Architectural Comparisons for a Quantum Monte Carlo Application

Akila Gothandaraman, Rick Weber, Gregory D. Peterson
Department of Electrical Engineering and Computer Science
University of Tennessee, Knoxville
Knoxville, USA
{akila, fweber1, gdp}@utk.edu

Robert J. Hinde, Robert J. Harrison
Department of Chemistry
University of Tennessee, Knoxville
Knoxville, USA
{rhinde, robert.harrison}@utk.edu

Abstract— **Recent technological advances have led to a number of emerging platforms such as multi-cores, reconfigurable computing, and graphics processing units. We present a comparative study of multi-cores, field-programmable gate arrays, and graphics processing units for a Quantum Monte Carlo chemistry application. The speedups of these implementations are measured relative to a multi-core implementation and the accuracy of each implementation compared against the double-precision CPU implementation. The Brook+ AMD implementation shows the best overall speedup and accuracy, with the mixed-precision NVIDIA CUDA implementation outperforming the Brook+ implementation for larger atomic clusters with a slightly lower accuracy than the Brook+ and FPGA implementations.**

Keywords- Quantum Monte Carlo, Field-Programmable Gate Arrays, Multi-core processors, Graphics Processing Units.

I. INTRODUCTION

Until a few years ago, single-core processors kept up with the trend predicted by Moore's law offering higher clock rates by packing more transistors on a chip using the latest transistor technologies. Cache memories have been used to mitigate the effects of the widening gap between processor and memory performance while optimizations such as pipelining, superscalar, and out-of-order execution have been used to exploit the available instruction level parallelism. The increased design complexity and power dissipation resulting from this trend necessitated a paradigm shift: the development of multi-core processors which use reduced clock rates and provide increased performance in an energy efficient manner. Graphics Processing Units (GPUs) and Reconfigurable Computing (RC) using Field-Programmable Gate Arrays (FPGAs) are emerging alongside multi-core processors as attractive platforms for accelerating computationally intensive applications. Present High Performance Reconfigurable Computing (HPRC) platforms provide a high-speed interconnect between the FPGA and the processor. GPUs, originally using fixed-function pipelines for rendering graphics, are now programmable enough for general-purpose use. Future supercomputing platforms are likely to be heterogeneous systems with one or more of the above technologies. Besides performance, the adoption of one or more of these platforms for scientific computing depends on factors such as numerical precision, ease of programmability, cost, power consumption, and reliability. GPU offerings from NVIDIA and AMD provide floating-point (single and double-precision) and current FPGAs with increased gate densities provide the flexibility to use floating-point, fixed-point, or a customized precision. On present GPUs, there is a huge gap in single versus double-precision floating-point performance and also a lack of double-precision support for certain mathematical functions on AMD/ATI GPUs [1]. Prior research efforts have shown that double-precision is not often required for the entire application. Mixed-precision and iterative refinement approaches have been proposed [2] and successfully demonstrated on FPGAs [3]. Languages such as Compute Unified Device Architecture (CUDA) [4] from NVIDIA and Brook+ [5] from AMD/ATI have alleviated the difficulty of programming the GPUs for general-purpose computing. OpenCL is a standard that intends to provide cross-platform support for processors and GPUs [6]. For programming FPGAs, a number of high-level languages have been developed but most languages are specific to the target RC platform. Hardware Description Languages such as VHDL and Verilog are still the best ways to exploit an FPGA's computational potential. Unfortunately, these languages require a deep low-level understanding of hardware design principles and make application development very time consuming. Our work provides a comparative study of various architectures such as single-core and multi-core CPUs, FPGAs, and GPUs for a Quantum Monte Carlo application. We describe our implementations and performance results, such as speedups and accuracy, on the NVIDIA and AMD GPUs as well as with FPGAs. Our FPGA VHDL implementation uses a general-purpose framework that allows us to reuse the design blocks; this reduces design time. For the CUDA implementation, we use a mixed-precision approach, using single-precision for most of the computations and double-precision only where necessary, leading to a significant improvement in performance over double-precision with little compromise in accuracy.

II. OVERVIEW OF QUANTUM MONTE CARLO

Quantum Monte Carlo methods are widely used in physics and physical chemistry to obtain the ground-state properties of atomic or molecular clusters [7]. This method provides a practical and efficient way of solving the many-body Schrödinger equation. Figure 1 shows the Variational Monte Carlo (VMC) algorithm. Step 3 of the algorithm consists of the computationally-intensive kernels requiring $O(N^2)$ (potential energy, wave function). For the GPU and FPGA implementations, we offload the kernel evaluations in Step 3 onto these platforms while Steps 1, 2, and 4 are retained on the host processor.

Research funded by National Science Foundation Grant, CHE-0625598

Step 1: Select a reference configuration, $R(x, y, z)$ for the system of atoms at random.

REPEAT (for I iterations)

REPEAT (for all configurations)

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

Step 3: Compute the ground-state properties (e.g., energy, wave function) of the atoms in the current configuration, R'. This step consists of the $O(N^2)$ calculations and chosen for hardware acceleration.

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

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

If $p \geq 1$, R' is accepted and retain the properties of this configuration.

If $p < rand()$, R' is rejected and R is retained and retain the properties of R.

UNTIL finished (for all configurations)

UNTIL finished (for all iterations)

Figure 1. Variational Monte Carlo Algorithm

III. IMPLEMENTATION

A. CPU Implementation

The x86 is a well-established platform in high performance computing [8]. As such, a double-precision baseline C implementation of an experimental version of QMC was used on the CPU (with -03 optimization). Pair-wise contributions are accumulated in-place to minimize memory use and keep data cache resident. Intel's Math Kernel Library (MKL) was tested to try to speed up the application, but we saw small speedups (1.04*x* faster for a cluster of 4096 atoms) at the expense of very complex code and increased memory footprint. Since QMC is embarrassingly parallel, we were able to run eight copies of the process on an 8 core (2 socket quad-core) Clovertown machine to achieve parallelism in performance testing. This, as expected, yielded near linear speedups while making explicit parallelism unnecessary.

B. Brook+ Implementation

AMD provides the Brook+ SDK for performing general purpose computing on their GPUs. It is based on the Brook project created at Stanford University [5]. Brook+ uses the streaming model [9], which simplifies programming by separating computation and communication [10].

Brook+ is an inherently parallel language with two components: kernels and streams. Streams are an abstract data structure, usually representing an array or matrix. Kernels are run many times to produce and use the elements in streams; they operate on streams that are declared as input or output. Input streams can only be read and output streams can only be written. A thread is created for each element in the output streams (the set of which is called the *domain*), where each thread produces one element [5]. For example, a kernel run on a 20x20 stream will have 400 threads, each thread computing one element. Threads may only write to their assigned element. As such, indexing into an output stream is forbidden in Brook+ as there is only one location that can be written by a kernel instance. Streams can be read from any location if the stream is declared using "[]." If "<>" is used to declare an input stream, it may only be read at the current location in the domain of execution [5].

We implemented the VMC algorithm in Brook+ in two ways. The first implementation treats each element in the output stream (potential, wavefunction, etc.) as the result of a single pair-wise particle interaction. This naturally maps onto the streaming model and each stream can be reduced to produce the total result. This naïve algorithm, uses $O(N^2)$ memory and has $O(N^2)$ threads executing in parallel. The second algorithm stores the sum of all interactions with a given particle in each element in a vector stream. The first element is the sum of all interactions with the first particle while the second element is the sum of all interactions with the second particle, and so on. This optimized algorithm uses $O(N)$ memory and provides $O(N)$ parallelism. While this in place reduction of interactions could be taken to its logical conclusion to sum the interactions in a single variable, this would provide no parallelism implicitly given by the streaming model. Both algorithms were implemented in single precision. The first algorithm performs the reductions on the GPU using a reduction kernel to avoid the overhead of transferring $O(N^2)$ elements while the second algorithm performs the reductions on the CPU using a double precision accumulator.

C. CUDA Implementation

Compute Unified Device Architecture (CUDA) is a programming environment provided by NVIDIA for general-purpose computation on the GeForce, Quadro, and Tesla GPUs [11]. The kernel offloaded to the GPU is specified as a computational grid of blocks of threads. The threads are grouped into warps, which use Single Instruction Multiple Data (SIMD) for instruction execution.

The partitioning scheme (number of blocks and threads) is chosen to keep the threads busy and hide the memory latencies. We use an $O(N^2)$ approach but only compute the contributions to the lower (or upper) triangular portion of the N x N matrix. We use 1D partitioning, dividing the rows of the matrix among a number of blocks that execute on the multiprocessors. Within a block, each thread calculates the interactions along the row, between its atom and other atoms in the system. The $O(N)$ (x, y, z) positions are transferred from the host to the device memory using the CUDA runtime functions [11]. We use the 16 KB shared-memory per block to store a subset of co-ordinate positions of the atoms [12]. The in-place reductions to obtain the accumulated energies and wave function along the rows are performed on the GPU, with $O(N)$ results sent back to the processor. We investigate the use of double-precision and mixed-precision for function evaluations, where we use single-precision for function evaluations and double-precision for the reductions.

D. FPGA Implementation

Our FPGA architecture is developed in VHDL [13] and targeted to the Cray XD1 HPRC [14] platform on a single Xilinx Virtex-4 LX160 FPGA [15]. We use a two-region approach, a generic-interpolation framework that can be

used to evaluate the functions, fixed-point precision, and deep pipelining [16]. The O(*N*) positions and interpolation coefficients for the evaluation of energy and wave function are transferred from the host memory to the on-chip Block RAMs on the FPGA using the Cray API functions. The FPGA architecture consists of a distance calculation block, which produces the distances used by the potential energy, and wave function calculation modules to produce a potential energy or wave function result every clock, which are accumulated to form O(*N*) partial results. These results are written to user registers and read within the QMC application on the host processor using the Cray XD1 API. The processor reconstructs the floating-point values of the energies and wave functions from the fixed-point results.

IV. RESULTS

The Brook+ implementation is targeted to an AMD Firestream 9170. The CUDA implementation is run on an NVIDIA Tesla C1060. The GPUs are connected via PCIe2 interfaces to the respective host processors. We target a single Virtex-4 FPGA on the Cray XD1. The QMC simulation is run for various cluster sizes for 10 configurations and a total of 400 iterations. Figure 2 shows the speedups of CUDA, FPGA, and Brook+ implementations over eight Intel Xeon X5355 processors. The optimized-Brook+ shows the best overall speedup of 16*x* over the baseline multi-core implementation for small clusters. The mixed-precision CUDA outperforms the Brook+ implementation for 4096 atoms. The double-precision CUDA implementation is the slowest. In order to better assess the suitability of a platform for the QMC application, we plot the relative errors in the total pair-wise potential energies for a single iteration over the CPU double-precision in Figure 3. The optimized-Brook+ and fixed-point FPGA implementations have the least relative errors, followed by the mixed-precision CUDA. Not shown in Figure 3 are the errors for single-precision and double-precision CUDA (which are 1.34×10^{-5} and 1.6555×10^{-12}, respectively).

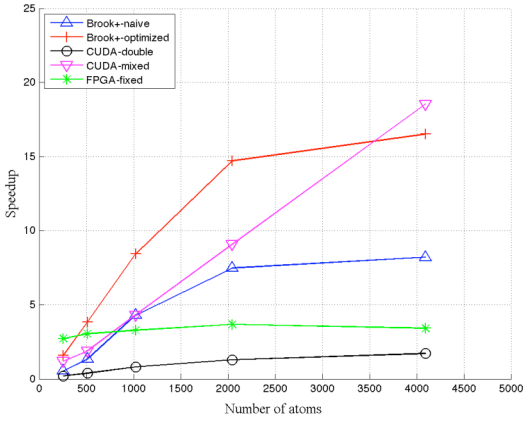

Figure 2. Speedup versus number of atoms

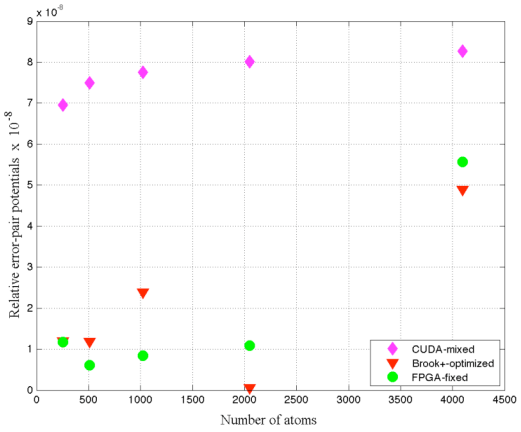

Figure 3. Relative error versus number of atoms

ACKNOWLEDGMENT

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.

REFERENCES

[1] AMD/ATI Release Notes, http://developer.amd.com/gpu_assets/BrookPlus_Release_Notes.pdf

[2] A. Buttari, J. Dongarra, J. Langou, J. Langou, et al., "Mixed Precision Iterative Refinement Techniques for the Solution of Dense Linear Systems," *International Journal of High Performance Computer Applications*, Vol. 21, pp. 457-466, 2007.

[3] J. Sun, G D. Peterson, O.O. Storaasli: High-Performance, "Mixed-Precision Linear Solver for FPGAs," *IEEE Transactions on Computers*, Vol. 57 , Issue 12, pp. 1614-1623, 2008.

[4] NVIDIA's CUDA: http://www.nvidia.com/object/cuda_home.html

[5] I. Buck, T. Foley, D. Horn, J. Sugerman, *Brook for GPUs*, 2003.

[6] OpenCL, http://www.khronos.org/opencl/

[7] J. Thijssen, "Computational Physics," *Cambridge University Press*, September 2007.

[8] *Processor Family share for 11/2008*. 2008 [cited 2009 April 8].

[9] AMD Stream Computing User Guide. 2008, AMD.

[10] W Dally. *Stream Computing*. [Lecture] 2008 June 10, 2008, Available from: http://www.youtube.com/watch?v=8x7OqjUNbyo.

[11] NVIDIA's CUDA Programming Guide: http://www.nvidia.com/object/cuda_develop.html

[12] L. Nyland, M. Harris, J. Prins, "Fast N-Body Simulation with CUDA," *GPU Gems 3*, Chapter 31, Addison Wesley, 2007.

[13] A. Gothandarman, G. D. Peterson, G. L. Warren, R. J. Hinde, R. J. Harrison, "A Hardware-accelerated Quantum Monte Carlo (HAQMC) framework for *N*-body systems," *Computational Physics Communications*, 2009 (accepted for publication).

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

[15] Xilinx Inc., http://www.xilinx.com/ipcenter/index.htm

[16] A. Gothandaraman, G. D. Peterson, G. L. Warren, R. J. Hinde, R. J. Harrison, "FPGA acceleration of a Quantum Monte Carlo application," *Parallel Computing*, Vol. 34, Issue 4-5, May 2008.