Research Article

A Pipelined and Parallel Architecture for Quantum Monte Carlo Simulations on FPGAs

Akila Gothandaraman,^{1,2} Gregory D. Peterson,¹ G. Lee Warren,³ Robert J. Hinde,⁴ and Robert J. Harrison⁴

¹Department of Electrical Engineering and Computer Science, The University of Tennessee, Knoxville, TN 37996, USA

²Center for Simulation and Modeling, Department of Chemistry, The University of Pittsburgh, Pittsburgh, PA 15260, USA

³Department of Chemistry, Yeshiva University, New York, NY 10033, USA

⁴Department of Chemistry, The University of Tennessee, Knoxville, TN 37996, USA

Correspondence should be addressed to Akila Gothandaraman, agothan1@utk.edu

Received 5 June 2009; Accepted 19 November 2009

Academic Editor: Ethan Farquhar

Copyright © 2010 Akila Gothandaraman et al. This is an open access article distributed under the Creative Commons Attribution License, which permits unrestricted use, distribution, and reproduction in any medium, provided the original work is properly cited.

Recent advances in Field-Programmable Gate Array (FPGA) technology make reconfigurable computing using FPGAs an attractive platform for accelerating scientific applications. We develop a deeply pipelined and parallel architecture for Quantum Monte Carlo simulations using FPGAs. Quantum Monte Carlo simulations enable us to obtain the structural and energetic properties of atomic clusters. We experiment with different pipeline structures for each component of the design and develop a deeply pipelined architecture that provides the best performance in terms of achievable clock rate, while at the same time has a modest use of the FPGA resources. We discuss the details of the pipelined and generic architecture that is used to obtain the potential energy and wave function of a cluster of atoms.

1. Introduction

Reconfigurable Computing (RC) using Field-Programmable Gate Arrays (FPGAs) is emerging as a computing paradigm to accelerate the computationally intensive functions of an application [1]. RC provides users the flexibility of a software solution with hardware-like performance. We develop a parallel and pipelined reconfigurable architecture for accelerating the computationally intensive functions of a Quantum Monte Carlo (QMC) application. QMC is a simulation technique that provides a practical and efficient way of solving the many-body Schrödinger equation [2]. Using this method, we can obtain the ground-state properties, such as energy and wave function, of a system of interacting particles. A software QMC application can be used to simulate a system of particles on general-purpose processors. To speed up the application and to provide better computational scaling with the system size, we use a hardware-software approach to speed up the computationally intensive functions of the application. In our implementation, we use hardware-based

techniques to exploit parallelism using pipelining. Due to the complex nature of the functions involved, we use an interpolation-based approximation method for calculating these functions. The exact shape of the function depends on the nature of the atoms involved. Hence, the use of FPGAs gives us the flexibility to develop a programmable architecture so we can vary the parameters depending on the system. We propose a reconfigurable hardware architecture using Field-Programmable Gate Arrays (FPGAs) to implement the kernels of our application. In [3, 4], we presented our ongoing research and our hardware accelerated framework where we achieve promising results when the kernels of the application are ported to the Cray XD1 reconfigurable supercomputing platform [5]. Here, we focus on the architectural details, our design goals, and choices while designing the pipelined architecture to calculate the potential energy and wave function of atomic clusters.

Our design goals are performance, numerical accuracy, and flexibility. To quantify performance, we measure the speed up achieved by our hardware implementation over the

optimized software application. *Accuracy* is guaranteed by employing the numerical precision that delivers results that are comparable to the software version employing double-precision floating-point representation. We satisfy the above design goals by developing a deeply pipelined reconfigurable architecture that employs a fixed-point representation chosen after careful error analysis. *Flexibility* is provided by creating a general user-friendly framework [6] that can be used to simulate a variety of atomic clusters, which have the same overall function, with slightly different simulation parameters. The current framework can be extended to calculate other properties during the simulation. Our present implementation meets the above design goals by providing a speed up of 40x for the potential energy and wave function kernels targeted to the Xilinx Virtex-4 FPGA [7] on the Cray XD1 platform and accuracy comparable to the double-precision software implementation on the processor. A general interpolation framework allows us to calculate both the pairwise potential energies and wave function of atomic clusters with the flexibility to both vary the identity of the atoms as well as to realize different forms of potential energy or wave function.

The paper is organized as follows. In Section 2, we provide an overview of the QMC algorithm. Section 3 describes the different components of the pipelined reconfigurable hardware. In Section 4, we present the results of our implementation on the Cray XD1 platform. We provide conclusions and directions for future research in Section 5.

2. Algorithm

Quantum Monte Carlo (QMC) methods are widely used in physics and physical chemistry to obtain the ground-state properties of atomic or molecular clusters [2]. These methods are used to solve the many-body Schrödinger equation. Equation (1) represents the Schrödinger equation, the fundamental equation of quantum mechanics. In this equation, H refers to the Hamiltonian operator; E represents the energy of the system. In (2), we rewrite the Hamiltonian as the sum of kinetic and potential energy terms. This equation then 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 (3). Solving this equation is trivial for small systems, but as the dimensions of the system increase, it is impractical to solve the equation analytically. In (2) and (3), $\hbar$ is the Planck's constant, ψ is the wave function, and ∇^2 is the Laplace operator,

$$H\psi = E\psi, \quad (1)$$

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

$$\left[-\frac{\hbar^2}{2m} \nabla^2 + V(r) \right] \psi(r) = E\psi(r). \quad (3)$$

We use a flavor of QMC called the Variational Monte Carlo (VMC) algorithm [2, 3]. In Step 1 of the algorithm,

FIGURE 1: Data movement in the QMC algorithm.

we initialize a reference configuration of atoms. In Step 2, we add a small random displacement to all the atoms in the configuration obtained in Step 1 to obtain a new configuration. Step 3 of the algorithm consists of the calculation of properties namely potential energy and wave function. These are pairwise functions and hence $O(N^2)$ in the number of atoms, N . In Step 4, we accept or reject the configuration using the ratio of the wave function values of the new and old configurations. Depending on the ratio, we use a uniformly distributed random number to decide whether or not to accept a configuration and its associated properties. These steps are repeated for all configurations and thousands of iterations, to accurately estimate the properties.

Figure 1 shows the data movement in the QMC algorithm, which will help us decide how to partition the steps of the algorithm between the software application and reconfigurable hardware. The potential energy and wave function are both functions of the coordinate distances between the atoms. The coordinate distance calculation, calculation of potential energy and wave function are all $O(N^2)$ in the number of atoms. The total potential energy, V_{total} , is the sum of the $N(N-1)/2$ pairwise contributions as shown in (4). The total wave function, ψ_{total} , is approximated as the product of the pairwise contributions as shown in (5). In (4) and (5), r_{ij} denotes the radial distance between atoms, i and j . Upon the completion of Step 3 of the VMC algorithm, we have a single energy or wave function result that is transferred back to the host processor,

$$V_{\text{total}} \approx \sum_{i < j} V_2(r_{ij}), \quad (4)$$

$$\psi_{\text{total}} \approx \prod_{i < j} \psi_2(r_{ij}). \quad (5)$$

Table 1 shows the execution time for the different components, total time and compute time (for 1 iteration and $N = 4096$), and the percentage of total time for computing the kernels (Step 3 of the QMC algorithm). As we can see, the compute time (time for potential energy, and wave function calculations) dominates the total time for an iteration of the algorithm. Hence, we offload Step 3 of the algorithm to the FPGA. We have $O(N)$ coordinate positions to be moved from the processor to the FPGA for every iteration. A high-speed interconnect between the FPGA and processor helps keep the cost of the data movement low for each iteration, so we can speed up the computation of properties in Step 3.

TABLE 1: Total execution time versus compute time for 1 iteration.

Components	Time (in s) or %
Potential energy calculation	0.7615
Wave function calculation	2.739
Compute time for 1 iteration	3.500
Total time for 1 iteration	3.715
% of total time spent on compute	94%

FIGURE 2: Top-level Block Diagram.

3. Architecture

Figure 2 shows the top-level block diagram of the pipelined architecture implemented on the FPGA. The architecture consists of the following components: *CalcDist* calculates the squared distances between the pairs of atoms. *CalcFunc* module uses as inputs the squared distances from *CalcDist* and evaluates the functions using an interpolation scheme. This is a generic module used in our architecture to compute the potential energy and wave function. These blocks are denoted as *CalcPE* and *CalcWF* in Figure 2. *AccFunc*, a generic module, accumulates the energies and wave functions to result in partial results that are delivered to the host processor. The accumulators for potential energy and wave function are labeled in Figure 2 as *AccPE* and *AccWF*. Different accumulator configurations are used depending on whether we compute the sum or product of the computed functions.

In addition to the above components, lookup tables using on-chip Block RAMs (BRAMs) are used to store the coordinate positions of atoms and the interpolation coefficients for the function evaluations. A state machine

FIGURE 3: Memory Layout.

TABLE 2: BlockRAM Usage by the Lookup Tables.

	<i>BlockPos</i>	<i>BlockPE</i>	<i>BlockWF</i>
$x/(a1 \text{ and } a2)$	16 KB (8)	18 KB (10)	18 KB (10)
$y/(b1 \text{ and } b2)$	16 KB (8)	18 KB (10)	18 KB (10)
$z/(c1 \text{ and } c2)$	16 KB (8)	18 KB (10)	18 KB (10)
Total	48 KB (24)	54 KB (30)	54 KB (30)

controller is used to generate the addresses to the BRAMs that store the coordinate positions and transfer the (x, y, z) positions to the *CalcDist* module. The *CalcBin* module is used to compute the bin address, which is used to fetch the (a, b, c) interpolation coefficients. This module also produces an output, *delta*, that is used along with the interpolation coefficients by the *CalcFunc* modules. The distances from the *CalcDist* module are used by both *CalcPE* and *CalcWF* modules, which operate in parallel. Also, the components are deeply pipelined and designed to produce a result every clock cycle.

3.1. Lookup Tables. The 18-kbit embedded BRAMs on the Xilinx FPGA are used as lookup tables to store the (x, y, z) coordinate positions and (a, b, c) interpolation coefficients for the function evaluation. These memories are instantiated using the Xilinx Core Generator tools [8]. Figure 3 shows the layout of the BRAMs used to realize the above lookup tables. The BRAM used to store positions, called the *BlockPos* can support clusters of up to 4096 atoms. (Note that this is a limitation due to the size of the target FPGA. On the currently targeted FPGA, we use any remaining BRAMs to calculate additional properties of interest. On newer FPGAs, we can configure the BRAMs to support larger clusters due to larger memories). We use a single dual-port BRAM to store the 4096 32-bit positions. Three BRAMs are used to store the (x, y, z) positions for a total of 48 KB as shown in Table 2. This requires 24 BRAMs on the FPGA.

From our initial experiments varying the order of interpolation, we infer that quadratic interpolation with coefficients (a, b, c) guarantees the required accuracy with modest use of BRAM resources. The potential energy and wave function are divided into two regions, region I and

region II with each region approximated using a unique set of interpolation coefficients [3]. The lookup tables for each function store 256 (a_1, b_1, c_1) 52-bit coefficients for region I and 1344 (a_2, b_2, c_2) 52-bit coefficients for region II. The BRAMs to store the region I and II coefficients are configured with a depth of 256 and 2048 and width of 64-bit, respectively. These lookup tables are referred to as *BlockPE* and *BlockWF*.

For region I coefficients, we require 6 KB of memory, and for region II coefficients, we require 48 KB of memory for a total of 54 KB. For potential energy and wave function calculations, the coefficient memories consume a total of 60 BRAMs on the FPGA. The amount of memory needed to store the interpolation coefficients is given in Table 2. The number of Block RAMs needed for the above lookup tables is given in parenthesis in Table 2.

We use dual-port BRAMs, such that Port A writes the coordinate positions and Port B is used to read the positions. Ports A and B have a read pipeline latency of one clock cycle. In an RC setup, the host processor would load the positions to the *BlockPos* memory and the interpolation coefficients to the *BlockPE* and *BlockWF* BRAMs, respectively. The design implemented on the FPGA would read the *BlockPos* memory to calculate the distances. These distances would be used along with the interpolation coefficients from *BlockPE* or *BlockWF* to obtain the potential energy or wave function. Since the analytical functions for potential energy and wave function remain fixed throughout the simulation, the interpolation coefficients need to be loaded to the *BlockPE* or *BlockWF* memories by the processor only once prior to the beginning of the pipeline operation. However, the change in coordinate positions of the atoms, which relates to the physical movement of the atoms during the simulation, requires us to load the new positions to the *BlockPos* memory. As described earlier, the atoms are perturbed during every iteration of the algorithm and hence new positions are copied by the processor to the BRAMs for every iteration. The state machine transitions from one state to another to generate the addresses to the *BlockPos* memory so that the positions are read from the memory and transferred to the distance calculation and function evaluation modules. We calculate the energies and wave functions due to the atomic interactions in the shaded portion of the matrix (upper triangular part of the matrix) shown in Figure 4.

3.2. Distance Calculation (*CalcDist*). The ground-state properties such as potential energy and wave function are functions of the coordinate distance, r_{ij} , between a pair of atoms i and j . The distance calculation module, *CalcDist*, computes the squared distance between pairs of atoms. We eliminate the expensive square root on the FPGA by rewriting these properties as functions of the squared distance. Figure 5 shows the block diagram of this component. The latencies of the submodules are shown in clock cycles. The state machine controller provides the read addresses to the position memory, *BlockPos*, every clock cycle. The data outputs from the Position BRAMs are connected to the inputs of the *CalcDist* module. Since $r_{ij} = r_{ji}$ and $r_{ij} = 0$ when

FIGURE 4: State machine generates addresses to read *BlockPos* BRAM in the order shown.

FIGURE 5: Squared distance calculation, *CalcDist*.

$i = j$, we only need to calculate $N(N - 1)/2$ squared distances (upper or lower triangular portion of the matrix in Figure 4).

We use the Xilinx IP cores from the CoreGen library [8] 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. For example, the multipliers are available with minimum or maximum pipelining with two- or seven-clock cycle latency respectively. We use the multipliers that are available as hard macros on the Xilinx Virtex4 series, in the form of DSP48s. To increase clock rates, the multipliers are configured for maximum pipelining, which corresponds to a latency of seven clock cycles. Table 3 shows the data widths and latencies of this module. The data widths for inputs and outputs are chosen after analyzing the errors with various fixed-point representations. The initial latency of this module is ten clock cycles after which the pipeline is full and it produces a 53-bit 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 coordinate distances. However, since the entire architecture is pipelined,

TABLE 3: Data Widths and latencies of *CalcDist*.

Signal/Core	Data Widths	Latency
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)	—

FIGURE 6: Region I Bin Calculation.

the calculation of the pairwise distances is overlapped with the function evaluation.

3.3. Bin Calculation (*CalcBin*). The schemes to look up the interpolation coefficients are different for the two regions in each function, as the functions exhibit different numerical behavior in each region. The squared distance from the *CalcDist* module is compared with a cutoff value, σ^2 , to classify the distance as a region I or region II distance. Region I of the function is approximated using 256 bins. Using the inverse of the bin width, we can choose from the 256 values corresponding to the squared distances. The bin lookup scheme for region I is shown in Figure 6. The lower 8-bits of the product of the squared distance and the inverse bin width (by extracting the value to the left of the decimal point) are used to fetch the interpolation coefficients from the memory.

We employ a logarithmic binning scheme in region II. We divide the region II into 21 regimes. Each regime is divided into 64 bins for a total of 1344 coefficients. For regions I and II, we have a total of $[256 + 21 * 64] * 3$ coefficients for quadratic interpolation. First, we determine the regime and then determine the bin within the regime. We store the inverse of the bin widths corresponding to each bin in the BRAMs, eliminating the need for a division operation. The block diagram of the first stage of the lookup scheme for region II is shown in Figure 7. The difference between the squared distances and the sigma value is used by the leading zero count detector to compute the index of the regime. The detector logic is implemented using a set of three priority encoders (Pr1, Pr2, Pr3). Now that we have the index of the regime (i.e., the right power of 2), we use this to fetch the bin widths associated with this regime. Figure 8 shows the second stage of the lookup scheme to obtain the bin address to fetch the interpolation coefficients. To compute the bin location for region II, we require additional constants (region

FIGURE 7: Region II Bin Calculation (First stage to obtain the regime).

FIGURE 8: Region II Bin Calculation (Second stage to obtain the bin address).

II shift constants and inverse bin width constants), which are obtained from BRAMs addressed using the regime. We can use the computed address to retrieve the interpolation coefficients for region II.

3.4. Calculation of Functions (*CalcFunc*). Figure 9 shows the data path of the *CalcFunc* module. This is a generic module that evaluates a function using polynomial interpolation. The interpolation coefficients, squared distances from the distance calculation module, and a delta value related to the squared distances are used to compute the function. In our reconfigurable architecture, we instantiate two copies of the *CalcFunc* module, one to calculate the potential energy, namely, the *CalcPE* and the other to compute the wave function, called the *CalcWF* as shown earlier in Figure 2. We load the corresponding (region I or region II) interpolation coefficients from the respective Block RAMs for each function. The address computed by the bin calculator, *CalcBin*, is used to fetch the (a, b, c) interpolation coefficients for the functions. The Xilinx IP cores (from Coregen) are used to implement the adders and multipliers. The multipliers can be configured to have a latency of two or seven clock cycles. We configure the multipliers for maximum pipelining (latency of seven clock cycles). This module outputs a 52-bit potential energy or wave function every clock cycle. Figure 9 shows the latencies (in clock cycles) in each step. Table 4 shows the data widths and latencies of the components of this module.

3.5. Accumulation of Functions (*AccFunc*). *AccFunc* is a generic module that is used to accumulate the results from the function evaluation module, *CalcFunc*. We instantiate two copies of the module, one for potential energy called

TABLE 4: 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	Pairwise Potential Energy/Wave Function: signed 0.52	—

FIGURE 9: *CalcFunc* Module.

AccPE and the other for wave function denoted as *AccWF*. Figure 10 shows the *AccPE* module that accumulates the intermediate values of potential energy produced by *CalcPE*. The potential energy function is transformed prior to interpolation [3]; hence we use two types of accumulators: we multiply the intermediate results of potential energies for region I and we add the intermediate results of the energies for region II. The potential values which result from region I squared distances and interpolation constants are called region I potential values. Similarly, region II potential values result from region II squared distances and interpolation constants. The region II accumulator, which is instantiated from the Xilinx CoreGen library, accumulates the region II potential, if produced during that clock cycle or a zero if a region I potential is produced that clock cycle. For accumulating region I potential values, we have to form products of the potential energy result every clock cycle. The region I accumulator accumulates the region I potential, if produced during that clock cycle or a one if a region II potential is produced that clock cycle. The Xilinx CoreGen multipliers (with maximum pipelining) have a latency of seven clock cycles. We use eight instances of the multipliers (as the multiplier outputs are registered) and switch between the multipliers to increase the data rate and produce a partial product every clock cycle. At the end of our calculations for N atoms, we have a single partial sum (PS) from region II and eight partial products (PP1-PP8) from region I. The region I accumulator has an initial latency of eight clock cycles, after which one partial result is produced every clock cycle. The region II accumulator has a latency of two clock cycles. The worst-case latency that this module adds to the pipeline is eight clock cycles.

Figure 11 shows the *AccWF* module that accumulates the intermediate results of the wave function produced

FIGURE 10: Potential energy accumulation (*AccPE*). FIGURE 11: Wave Function accumulation (*AccWF*).

by *CalcWF*. The wave function does not undergo any transformation prior to interpolation. Hence we use one accumulator, which multiplies the intermediate results of wave function contributions from regions I and II. As shown in (5), the total wave function is the product of the pairwise wave functions. Hence the accumulator, that adds the intermediate results, is omitted to conserve FPGA resources. *AccWF* produces eight partial products (PP1-PP8) from region I and II wave function results.

We perform successive multiplications of region I potential values and region I and II wave function values. The distances we sample are such that most values of potential energy and wave function are close to one. Hence, repeated multiplication of a number of these values (especially for large N) makes the product tend towards zero. This leads to a loss of precision in a fixed-precision register, with the appearance of leading zeros as the product approaches zero. To guarantee sufficient accuracy, we bit-shift the result to the left after computing the partial product (if it is less than 2^{-1}) and keep track of the number of shifts. The partial results along with the number of shifts are transferred back to the host processor.

There are also overflow issues associated with the accumulator while accumulating region II potential values. To overcome this problem, we accumulate the region II potentials in a large register that can hold N evaluations of the maximum value of the rescaled potential. The accumulated results are processed by the host processor, which reconstructs the floating-point value of the total potential energy and wave function.

FIGURE 12: Plot of relative error versus number of atoms.

4. Results

We target the FPGA implementation to the Cray XD1 high performance reconfigurable computing system [5]. This system incorporates Xilinx FPGAs to its compute nodes. We use a single-chassis Cray XD1 consisting of six compute nodes. Each compute node is equipped with a Xilinx Virtex-II Pro XC2VP50 or Virtex-4 LX160 FPGA [7, 9]. We use the compute node consisting of the Virtex-4 LX160 connected to a dual-processor dual-core 2.2 GHz AMD Opteron. The FPGA and processor are connected by the high-speed RapidArray interconnect, providing a bandwidth of 1.6 Gbytes/sec. As our baseline, we use an entirely software QMC application which uses double-precision that runs on the one core of the Opteron. We use the Scalable Parallel Random Number Generator (SPRNG) library [10] to generate the random numbers for perturbing the atom positions in Step 2 of the QMC algorithm as described earlier. Cray provides API functions that allow us to read from or write to the BRAMs and registers within our software application on the Opteron, simplifying the process of rewriting the software application to use the FPGA accelerators for the most computationally intensive functions. To port the hardware-accelerated QMC application, first we modify the software application to initialize the interpolation coefficients onto the BRAMs. We replace the potential energy and wave function calculations with calls to the FPGA. Next, we write the coordinate positions to the BRAMs after which a control signal is sent to the FPGA to begin the pipeline operation. Once the FPGA completes the operation and stores the results in the user registers, the FPGA sends a status signal back to the processor indicating that the results are ready. The results can be read within the hardware-accelerated application using the Cray API. Our hardware-accelerated framework is available from [6].

The design modules are developed using VHDL and synthesized using Xilinx XST tools. The implementation process, consisting of translating, mapping, and placing and

FIGURE 13: Speed up versus number of atoms.

TABLE 5: Resource usage on the Virtex-4 FPGA.

Resource type	Virtex-4 LX160 FPGA
SLICES	50%
BLOCK RAMs	51%
DSP48s	52%

routing of the design is done using the Xilinx ISE tools. The user design is clocked at 100 MHz. Table 5 shows the usage of resources—Slices, Block RAMs, and DSP48s (used as multipliers) when the design is targeted to a single Virtex-4 FPGA on one of the computing nodes of the Cray XD1 platform. The design consumes roughly 50 percent of the resources for wave function and potential energy. Careful redesign of the modules should allow us to fit an additional copy of both PE and WF pipelines on a single FPGA. We could also use the remaining resources to fit additional properties of interest, such as the kinetic energy. We compare the numerical results and the execution times of the software QMC application to the hardware-accelerated application targeted to a Virtex-4 LX160 FPGA. Figure 12 shows the relative error of the fixed-point FPGA potential energies compared to the double-precision results obtained on the CPU as we vary the number of atoms. The error performance is acceptable and within tolerance limits for the energies. Figure 13 shows the speed up obtained for the FPGA implementation over a single core of the Pacific XD1. This implementation provides a speed up of 40x over the software application running on a single core of the 2.2 GHz AMD Opteron.

5. Conclusions

We present a pipelined reconfigurable architecture to obtain the potential energy and wave function of clusters of atoms. We outline some of the goals that are critical for our design. From the design choices available for the building blocks of our architecture, we carefully evaluate the choices to obtain

optimal performance, numerical accuracy, and resource usage consumption. Our design choices including the use of a pipelined architecture, fixed-point representation, and a quadratic interpolation scheme to evaluate the function enable us to achieve a significant performance compared to the software version running on the processor. Our hardware design strategy for the Quantum Monte Carlo simulation offers a speed up of 40x on the Cray XD1 platform using a single FPGA over the software application while providing the desired numerical accuracy. Present extensions of our work include using multiple FPGAs on the Cray XD1 platform. This will allow us to distribute the configurations in the simulation among the FPGAs thereby exploiting additional parallelism. In this case, we can expect the total speed up to equal the number of FPGA equipped computing nodes times the speed up on a single FPGA node.

Acknowledgments

This work was supported by the National Science Foundation Grant NSF CHE-0625598, and the authors gratefully acknowledge prior support for related work from the University of Tennessee Science Alliance. The authors also thank Mr. Dave Strenski, Cray Inc., for providing them access to the Cray XD1 platform that was used to target our implementation.

References

- [1] S. Hauck and A. DeHon, *Reconfigurable Computing: The Theory and Practice of FPGA-Based Computation*, Morgan Kaufmann, San Francisco, Calif, USA, 2007.
- [2] J. M. Thijssen, *Computational Physics*, Cambridge University Press, Cambridge, UK, 1999.
- [3] A. Gothandaraman, G. D. Peterson, G. L. Warren, R. J. Hinde, and R. J. Harrison, "FPGA acceleration of a quantum Monte Carlo application," *Parallel Computing*, vol. 34, no. 4-5, pp. 278–291, 2008.
- [4] A. Gothandaraman, G. D. Peterson, G. L. Warren, R. J. Hinde, and R. J. Harrison, "Design decisions in the pipelined architecture for Quantum Monte Carlo Simulations," in *Proceedings of the IEEE MidWest Symposium on Circuits and Systems*, August 2008.
- [5] Cray Inc., <http://www.cray.com/Home.aspx>.
- [6] A. Gothandaraman, G. D. Peterson, G. L. Warren, R. J. Hinde, and R. J. Harrison, "A hardware-accelerated quantum Monte Carlo framework (HAQMC) for N-body systems," *Computer Physics Communications*, vol. 180, no. 12, pp. 2563–2573, 2009.
- [7] Xilinx Virtex-4 FPGA, <http://www.xilinx.com/support/documentation/virtex-4.htm>.
- [8] Xilinx Inc., <http://www.xilinx.com/ipcenter/index.htm>.
- [9] Xilinx Virtex-II Pro FPGA, <http://www.xilinx.com/support/documentation/virtex-ii.pro.htm>.
- [10] Scalable Parallel Random Number Generator (SPRNG) Library, <http://sprng.fsu.edu>.