials for molecular modeling applications," in *ACM International Conference on Computing Frontiers*, 2008, pp. 273-282.

[90] C. P. Cowen and S. Monaghan, "A reconfigurable Monte Carlo clustering processor," in *IEEE Symposium on Field-Programmable Custom Computing Machines (FCCM)*, 1994, pp. 59-65.

[91] G. L. Zhang, P. H. W. Leong, C. H. Ho, K. H. Tsoi and et al, "Reconfigurable acceleration for Monte Carlo based financial simulation," in *IEEE International Conference on Field-Programmable Technology (FPT)*, 2005, pp. 215-222.

[92] Scalable Parallel Random Number Generator Library,< http://sprng.fsu.edu >.

[93] J. Lee, G. D. Peterson, R. J. Hinde and R. J. Harrison, "Hardware accelerated scalable parallel random number generator for Monte Carlo methods," in *IEEE Midwest Symposium on Circuits and Systems (MWSCAS)*, 2008.

[94] M. Matsumoto and T. Nishimura, "A 623-dimensionally equidistributed uniform pseudorandom number generator," *ACM Transactions on Modeling and Computer Simulation*, vol. 8, pp. 3-30, 1998.

[95] CASINO,< http://www.tcm.phy.cam.ac.uk/~mdt26/casino2.html >.

[96] CHAMP,< http://pages.physics.cornell.edu/~cyrus/champ.html >.

[97] ZORI,< http://www.zori-code.com/ >.

[98] QMCBeaver,< http://www.qmcbeaver.sourceforge.net/ >.

[99] QMC@home,< http://qah.uni-muenster.de/ >.

[100] A. G. Anderson, W. A. Goddard III and P. Schröder, "Quantum Monte Carlo on graphical processing units," *Computer Physics Communications*, vol. 177, pp. 298-306, 2007.

[101] W. Kahan, "Pracniques: Further remarks on reducing truncation errors," *Communications of the ACM*, vol. 8, pp. 40-40, 1965.

[102] L. Hu and I. Gorton, "Performance Evaluation for Parallel Systems: A Survey," University of New South Wales, Australia 1997.

[103] K. Kant, *Introduction to Computer System Performance Evaluation*: McGraw-Hill Inc., 1992.

[104] G. D. Peterson, "Parallel application performance on shared heterogeneous workstations," *Ph.D. Dissertation*, Washington University, 1994.

[105] M. C. Smith and G. D. Peterson, "Parallel Application Performance on Shared High Performance Reconfigurable Computing Resources," *Performance Evaluation*, vol. 60, pp. 107-125, 2005.

[106] B. Holland, K. Nagarajan, C. Conger, A. Jacobs and A. George, "RAT: A Methodology for Predicting Performance in Application Design Migration to FPGAs," in *Proceedings of High Performance Reconfigurable Computing Technologies and Applications Workshop (HPRCTA)*, 2007.

[107] "Virtex-II Platform FPGAs: complete data sheet, http://direct.xilinx.com/bvdocs/publications/ds031.pdf," March 2005.

[108] Altera Inc.,< http://www.altera.com/ >.

[109] W. Rick Steven, D. L. Lynch and J. D. Doll, "A variational Monte Carlo study of argon, neon, and helium clusters," *The Journal of Chemical Physics*, vol. 95, pp. 3506-3520, 1991.

[110] D. Bressanini, M. Zavaglia, M. Mella and G. Morosi, "Quantum Monte Carlo investigation of small ^{4}He clusters with a He^{3} impurity," *Journal of Chemical Physics*, vol. 112, 2000.

[111] Xilinx Design Tools,< http://www.xilinx.com/tools/designtools.htm >.

[112] J. Nickolls, I. Buck, M. Garland and K. Skadron, "Scalable Parallel Programming with CUDA," *ACM Queue*, vol. 6, pp. 40-53, 2008.

[113] J. A. Stratton, S. S. Stone and W. W. Hu, "MCUDA: An efficient implementation of CUDA kernels for multi-core CPUs," in *Proceedings of the 21st International Workshop on Languages and Compilers for Parallel Computing*, 2008.

[114] E. Lindholm, J. Nickolls, S. Oberman and J. Montrym, "NVIDIA Tesla: A Unified Graphics and Computing Architecture," *Micro, IEEE*, vol. 28, pp. 39-55, 2008.

[115] L. Nyland, M. Harris and J. Prins, "Fast N-Body Simulation with CUDA," in *GPU Gems 3*: Addison-Wesley Professional, 2007.

[116] Intel Nehalem,< http://www.intel.com/design/corei7/documentation.htm >.

[117] Intel Math Kernel Library (MKL),< http://software.intel.com/en-us/articles/intel-math-kernel-library-documentation/ >.

[118] A. Gothandaraman, G. D. Peterson, G. L. Warren, R. J. Hinde and R. J. Harrison, "An open-source hardware-accelerated Quantum Monte Carlo (HAQMC) framework for N-body systems," *Computational Physics Communications (accepted)*, 2009.

[119] G. M. Amdahl, "Validity of the Single Processor Approach to Achieving Large Scale Computing Capabilities," in *AFIPS Conference Proceedings*, 1967, pp. 483-485.

[120] J. L. Gustafson, "Reevaluating Amdahl's Law," *Communications of the ACM*, vol. 31, pp. 532-533, 1988.

[121] D. E. Culler, J. P. Singh and A. Gupta, *Parallel Computer Architecture: A Hardware/Software Approach*: Morgan Kaufmann, 1998.

[122] G. D. Peterson and R. D. Chamberlain, "Parallel application performance in a shared resource environment," *IEE Distributed Systems Engineering*, vol. 3, pp. 9-19, 1996.

[123] K. D. Underwood, K. S. Hemmert and C. Ulmer, "Architectures and APIs: assessing requirements for delivering FPGA performance to applications," in *Supercomputing*, 2006.

[124] Amirix Inc.,< http://www.amirix.com/ >.

Vita

Akila Gothandaraman joined the Department of Electrical Engineering and Computer Science at the University of Tennessee, Knoxville as a graduate student in August 2001. She received her B.E. degree in Electrical and Electronics Engineering from PSG College of Technology, Coimbatore, India and her M.S. degree in Electrical Engineering from University of Tennessee, Knoxville. She has been part of the Tennessee Advanced Computing Laboratory since August 2004. Her current research focuses on deploying emerging architectures for accelerating computer simulations in chemistry. She was a recipient of the Analog Devices Fellowship in 2003 and received the ACM award for Best Student Poster in the Student Research Competition at Supercomputing Conference, 2008. She is a student member of IEEE, ACM, SWE, and Eta Kappa Nu. Post-graduation, she will be joining the Center for Simulation and Modeling, University of Pittsburgh as Research Assistant Professor/HPC Consultant.