A GPU Support for Large Scale Quantum Chemistry Applications

Selva Kumar Sengottaiyan[1], **Fang Liu**[1], **Masha Sosonkina**[1]
[1] Ames Laboratory, Iowa State University

Abstract— *GPU/GPGPU computing has been used widely in scientific simulation to improve the performance on hybrid architectures. The quantum chemistry field has benefited greatly from using GPUs, including tasks such as visualization of molecular orbitals and computation of electronic structures. To gain significant success in using GPUs, a large amount of code rewriting and restructuring is required, which is done primarily by those who understand the algorithm in great detail. In this paper, two widely used quantum chemistry packages are investigated to identify the hot spots that can benefit most from GPUs, as well as be the least intrusive to the existing code base. The paper uses an experimental approach to integrate GPU capability without restructuring the application. Experimental results show that the bottleneck is in CPU–GPU data transmission. Additionally, a GPU-based DFTFOCK method is implemented in GAMESS/NWChem and a GPU-based eigensolver is integrated with NWChem successfully. Further performance tuning is ongoing.*

Keywords: GPU Computing, Eigensolver, Quantum Chemistry, GAMESS, NWChem

1. Introduction

Graphics processing unit (GPU) computing or general-purpose computing on graphics processing units (GPGPU) is the use of a GPU to do general purpose scientific and engineering computing. The model for GPU computing is to use a central processing unit (CPU) and GPU together in a heterogeneous co-processing computing model. While one part of the application runs on the CPU, another computationally intensive part can be accelerated by the GPU. GPUs are an excellent accelerator for data-parallel applications. From the user's perspective, the application just runs faster because it is using the high performance of the GPU to boost performance. Large gains in performance have been achieved through GPUs in recent years and GPUs have become ubiquitous in handhelds, laptops, desktops, and supercomputer clusters. The power of GPUs has been incorporated into simulations or experiments and shows a large benefit. However, a large amount of code rewriting and restructuring is often required. GPU computing has recently begun to be adopted in the quantum chemistry domain [7] from visualization of molecular orbitals to computation of electronic structures.

The General Atomic and Molecular Electronic Structure System (GAMESS) is a widely used computational chemistry package [3], [8] for *ab initio* molecular quantum chemistry. Using GAMESS, a variety of molecular properties, ranging from simple dipole moments to frequency dependent hyperpolarizabilities may be computed. GAMESS is capable of a very broad range of electronic structure theory calculations and is therefore very widely used, with an estimated user base of $150,000$ in more than 100 countries.

NWChem [12] is a high-performance computational chemistry software package that focuses on providing new and essential scientific capabilities to its users in the areas of the kinetics and dynamics of chemical transformations. Initially, the problems of interest focused on environmental issues, but recently NWChem has been applied to the examination of metal clusters, biological systems, nanostructures, and materials. Both GAMESS and NWChem offer a multitude of highly correlated methods, density functional theory (DFT) with many exchange correlation functionals. Additionally, NWChem provides plane-wave DFT with exact exchange and Car-Parrinello simulations, molecular dynamics with the AMBER and CHARMM force fields, and combinations of these methods.

GPU and CPU hybrid platform have become widely used during the past few years. Well designed algorithms can fully take advantage of the performance of GPUs. Since quantum chemistry applications such as GAMESS and NWChem have always been at the forefront of improving time to solution for platforms from desktops to supercomputers, it is natural for these codes to use the computing power of GPUs.

In this paper, after briefly introducing the GPU computing model, two GPU eigensolver packages are compared. Next, details of the integration work are given, followed by the experimental results, related work and conclusions.

2. GPU computing model

The success of GPGPUs in the past few years has been due in part to the ease of programming of the associated CUDA parallel programming model. In this programming model, the application developers modify their application to take the compute-intensive kernels and map them to the GPU. The rest of the application remains on the CPU. Mapping a function to the GPU involves extensively rewriting the function, usually in C or C++, to expose the parallelism in the function and adding keywords to move data to and from the GPU. The developer is tasked with launching $10s$ to $1000s$ of threads simultaneously. The GPU hardware manages the threads and does thread scheduling.

Figure 1 shows the GPU computing model in which the GPU is a compute device which serves as a co-processor for the host CPU. The GPU architecture consists of a scalable number of streaming multiprocessor (SM)s, a multi threaded instruction fetch and issue unit, and a read-only constant

cache. A SM consists of Scalar Processor (SP) cores, special function units for transcendentals, a multi threaded instruction unit, and on-chip shared memory. The SM creates, manages, and executes concurrent threads in hardware with zero scheduling overhead. To manage hundreds of threads running several different programs, it employs a new architecture single-instruction, multiple-thread (SIMT). The SIMT unit creates, manages, schedules, and executes threads in groups of 32 parallel threads called warps.

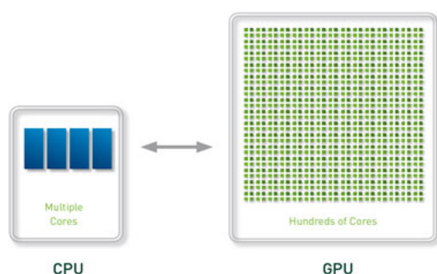

Fig. 1: GPU computing model

a) Threads: For GPU acceleration, a program must be decomposed into a large number of concurrently schedulable units so that groups of threads can execute the same code parallel to each other or suffer poor performance. In GPUs these groups, known as warps, consist of 32 threads. If threads within a warp diverge on a branch, the full warp is serially executed on each branch path, with threads converging into a single execution path only after the divergent branch is finished. CUDA also requires the organization of warps into larger units called blocks. Threads are assigned in units of blocks and can only communicate directly with other threads in the same block. Communication across blocks requires termination of the GPU kernel and data transfer into CPU memory where the required data manipulation can be performed. These issues limit the applicability of GPUs primarily to data parallel applications.

b) Memory hierarchy: Each thread has a private local memory. Each thread block has a shared memory visible to all threads of the block and with the same lifetime as the block. All threads have access to the same global GPU memory. The memory model gives the developer the freedom to choose global or local memory, which may affect the performance.

3. GPU eigenvalue computing packages

There are some eigensolver packages with GPU support which are publicly available. The GPU-accelerated linear algebra libraries (CULA) and Matrix Algebra on GPU and Multicore Architectures (MAGMA) packages utilize the NVIDIA CUDA parallel computing architecture to dramatically improve the computational speed of sophisticated mathematics.

CULA[1] is a high-performance linear algebra library that executes in a unified CPU/GPU hybrid environment. CULA provides easy interfaces through which an application can be integrated without extensive GPU programming experience. CULA can provide significant speedups over existing packages and supports both dense and sparse linear algebra. CULA features a wide variety of linear algebra functions, including but not limited to, least squares solvers (constrained and unconstrained), system solvers (general and symmetric positive definite), eigensolvers (general and symmetric), singular value decompositions, and many useful factorizations (QR, Hessenberg, etc.). All such routines are presented in four standard data types in the Linear Algebra PACKage (LAPACK) computations: single precision real (S), double precision real (D), single precision complex (C), and double precision complex (Z).

The **MAGMA** project[2] aims to develop a dense linear algebra library similar to LAPACK but for heterogeneous/hybrid architectures, starting with current "Multicore+GPU" systems. MAGMA provides linear algebra algorithms, designs and frameworks for hybrid many core and GPU systems that can enable applications to fully exploit the power that each of the hybrid components offer. This package also aims to solve a nonsymmetric linear system of equations by increasing speed for the price of relaxed accuracy.

a) Comparison between the two packages: CULA supports both dense and sparse linear algebra while MAGMA mainly supports dense linear algebra. CULA supports both single and double precision mathematics, and MAGMA supports only single precision. The eigensolver routine in CULA is a commercial code, while MAGMA is free. Both packages have multi-GPU support.

Based on requirement in NWChem,where single precision is sufficient. We have chosen MAGMA for our first stage of integration.

4. Integration of GPU support

The DFT algorithms in GAMESS and NWChem are integrated with GPU support, while the GPU-based eigensolver, MAGMA, is linked to NWChem. The experimental work examined these two well-known packages and identified the hot spots for GPU integration.

4.1 DFT integration

a) GAMESS DFT with GPU: In GAMESS, the typical DFT approach solves the Kohn-Sham equation [6] in which the total energy of the molecular system is a function of the positions of the atoms and one-particle densities. The approach in DFT is to assume an initial charge density and obtain successively better approximations of the density and energy. When the total energy is minimized with respect to the variational parameters, the resulting one-particle

[1]CULA tools `http://www.culatools.com`
[2]MAGMA `http://icl.cs.utk.edu/magma/index.html`

equations are exactly the same as the Hartree-Fock method except for the handling of the exchange terms and the way the electron exchange correlation is incorporated. DFT methods can yield results similar to those obtained with *ab initio* methods such as MP2, but at a substantially reduced computational effort.

The major hot spot in a GAMESS DFT energy and gradient calculation is in a routine called 'GRDDFT' which calculates the correlation correction to Self-Consistent Field (SCF) with an arbitrary set of density functionals. The calculation of 'GRDDFT' takes 94% of the total DFT calculation time. The routine consists of four parts:

1) Memory allocation
2) Geometry and symmetry setting (DFTSET).
3) Calculating the exchange correlation energy (DMATD).
4) Calculating the exchange correlation energy gradient (DFTGRAD).

The calculation of the energy exchange correlation matrix takes almost 99% of the total GRDDFT execution time. The function DMATD calculates the exchange correlation energy by looping over radial grids which in turn loop over the angular grids surrounding atoms. The looping over radial grids and angular grids takes almost 99% of the total DMATD execution time. Inside the loop over grid points, DFTFOCK (which adds the DFT exchange/correlation contribution to the Fock matrix) takes the largest amount (72%)of the execution time. Thus DFTFOCK is chosen as the routine to be executed on the GPU. Other reasons are that the data dependency of DFTFOCK as compared to other subroutines is minimal and the amount of parallelism inside the function is high.

Figure 2 shows the integration of the CPU Fortran code with the GPU Fortran code in GAMESS. Inside the GAMESS Fortran code, the application code that has a high degree of parallelism and that takes most of the CPU time is identified. Then the identified code is transfered into CUDA Fortran. In the above case, the DFTFOCK subroutine is identified as a hot spot and converted into CUDA Fortran. Then the changed code is compiled using the CUDA Fortran compiler (PGFortran). The compiled code is linked with GAMESS and the CUDA libraries.

b) NWChem DFT with GPU: NWChem contains a parallel implementation of the Hohenberg-Kohn-Sham formalism [4] of DFT which differs significantly from other *ab initio* methods in the treatment of the exchange-correlation term used in building the Fock matrix. The computationally intensive components of a DFT calculation include the fitting of the charge density, construction of the Coulomb potential, construction of the exchange correlation potential, and the subsequent diagonalization of the resulting equations. The GPU accleration of DFT in NWChem concentrates getting exchange-correlation contribution to the gradient and adding the Bonacic-Fantucci repulsive term [6].

Figure 2 shows the integration of the CPU Fortran code with GPU Fortran code in NWChem. As with the GAMESS

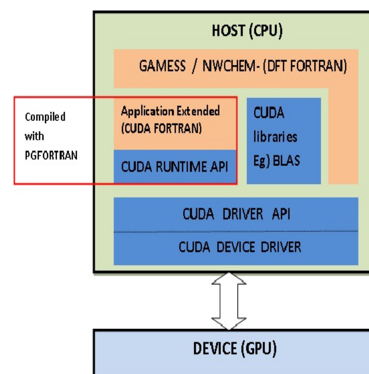

Fig. 2: GAMESS/NWChem DFT-GPU Programming

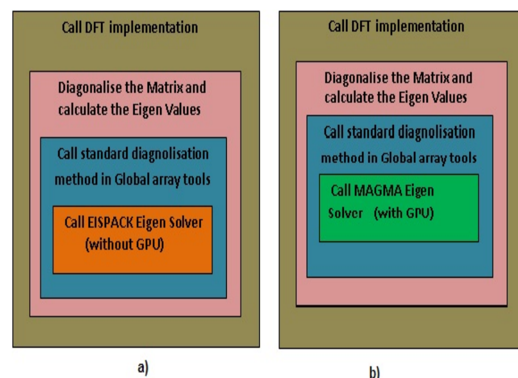

Fig. 3: Function calls inside the NWChem eigensolver implementation without and with GPU implementation

integration, the application code that has a high degree of parallelism and that takes most of the CPU time is identified. Then the identified code is changed into CUDA Fortran. The NWChem GPU kernel is similar to GAMESS GPU kernel (Figure 2) as in both cases the GPU kernel uses the CUDA runtime API and CUDA libraries. During integration of the GPU kernel in DFT for both NWChem and GAMESS, the GPU kernel implementation requires only minimum code rewriting as using the existing code is one of objectives of this GPU acceleration implementation.

4.2 Integration of NWChem DFT eigensolver with MAGMA

Figure 3a shows the order of function calls inside the NWChem the DFT eigensolver DFT implementation without GPU implementation. Figure 3b shows the changed order of function calls inside the NWChem DFT eigensolver implementation with GPU implementation. Figure 4 shows the integration of the CPU Fortran code with GPU Fortran code (MAGMA eigenvalue package) in NWChem. The calculation of eigenvalues and eigenvectors is a time consuming part in the Fock matrix updating procedure. The MAGMA library of C functions offers two LAPACK-style interfaces, referred to as the GPU interface and the

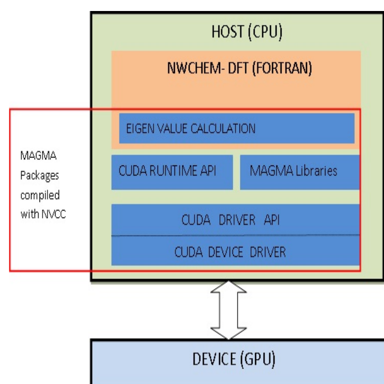

Fig. 4: NWChem-GPU MAGMA Interface

CPU interface. The GPU interface takes input and produces results in the GPU's memory, whereas the CPU interface produces results in the CPU's memory. The GPU and CPU interfaces, although similar, are not derivatives of each other, but instead have different communication patterns. The GPU function returns or sets the logical flag indicating whether the GPU interface is to be used for calculations involving the specified MAGMA matrix. NWChem normally calculates the eigenvalue by using the EISPACK available with the GA tool.

Global Array eigenvalue calculation

The Global Array diagnolization subroutine calls a sequence of subroutines from the eigensystem subroutine package (EISPACK) [10] to find the eigenvalues and eigenvectors (optional) of a real symmetric matrix. In NWChem for serial execution, the eigensolver 'rs' from EISPACK is used. (The eigensolver in ScaLAPACK [1] is called only for parallel execution.)

In order to demonstrate the GPU version eigensolver can benefit NWChem, the EISPACK eigenvector calculation in NWChem will be replaced by the MAGMA GPU solvers. Since NWChem invokes the 'rs' solver through the Global Array interface, the Global Array diagonalization subroutine EISPACK calls are replaced by MAGMA calls. The MAGMA eigensolver routine is Magma_dsyev which is very similar to the analogous LAPACK routine. The major difference between the 'rs' subroutine and Magma_dsyevd is that 'rs' uses the whole real symmetric matrix whereas Magma_dsyev uses the lower or upper triangular matrix. By switching from the 'rs' routine to the Magma_dsyevd, the matrix needs to be converted to a new format.

5. Experiments

All the tests are conducted at the US Department of Energy Ames Laboratory GPU computing cluster - Exalted, which consists of 26 nodes, each having six 2.66 GHz Intel Xeon processors, 8 nodes have 4 NVIDIA Tesla C2070 GPUs and 18 nodes have 2 NVIDIA Tesla C2070 GPUs. The nodes are connected via Mellanox QDR InfiniBand and each node has 24GB of RAM. Despite having two GPUs per node available, we focused on using only a single GPU per node. The speedup here is defined as the ratio of the existing sequential algorithm execution to the parallel execution time.

5.1 GPU implementation of DFT in GAMESS

This experiment on the DFT method in GAMESS focuses on performance improvement of the algorithm in terms of speed up. As explained in Section 4.1, the DFTFOCK routine is separated from the existing application and is tested with various problem sizes (Fock matrix sizes) to find the actual performance gain in terms of speed up. Various scenarios with different optimization techniques are tested and results are explained below.

Scenario 1: The GPU CUDA memory model comprises various memory spaces, which differ enormously in latency times, availability of caches, etc. Particularly important features are global and shared memories, the former being an off-chip memory and thus featuring high-access latency; the latter being on-chip memory and thus having reduced latency. This scenario tests the global and shared memory usage inside the GPU. The experiment results with problems of various matrix size are shown in Table 3. Results are after optimization for the number of threads and blocks based on the problem size. In order to find the break point where DFTFOCK can start to take advantage of the GPU, the matrix size is continually increased. The GPU implementation shows better timings for problems with matrix size of 10,000 or more (Table3). For the problem size of 20,000, there is not much difference in performance between CPU implementation and CPU-GPU implementation using global memory, but the CPU-GPU implementation using shared memory increases the speed up from the CPU implementation (Table 3, columns 1&3) 2 times and the CPU-GPU shared memory implementation also increases the speed up from the global memory implementation (columns 2&3) by about 2 times. The experimental results prove that the GPU implementation should involve less kernel calls and memory allocations for better performance. Shared memory is exploiting the availability of on-chip shared memory by enabling kernels to load all needed field components, including the components corresponding to adjacent blocks and provides huge performance gains in terms of GPU and CPU execution time. Such gains are due both to the high efficiency of shared-memory access, and due to the limited number of separate memory transaction issued for each thread i.e., reducing the effective number of memory read operations for each thread. GPU speed up can be increased more if memory usage is optimized. Hence it is necessary to carefully design and implement kernel codes in order to minimize the number of global-memory reads, by making use of the other available kinds of memory.

Scenario 2: In this scenario, we used various Fock matrix sizes to explain the factors affecting CPU-GPU implementation. An experiment (CASE A) is built with 50 outer loops and 12525 inner loop iterations. This computation takes about 20 microseconds in the CPU implementation. The total execution time for the CPU-GPU implementation is 70

Table 1: **Comparison of CPU, global memory GPU and shared memory GPU times in seconds (s)**

Size of Matrix	Total CPU execution time(s)	CPU+GPU execution time(Global)(s)	CPU+GPU execution time (Shared (s)
50	$20*10^{-6}$	$70 *10^{-3}$	$70*10^{-3}$
500	$100*10^{-6}$	$74 *10^{-3}$	$74*10^{-3}$
1000	$2.145237*10^{-3}$	4.213462	2.561576
5000	1.80	27.8	17.4
10000	211.2	277.0	173.6
20000	1732.1	1687.1	973.2

milliseconds. As expected, the GPU time is much greater than the CPU execution time. The breakdown of execution time for the CPU-GPU implementation (Table 2) shows that memory allocation time in the CPU-GPU code far exceeds the total CPU execution time. So experiments (B,C,D,E) are built with a problem size of 500 outer loops and 125250 inner loop iterations and execute with various GPU-CPU implementation/optimization techniques. The results are shown in Table 2. For all cases (B,C,D,E), each GPU block has 512 threads and the warp size is 32.

CASE B : Implementation of inner loop (DFTFOCK) in the GPU kernel, allocating the memory outside the kernel call. In this case, a small kernel size and large number of threads and memory allocation calls are needed in the GPU.
CASE C : Implementation of inner loop (DFTFOCK) in the GPU and allocating the memory inside the CUDA kernel.
CASE D : Implementation of outer loop (DFTFOCK) in the GPU kernel, allocation of memory outside the outer loop with a single kernel. In this case, the kernel size is increased but the number of threads inside the problem is decreased.
CASE E : Implementation of outer loop (DFTFOCK) in the GPU kernel, allocation of memory outside the outer loop and splitting the GPU kernel into two parts. In this case, the kernel size needed for implementation is increased and the number of threads inside the problem is decreased.

From Table 2, CASE A shows that the kernel call in a loop is the bottleneck as each kernel initialization takes a lot of time. Cases B, C, D and E also demonstrate that allocation takes more time than kernel execution, and therefore GPU implementation should involve less kernel calls and memory allocations.Kernel execution time is in micro seconds range where as total execution time is in milliseconds range. The best execution time is from the results of CASE D (i.e., the implementation involves less GPU memory allocations and less number of kernel calls irrespective of kernel size). Test case D has increased kernel size (i.e., more functions) compared to the kernels in other test cases, irrespective of which it almost takes same time as the other test cases. Thus, for GPUs, global inter-thread synchronization from kernel calls is very costly, because it involves a kernel termination and a new kernel call overhead from the host. The application specific software optimization is critical to fully utilize compute/bandwidth resources for both CPUs and GPU.

5.2 DFT acceleration using GPU in NWChem

The experiment on the DFT-FOCK method in GAMESS focuses on performance improvement of the algorithm in terms of speed. As explained in Section 4.1, the DFT GPU implemention is also done in the NWChem package. The problem size is determined by the number of atoms used in the input to the DFT NWChem package. The number of GPU threads inside each block is 512 and the warp size is 32. Table 3 shows the speedup in DFT using the GPU for various problem sizes. With the increase in problem size, the speedup increases. Due to data transfer latency, the benefit of using the GPU appears only if the problem size is increased to more than 2000 atoms. The size of the kernel is also very important in determing the speedup in the GPU.

Figure5, shows the running time for different numbers of threads for a problem size of 10000 atoms. Figure5 explains performance of GPU in terms of number of threads per block. The decrease in time with an increase in GPU occupancy (threads) shows that more performance gain is possible. A more occupancy indicates that the application fully exploits available processing units. Unfortunately, the amount of shared memory and registers used by each thread block limits the occupancy value. The size of thread blocks and/or shared-memory and registers usage must be designed with care in order to maximize the occupancy. The speedup increases with the increase in problem size but is greatly limited by data transfer between the CPU and GPU. The limitation in speedup highlights the importance of software optimizations (memory and GPU occupancy), and an application driven design methodology. s

5.3 MAGMA with eigensolver

The MAGMA package is integrated with NWChem package and the eigensolver of EISPACK available with the GA tool is replaced with MAGMA eigensolver. We tested the integration for various input molecules like Ozone, Cr_2. Since the input molecules available for testing have relatively small matrix size, we separated the Global Array eigensolver from the NWChem calculation and ran the tests separately from the NWChem execution. This allowed us to experiment with larger matrix sizes in the MAGMA-based eigensolver. Specifically, the test case uses the Global Array tools (which NWChem also calls to calculate the eigenvalues) where the existing EISPACK sequential eigenvalue solver is replaced with the GPU eigenvalue solver from the MAGMA package.

Table 2: **Time comparisons for various DFTFOCK CUDA implementations**

Experiment	**Total execution time(Milli s)**	**Memory allocation time(Milli s)**	**Total Kernel execution time(Micro s)**
A	70	70	87
B	3.8 $*10^3$	73	270
C	76	73	270
D	74	73	87
E	75	73	110

Table 3: **Comparison between CPU and GPU implementation times of DFT in NWChem**

Size of Matrix	**Total CPU execution time(s)**	**CPU+GPU execution time (Shared)(s)**	**Speed up**
10	0.000	4.3954	Nil
100	$1.326*10^{-3}$	7.8526	Nil
1000	15.1234	17.2368	Nil
2000	38.3678	27.8459	1.3
5000	228.398	80.2697	3
10000	876.679	158.9643	5

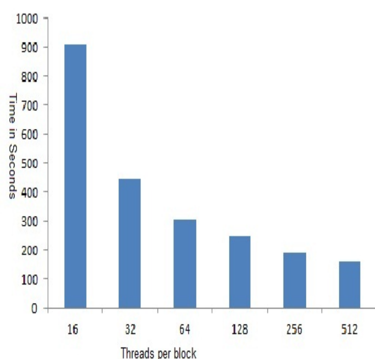

Fig. 5: Timings in DFT calculations using various number of threads in NWChem

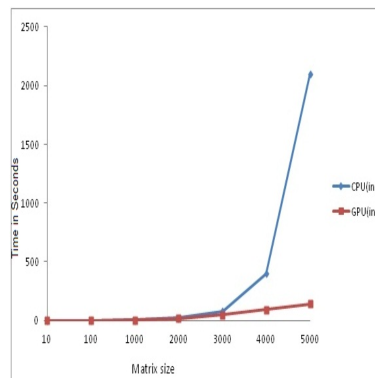

Fig. 6: Speedup in eigenvalues calculation using MAGMA

Table 4 shows the speedup with various matrix sizes. If the matrix size is less than 2000 there is no speed up. So a GPU implementation for the eigenvalues can be effective if the size of the matrix is around 2000. The speedup increases with increase in matrix size.

Figure 6 shows execution time for various matrix sizes on CPU and GPU respectively. The speedup starts to show when the matrix size reaches 2000. To summarize, it is advisable to keep data on the GPU memory, coalesce global memory accesses to reduce latency of data transfer, take advantage of shared memory, and to use hybrid code with double-precision applications.

6. Related work

Several other quantum chemistry applications have been implemented on GPU to exploit the greater level of parallelism. The strategy and optimization techniques for back porting an optimized GPU kernel to a multi-core CPU platform for the application TeraChem - a quantum chemistry code that was developed from the ground up to run on NVIDIA GPUs - is discussed in [13]. For this study, one of TeraChem's largest and most complex GPU kernels is considered. This kernel is used to calculate the electron repulsion integrals involving d-functions. It also investigates which CPU-specific optimizations can be applied to improve performance of the backported kernel. The exploitation of Quantum Monte Carlo algorithms with multiple forms of parallelism and its package simulation code portability to the NVIDIA CUDA GPU platform are discussed in [2]. The restructuring of the CPU algorithms to express additional parallelism, minimize GPU-CPU communication, and efficiently utilize the GPU memory hierarchy is important in porting the CPU code to GPU [2]. The restricted Hartree-Fock method is implemented on a multi-GPU system (a conventional cluster outfitted with GPUs) and its effectiveness is demonstrated [9]. According to [11], GPUs can significantly

Table 4: **Speedup in eigenvalues calculation using MAGMA**

Size of Matrix	Total CPU execution time(s)	GPU execution time(s)	SpeedUp
10	0.00001	0.001300	Nil
100	0.00399	0.399	Nil
1000	2.4650	2.4550	Nil
2000	22.91551	19.17751	1.2
3000	80.09583	50.1824	2
4000	400.1783	95.3697	4.19
5000	2100.7624	143.2457	15

outpace commodity CPUs in the central bottleneck of most quantum chemistry problems. It also explains the method to separate memory bound operations by modifying the algorithm and the memory scheme. It also demonstrates speedups are readily achievable for chemical systems of practical interest, and the inherent high level of parallelism results in complete elimination of inter-block communication. Due to the relative number of single and double precision cores, the best performance for GPU accelerated code is achieved when performing operations at single precision. It also discusses various issues that come with GPU implementations like memory transfer, accuracy, and thread consistency [5].

7. Conclusion and future work

In this work, a GPU integration for two widely used quantum chemistry packages, GAMESS and NWChem, is successfully performed. The exploration of two applications gives insights into GPU needs, and the experiments results demonstrate that there is a trade-off between performance gains and the ease of integration. This work focuses mainly on the experimental exploration of two large code bases: it pinpoints the most time consuming part in the DFT algorithm and links the GPU based eigensolver for NWChem. The intention was to integrate the GPU support without much source code changes. To accomplish this goal, the standalone units are identified, such as the eigensolver in the DFT algorithm. However, in order to fully exploit the available GPU, several software strategies have to be carefully designed and implemented. Currently, the performance bottleneck is in the data transfer between CPU and GPU. Performance tuning and further investigation of adapting GPU to more complex algorithms used in computational chemistry is ongoing.

Acknowledgment

This work was supported in part by Ames Laboratory (Iowa State University) under the contract DE-AC02-07CH11358 with the U.S. Department of Energy, by the Director, Office of Science, Division of Mathematical, Information, and Computational Sciences of the U.S. Department of Energy under contract number DE-AC02-05CH11231, and by the National Science Foundation grants NSF/OCI – 0749156, 0941434, 0904782, 1047772. The authors are have benefited from many helpful discussions with Professors Theresa L. Windus and Mark S. Gordon and their students at Iowa State University.

References

[1] L. S. Blackford, J. Choi, A. Cleary, E. D'Azevedo, J. Demmel, I. Dhillon, J. Dongarra, S. Hammarling, G. Henry, A. Petitet, K. Stanley, D. Walker, and R. C. Whaley. *ScaLAPACK Users' Guide*. Society for Industrial and Applied Mathematics, Philadelphia, PA, 1997.

[2] K. Esler, J. Kim, L. Shulenburger, and D. Ceperley. Fully accelerating quantum monte carlo simulations of real materials on gpu clusters. *Computing in Science Engineering*, PP(99):1, 2010.

[3] Mark S. Gordon and Michael W. Schmidt. Advances in electronic structure theory: Gamess a decade later. *Theory and Applications of Computational Chemistry:the first forty years, C.E.Dykstra, G.Frenking, K.S.Kim, G.E.Scuseria (Editors)*, 2005.

[4] P. Hohenberg and W. Kohn. Inhomogeneous electron gas. *Phys. Rev.*, 136:B864–B871, Nov 1964.

[5] Karl Wilkinson Kevin J. Naidoo and Kyle Fernandes. Accelerating scientific computing code in fortran: The quantum chemistry project. *PGI Insider*, oct 2011.

[6] W. Kohn and L. J. Sham. Self-consistent equations including exchange and correlation effects. *Phys. Rev.*, 140:A1133–A1138, Nov 1965.

[7] Wen mei W. Hwu. *GPU Computing Gems Emerald Edition*. Morgan Kaufmann Elsevier, 2011.

[8] Michael W. Schmidt, Kim K. Baldridge, Jerry A. Boatz, Steven T. Elbert, Mark S. Gordon, Jan H. Jensen, Shiro Koseki, Nikita Matsunaga, Kiet A. Nguyen, Shujun Su, Theresa L. Windus, Michel Dupuis, and John A. Montgomery, Jr. General atomic and molecular electronic structure system. *J. Comput. Chem.*, 14(11):1347–1363, 1993.

[9] Guochun Shi, V. Kindratenko, I. Ufimtsev, and T. Martinez. Direct self-consistent field computations on gpu clusters. In *Parallel Distributed Processing (IPDPS), 2010 IEEE International Symposium on*, pages 1 –8, april 2010.

[10] B. T. Smith, J. M. Boyle, J. J. Dongarra, B. S. Garbow, Y. Ikebe, V. C. Klema, and C. B. Moler. *Matrix Eigensystem Routines — EISPACK Guide*, volume 6 of *Lecture Notes in Computer Science, Editors: G. Goos and J. Hartmanis*. 1976.

[11] I.S. Ufimtsev and T.J. Martinez. Graphical processing units for quantum chemistry. *Computing in Science Engineering*, 10(6):26 –34, nov.-dec. 2008.

[12] M. Valiev, E.J. Bylaska, N. Govind, K. Kowalski, T.P. Straatsma, H.J.J. Van Dam, D. Wang, J. Nieplocha, E. Apra, T.L. Windus, and W.A. de Jong. Nwchem: A comprehensive and scalable open-source solution for large scale molecular simulations. *Computer Physics Communications*, 181(9):1477 – 1489, 2010.

[13] Dong Ye, A. Titov, V. Kindratenko, I. Ufimtsev, and T. Martinez. Porting optimized gpu kernels to a multi-core cpu: Computational quantum chemistry application example. In *Application Accelerators in High-Performance Computing (SAAHPC), 2011 Symposium on*, pages 72 –75, july 2011.