# HARNESSING THE POWER OF GRAPHICS PROCESSING UNITS TO ACCELERATE COMPUTATIONAL CHEMISTRY

By

Yipu Miao

A DISSERTATION

Submitted to
Michigan State University
in partial fulfillment of the requirements
for the degree of

Chemistry - Doctor of Philosophy

2015

# ABSTRACT

HARNESSING THE POWER OF GRAPHICS PROCESSING UNITS TO
ACCELERATE COMPUTATIONAL CHEMISTRY

By

Yipu Miao

Electron Repulsion Integral (ERI) and its derivative evaluation is the limiting factor for self-consistent-field (SCF) and Density Functional Theory (DFT) calculations. Therefore, calculation of these quantities on graphical processing Units (GPUs) can significantly accelerate quantum chemical calculations. Recurrence relations, one of the fastest ERI evaluation algorithms currently available, are used to compute ERIs. A direct-SCF scheme to assemble the Fock matrix and gradient efficiently is presented, wherein ERIs are evaluated on-the-fly to avoid CPU-GPU data transfer, a well known architectural bottleneck in GPU specific computation. A machine-generated code is utilized to calculate different ERI types efficiently. However, only s, p and d ERIs and s, p derivatives can be executed on GPUs using the current version of CUDA and NVidia GPUs. Hence, we developed an algorithm to compute f type ERIs and d type ERI derivatives on GPUs. Our benchmarks shows the performance GPU enable ERI and ERI derivative computation yielded speedups of 10~100 times relative to traditional CPU execution. An accuracy analysis using double-precision calculations demonstrates the accuracy is satisfactory for most applications. Besides *ab inito* quantum chemistry methods, GPU programming can be applied to a number of computational chemistry applications, for example, The Weighted Histogram Analysis Method (WHAM), a technique to compute potentials of mean force. We present an implementation of multidimensional WHAM on Graphical Processing Units (GPUs), which significantly accelerates its computational performance. Our test cases, that simulate two-dimensional free energy surfaces, yielded speedups up to 1000 times in double precision. Moreover, speedups of 2100 times can be achieved when single precision is used whose use introduces errors of less than 0.2 kcal/mol. These applications of GPU computing in computational chemistry can significantly benefit the whole computational chemistry community.

*This dissertation is dedicated to my parents, girlfriend, and all those, whose support, encouragement, and personal sacrifice have made this research possible*

# ACKNOWLEDGMENTS

First of all I want to thank my adviser, Professor Kenneth Merz, Jr from Michigan State University and previously University of Florida, for providing me the opportunity to pursue my Ph.D. in United States and supporting and guiding me as my advisor during the five and half years of my Ph.D. Research.

Special thanks to my committee members, Professor Metin Aktulga, Professor Katharine Hunt and Dr. Dirk Colbry. Moreover, I want to thank Professor Rodney Bartlett, Professor Beverly Sanders, Dr. Erik Deumens and Dr. Benjamin Smith who were part of my original committee at University of Florida for their help before my transfer to MSU with Professor Merz.

I sincerely want to thank, Dr. Mark Burger and the many engineers from NVidia Cooperation who supported myself and my research group with the-state-of-art facilities for software development and research use. Their help was was a key to our success. After my Ph.D. I will be taking up a position with the well respected company, Amazon.com Inc.. Finally, I want to thank Professor Shuhua Li from Nanjing University, China, who helped me understand the foundations of quantum chemistry and opened the door to scientific research.

I want to give my heartfelt thanks to my parents Zhiguo Miao and Xiulan Ou who emotionally and financially supported me during my Ph.D. career. I also want to thank my girlfriend, Jiamin Liang, for her tireless support of me during my Ph.D.

Many group members from Merz group, especially Dr. Yue Yang, Dr. Nihan Ucisik, Dr. Dhruva Chakravorty, Dr. Alex Fu, To-be Dr. Li-Li Pan, Mr. Pengfei Li, Dr. Zheng Zheng, and Dr. Xue Li provided fruitful discussions during my research career, and Mona Minkara, Ting Wang, Nupur Bansal, Zhuoqing Yu and Lin Song also provided support to me during my Ph.D. . The UFHPC and MSU iCER supported our group with facilities and HPC courses, without which, my research would have been less successful, so I want to thank Dr. Erik Deuemns, Dr. Charles Taylor, Dr. Craig Prescott, Dr. Ying Zhang of the UF HPC and Dr. Kenneth Merz, Dr. Dirk Colbry and Dr. Benjamin Ong of the MSU iCER team.

The Department of Chemistry of the University of Florida and the Department of Chemistry at Michigan State University provided me with teaching assistant positions during my Ph.D. career and guided towards the completion of the Ph.D. degree in Chemistry. The Department of Computer and Information Science and Engineering of the University of Florida bestowed a Master of Science degree in Computer Engineering, thereby strengthening my knowledge of the computer science field. I want to thank all faculty and staff of these departments, which had an influence on my life and career.

Last but not least, I also want to thank all of my friends in Gainesville, FL (the Gator Nation) and East Lansing, MI for all their help and support.

# PREFACE

I was fortunate enough to work on a multitude of projects in computational chemistry during my five-year studies towards my Ph.D. degree with Professor Kenneth Merz, Jr. I focused on Graphic Processing Units (GPU) acceleration on computational chemistry and computational biology methods, but also had great opportunities to explore the properties of molecules especially biochemical molecules with potential application for drug design. These projects, which were both interesting and challenging, allowed me to gain experience in several fields especially at the interface of computer science and computational chemistry.

The fundamental idea of my research projects was to speed up traditional computational chemistry calculations using current generation computation technology with or without modification of the computational methods employed. It is my strong belief that computational chemistry will be a powerful tool for a broad range of chemistry disciplines, but many of these applications of computational chemistry methods continue to be limited by the speed of the computations rather than by their accuracy and theoretical justification. Therefore, I focused my efforts on the power of high-performance computing especially GPU-programming and its role in improving computational chemistry.

My dissertation focuses on three computational chemistry problems that are relevant to scientific research in computational chemistry. I. Acceleration of electron repulsion integral (ERI) evaluation on GPUs. II. Acceleration of the first order derivatives of ERIs (including high-angular momentum integrals) on GPUs. III. Acceleration of Molecular Dynamic trajectory analysis using WHAM on GPUs.

# TABLE OF CONTENTS

## LIST OF TABLES

# LIST OF FIGURES

# KEY TO ABBREVIATIONS

| | |
|---|---|
| AMBER | Assisted Model Building with Energy Refinement |
| CPU | Central Processing Unit |
| CUBLAS | the NVIDIA CUDA Basic Linear Algebra Subroutines library |
| CUDA | Compute Unified Device Architecture |
| CUFFT | the NVIDIA CUDA Fast Fourier Transform library |
| DFT | Density Functional Theory |
| DP | Double-Precision |
| DRAM | Dynamic Random-Access Memory |
| ERI | Electron Repulsion Integral |
| FLOPS | Floating-point Operations Per Second |
| GPU | Graphic Processing Unit |
| HF | Hartree Fock Theory |
| HRR | Horizontal Recurrence Relation |
| PDB | Protein Data Bank |
| PMEMD | Molecular Dynamics using Particle mesh Ewald |
| PMF | Potential Mean Force |
| MD | Molecular Dynamics |
| MM | Molecular Mechanics |
| QM | Quantum Mechanics |
| QM/MM | Mixed Quantum Mechanics and Molecular Mechanics |
| RHF | Restricted Hartree Fock Theory |

| | |
|---|---|
| RMSD | Root-Mean-Square Deviation |
| SCF | Self Consistent Field method |
| SIMD | Single-Instruction, Multiple Data |
| SP | Single-Precision |
| VRR | Vertical Recurrence Relation |
| WHAM | The Weighted Histogram Analysis Method |

# CHAPTER 1. INTRODUCTION TO GPU PROGRAMMING, AND ITS APPLICATION TO QUANTUM CHEMISTRY

## 1.1. INTRODUCTION TO GPU PROGRAMMING

In the past few years, the computing capability of Graphic Processing Units or GPUs has increased dramatically as Figure 1.1 shows. From this figure we see that the GPU FLOPS (Floating-point Operations Per Second) of DP(double precision) and SP(single precision) have increased exponentially over the past 14 years, following Moore’s Law, while the corresponding CPU FLOPS counts have not. Meanwhile, as Figure 1.2 shown, the bandwidth of both GPUs and CPUs have dramatically increased largely driven by the insatiable demand for graphics and high-performance computing that involve highly parallel multithreaded processors with tremendous computational power and high memory bandwidth. The Telsa, Fermi and Kepler architectures were released by NVidia between the years of 2008 to 2013 and we were fortunate enough to have had early access to them for benchmarking purposes They feature fast double-precision computing and dynamic parallelism with the former being critical to scientific computing and the latter to maximize the potential of GPU computing. The next two generation NVidia GPUs, the Maxwell and Volta microarchitectures, have been announced by NVidia and are expected to be release in the near future. They will be featured with Unified Virtual Memory and stacked DRAM that will make it easier for the CUDA developer and overcome some known drawbacks that have limited performance.

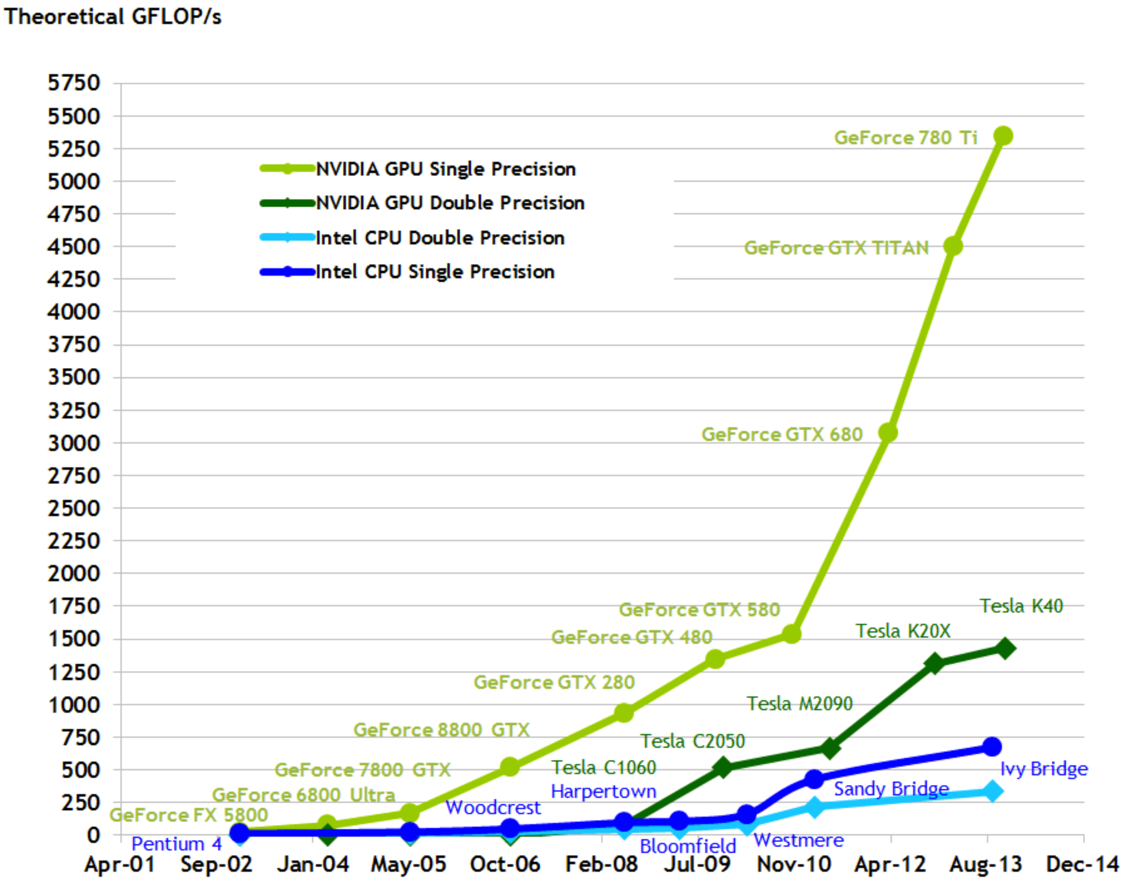


**Figure 1.1.** The FLOPS counts of CPU and GPU architectures as a function of time. Data is sourced from reference 1.

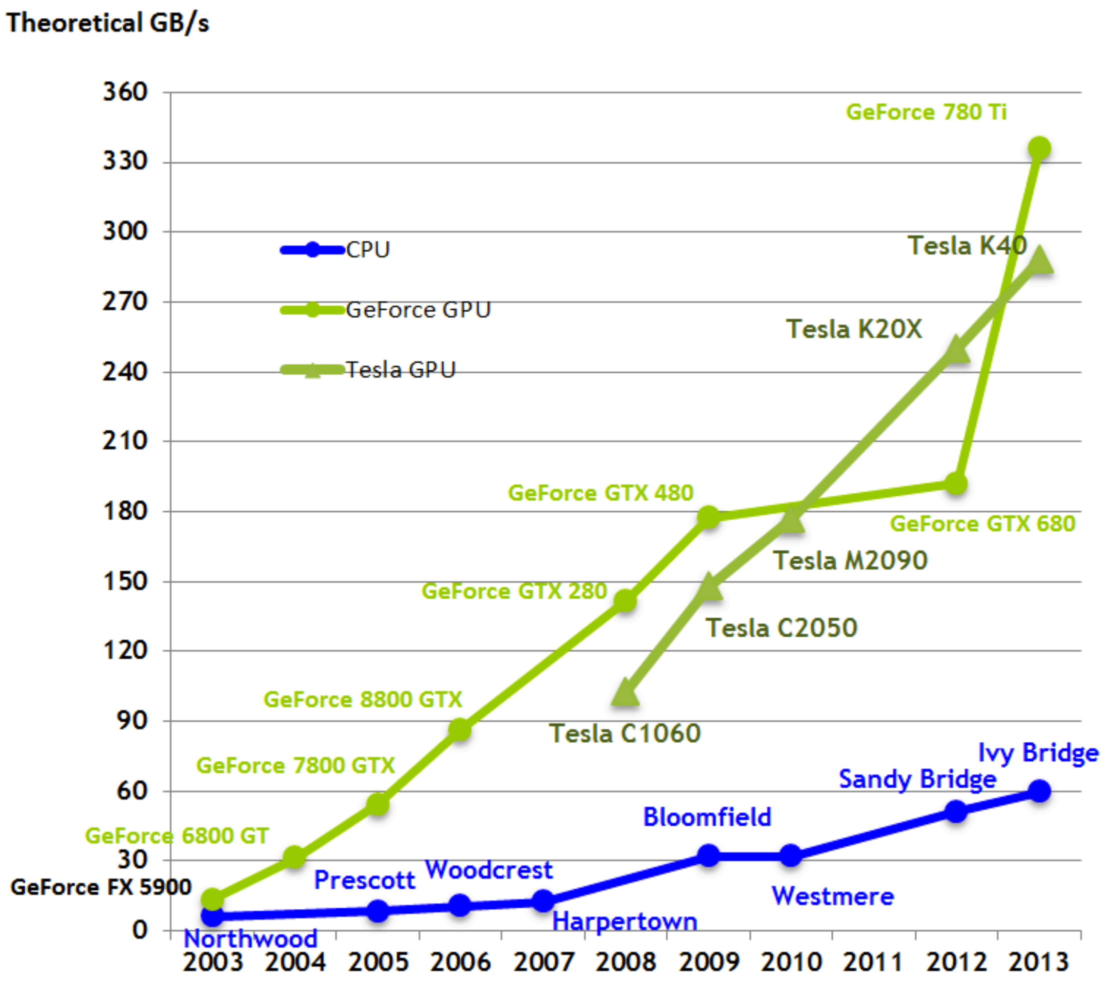


**Figure 1.2.** The bandwidth of CPU and GPU architectures as a function of time. Data is sourced from reference 1.

The main reason for the discrepancy in FLOP count between GPUs and CPUs is the former is specialized for compute-intensive, highly parallel computing driven by the needs of graphical rendering. Hence, GPU's are designed to have more transistors that are devoted to massive data processing rather than flow control as shown in Figure 1.3. Therefore, GPU's are better suited to address problems that can be split into data-parallel and data-independent computations; thus, massively parallel arithmetic threads instead of large data caches can mask memory access latency. CPUs on the other hand, with sophisticated flow control and data caching, are more suitable for calculations that are data dependent or are less arithmetically demanding.

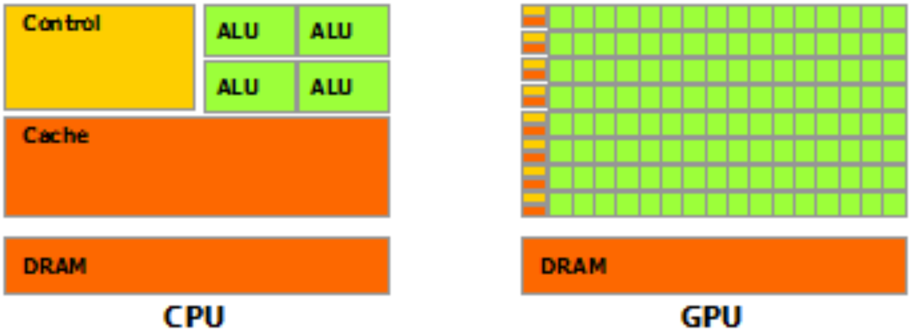


**Figure 1.3.** Simplified model of CPU and GPU architectures, illustrating that GPU's are designed for massive data processing while CPUs depend more on flow control and data caching.

With the growth of GPU computing power, computational chemists have ported a number of applications to GPU architectures, which are summarized in Figure 1.4 per NVidia. Computational chemists have used GPUs extensively to treat a wide range of problems[4], including AMBER PMEMD[5,6] (molecular dynamics simulation), quantum Monte Carlo[7], DFT (density functional theory), SCF (self-consistent-field)[8-11] and post

HF (Hartree-Fock) theory[12]. In this way, computationally intensive scientific applications, which previously required expensive supercomputing facilities, are now within reach to average users using relatively low-cost GPU cards. We will go into more detail regarding the current state of quantum chemistry on GPUs below.

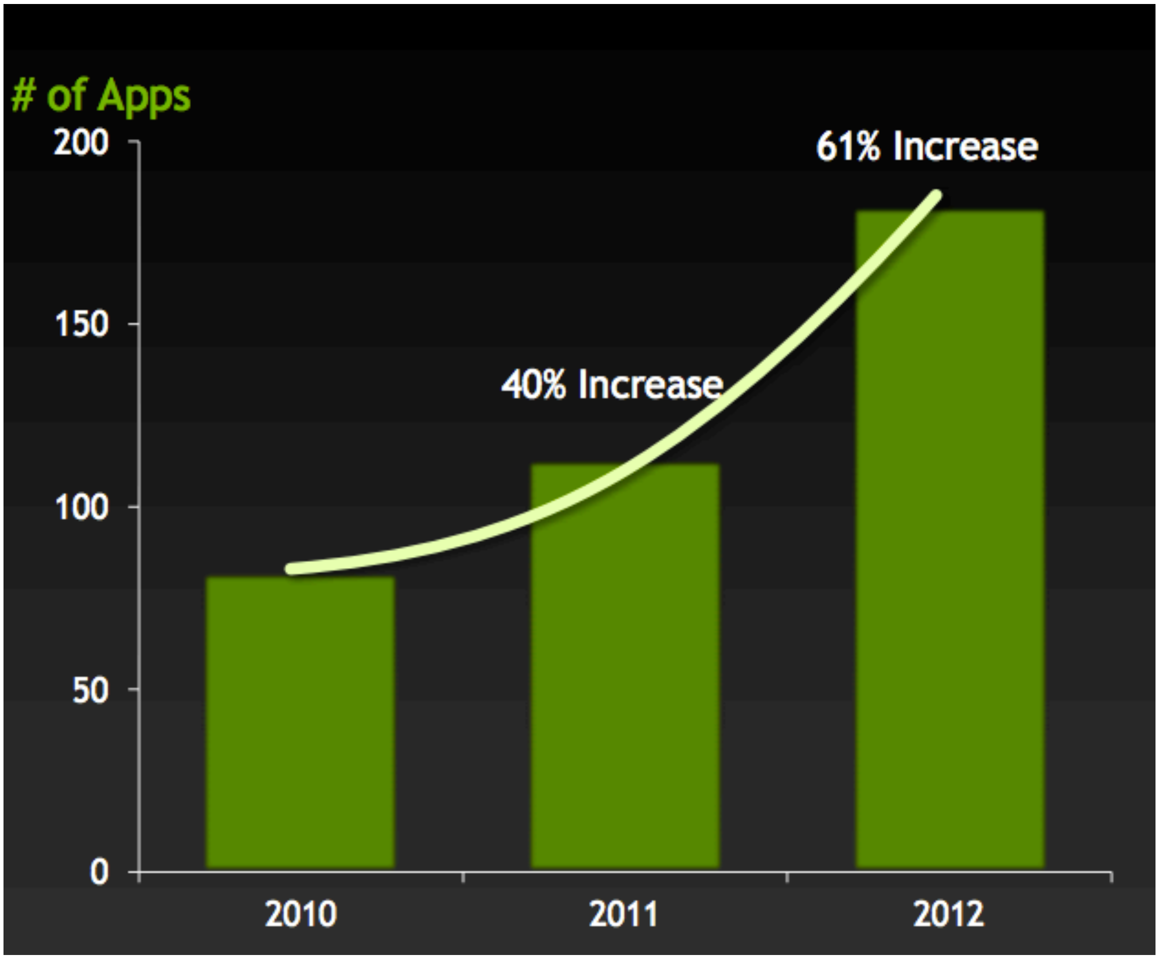


**Figure 1.4.** Number of Computational chemistry GPU applications published in the last few years. Numbers provided by NVidia.

A GPU is an example of the single-instruction, multiple data (SIMD) paradigm, which, unlike CPUs that are designed for rapid sequential code execution using a single thread, has a parallel architecture that executes many concurrent threads relatively slowly. Therefore, GPUs are well suited for high-performance computation with dense levels of data parallelism where the threads are data-independent from each other. CUDA is currently the most mature and widespread GPU computing platform for scientific applications. It provides developers direct access to parallel computational elements (GPUs) and enables code to run concurrently in CPUs. The assumption is that most numerically intensive components of a program will be executed in the GPU hardware with the remaining steps carried out in the CPU. The challenge in using GPUs lies in adapting the specialized hardware to take advantage of the expected performance increase. Moreover, memory allocation should be carefully handled to avoid memory latency issues. Single-precision should be carefully employed because its accuracy may be insufficient to handle the task at hand, and generally, single-precision is about 2-8 times faster than double-precision (depending on the device), so double-precision code will give relatively poor performance. Another important consideration is the fact that most existing computational chemistry software is written in Fortran, and to create GPU code (particularly C/C++ code, but there are commercial Fortran or other programming language CUDA compilers available), requires the creation of new software or incremental inclusion of GPU kernels into the code base.

GPU programming, especially the CUDA parallel programming model is designed to overcome the difficulty of developing application software that transparently scales its parallelism in order to leverage the increasing number of processor cores. The core of CUDA involves three key abstractions, a hierarchy of thread groups, memory types and barrier synchronization that are exposed to the developer as a minimal set of C language extensions. With these core abstractions, a CUDA developer can partition a problem into several sub-problems that can be solved independently in parallel with a thread in a block, and the overall problem can be solved cooperatively, by all threads, in all blocks, once the sub-problems have each been solved. Each block of threads, which is a logic concept, can be scheduled on any of the available multiprocessor within a GPU providing automatic scalability and only the runtime system needs to be aware of the running multiprocessor as Figure 1.5 illustrates. This automatic scalability scheduler allows the GPU architecture to span a range of markets by simply scaling the number of multiprocessors, from high-performance gaming platforms such as the GeForce GPU card, the professional level Quadro and high-performance computing platform Tesla.

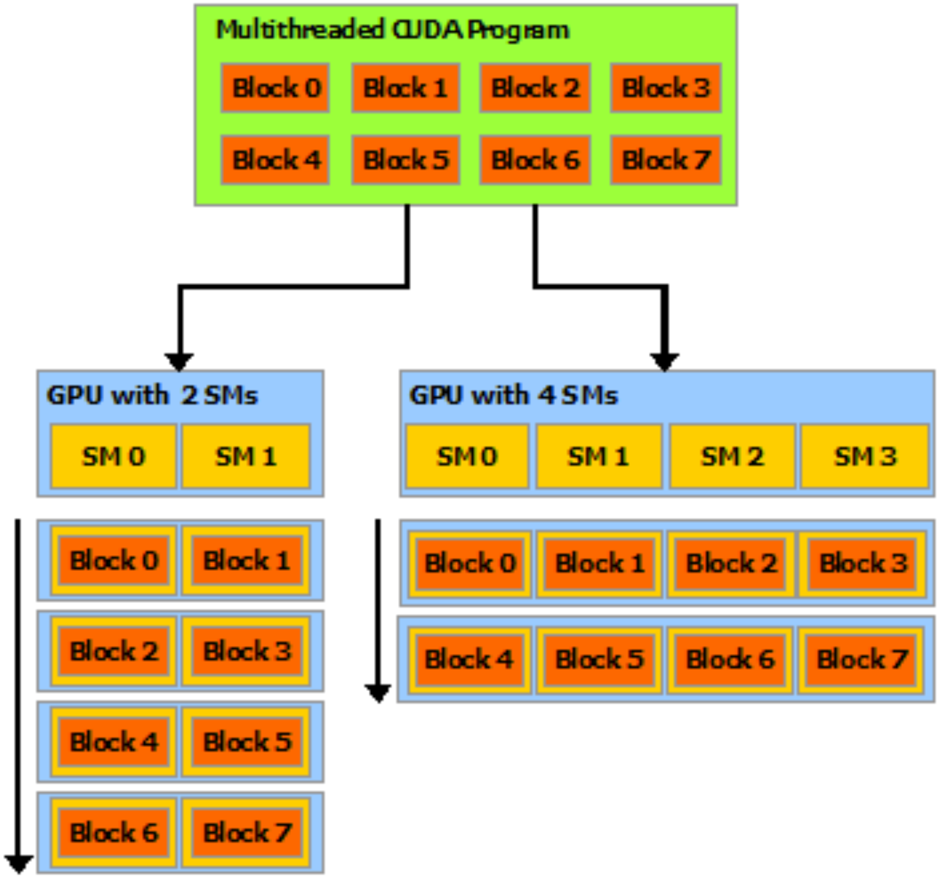


**Figure 1.5.** Illustration of the automatic scalability of CUDA. A block, a logical unit, can be executed in any available multiprocessor, which is a physical concept, by the CUDA scheduler.

For computational chemistry, Electron Repulsion Integral (ERI) calculations for ab initio theory along with with basic linear algebra operations typically define the computational cost of the application. ERI calculations formally scale as $N^4$ for the HF method where N is the number of basis functions, and typically it is reduced to $N^{<3}$ with careful pre-screening, cutoffs and efficient computational schemes. So for most calculations with N less than several thousand basis functions, ERI evaluation is the most time-consuming part while linear algebra, with a cubic scaling, will eventually dominate with very large number of basis functions. However, ERI evaluations are not as general as linear algebraic manipulations, and are less optimized than basic linear algebra subroutines (BLAS). In addition, unlike linear algebra and some other common subroutines with predictable and well-defined memory access patterns, ERI evaluations have many different types of integral classes and thread divergence could jeopardize computing efficiency. In addition, ERI evaluations require large amounts of both registers and memory. In GPU computation, slow memory access and shortages in register space are two factors that affect computational speed and need be considered carefully during software development. Hence, careful software design and special procedures are necessary to accelerate ERI evaluation using a GPU.

For the thread hierarchy, as illustrated in Figure 1.6, the most basic unit is the thread and threads are identified by the build-in variable *threadIdx*. This variable is a three-component vector, so that threads can be identified using one-two or three-dimensional indices forming one-two or three-dimensional blocks depending on the specifications of the developer's. The index scheme is especially suitable for the elements in a domain such as a vector or matrix. There is a limit to the number of threads per block since all threads of a block are expected to reside on the same multiprocessor and share limited memory. Blocks are further organized into a one-, two or three-dimensional grids of threads blocks. The number of thread blocks in a grid is usually decided upon by the size of the data being processed or the number of processors. Similar to the thread variable, *blockIdx* is also provided to identify a block in a thread. It is worth noting that the thread, block and grid are all logical concepts that are straightforward to map to physical concepts such as a multiprocessor in GPU to maximize performance.

In terms of the memory hierarchy, CUDA threads may access data from multiple memory spaces during execution as shown in Figure 1.7. First, each thread has fast and readable/writable local memory. Each thread block has shared memory that is visible to all threads within this block, which is also readable/writable with medium speed. Moreover, all threads have access to large, readable/writable but comparatively slow global memory. Two additional read-only memory spaces that are accessible by all threads (which are not displayed in Figure 1.7) are constant and texture memory. Texture memory also offers different addressing modes, as well as data filtering, for some specific data formats. The global, constant, and texture memory are persistent across kernel launches for the same application.

GPU programming, unlike CPU parallel programing, has several basic principals and tricks to maximize performance that is unique because of the architecture of the GPU. For example, in CUDA programming, the best performance is obtained when the number of threads in one block is a multiple of 32. Each 32 threads are bundled as a warp, and threads in a warp execute the same instructions, leading to the possible execution of the code differently than when executed sequentially - a phenomenon termed thread divergence. This divergence results in performance degradation and should be avoided in order to maximize parallel performance. Moreover, coalescing memory access ensures consecutive threads access consecutive memory addresses so that memory requests can be handled simultaneously and multiple active threads can hide memory latency by overlapping their computations. We will discuss these tuning procedures and show how to take full advantage of GPU architecture later on in this document.

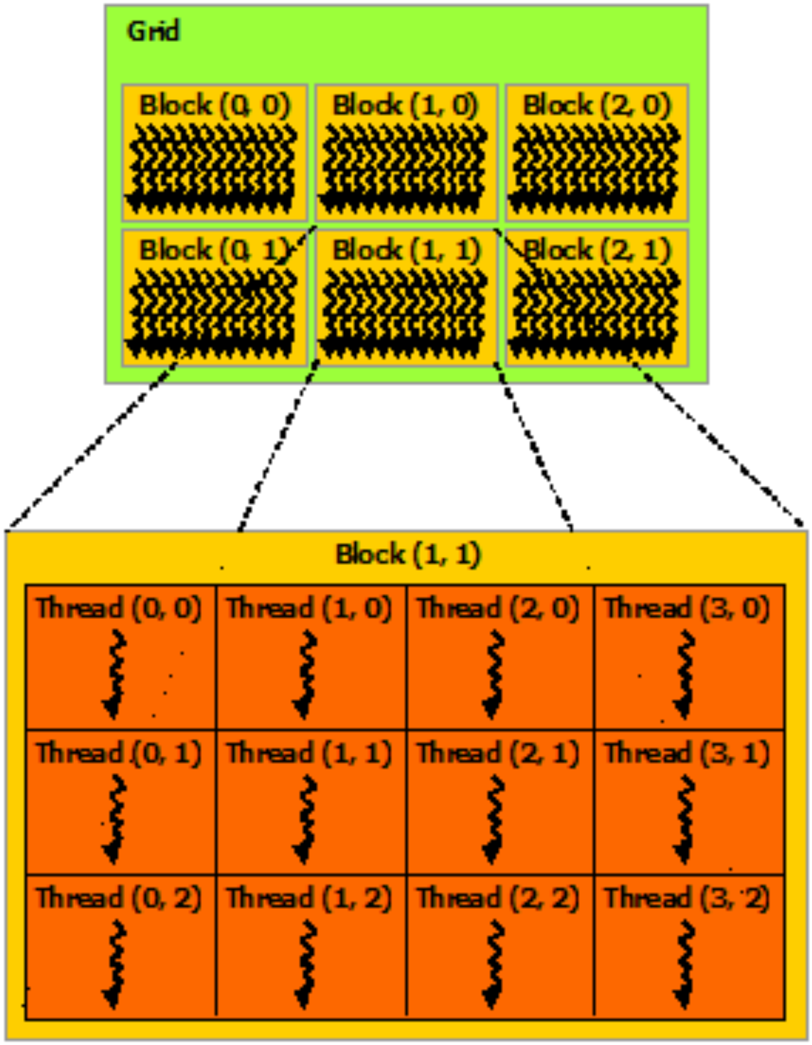


**Figure 1.6.** Illustration of the CUDA thread hierarchies. One, two or three-dimensional threads bundled as a block while one, two or three-dimensional blocks form a grid. Two-dimensional threads and two-dimensional blocks are presented.

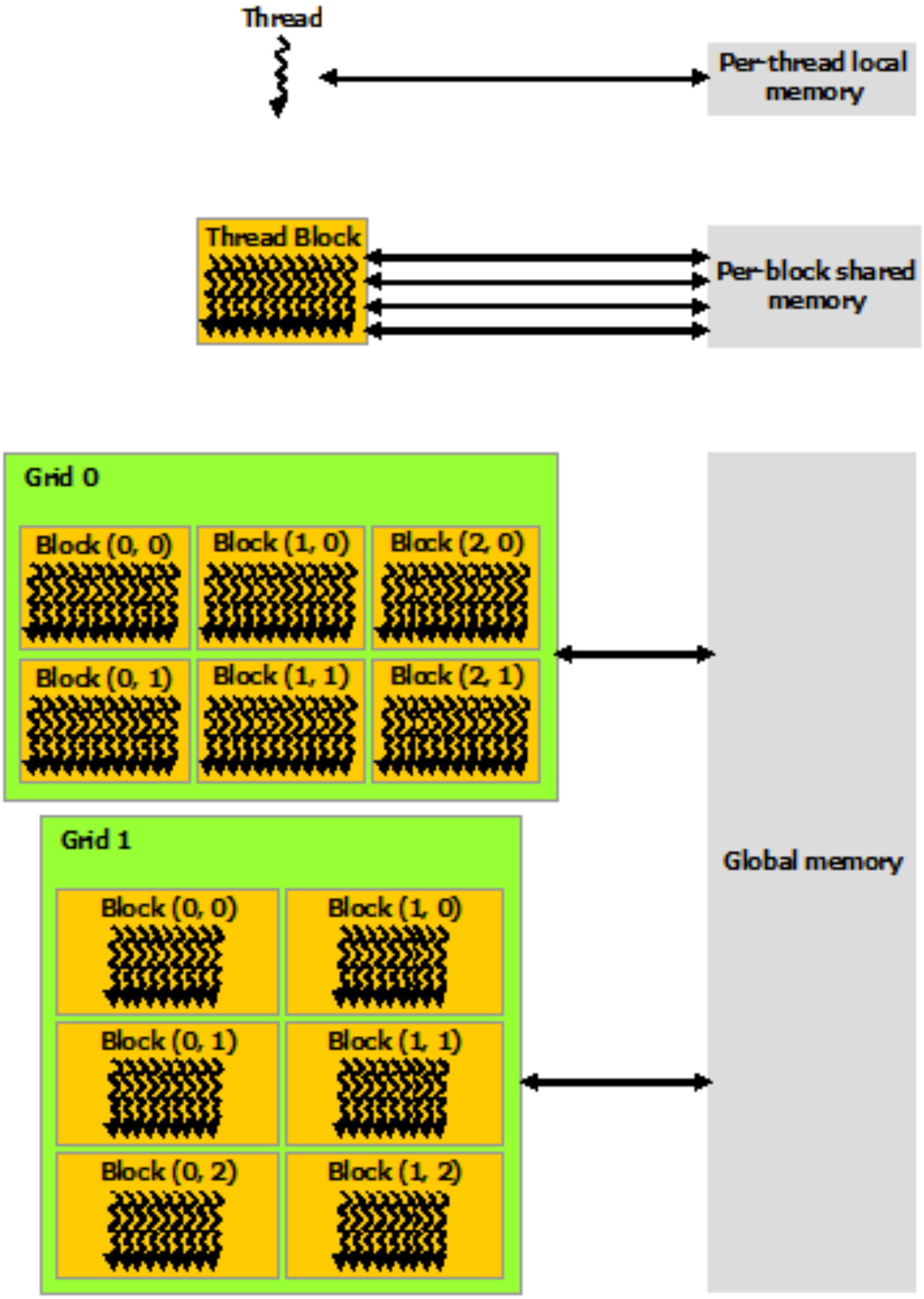


**Figure 1.7.** CUDA Memory hierarchies. Local memory, shared memory and global memory are presented. While the texture and constant memory are not presented but are described in the text.

## 1.2. GPU APPLICATIONS IN QUANTUM CHEMISTRY

Quantum Chemistry and solid-state physics code implement relatively complex algorithms[13]. The challenge in using GPUs and other parallel platforms lies in adapting these complex algorithms to take advantage of specialized hardware. For example, CUDA provides a swift way to utilize the power of GPUs by incorporating a CUDA library such as CUBLAS[3] or CUFFT[2] even without coding directly in CUDA, however, to maximize the performance, a careful consideration of memory hierarchy, thoughtful design of parallelism and maximization of memory access in order to minimize memory access latency[14] should be taken. For example, when using single-precision GPUs, numerical accuracy is a central issue because it is usually insufficient to match the accuracy of the underlying theoretical model. For example, using single-precision computation in *ab initio* calculation has as much as a 1kcal mol$^{-1}$ error[9], however, for some other applications, such as GPU-WHAM, described in chapter 5, an error of 0.2 kcal mol$^{-1}$ is realized using single-precision but the resultant code is 2-8 times faster than double-precision depending on the GPU device chosen. It is important to realize that many of these considerations are not only important for GPU programming, but for other platforms, including standard CPU clusters. Thus, many of the techniques employed to improve the parallel efficiency of quantum chemistry codes are also useful in GPU programming.

Hartree-Fock (HF) derived wavefunctions is the starting point for *ab initio* electron correlation method while Kohn-Sham Density Functional Theory(KS-DFT)[15,16] is usually used to calculate electronic ground states and their properties in chemistry due to its outstanding balance between accuracy and computing time. However, both KS-DFT and HF has two bottlenecks: 1) evaluation of the HF or DFT matrix elements, 2) solving the SCF equations. The latter relies on diagonalization of the Fock or KS matrix, which eventually dominates the computational cost because of its high order scaling ($N^3$), but GPU application to addressing this bottleneck has not has been touched extensively. It potentially could be solved via alternative electronic structure methods[17] or linear algebra librarys[18]. The former bottleneck is the major focus for quantum chemistry GPU software developers, which is dominated by the evaluation of the two-electron repulsion integrals (ERIs). ERIs are required by HF and the exchange-correlation (XC) contribution of DFT. Much work has been reported on the acceleration of ERI evaluation and these efforts are summarized in Table 1.1, including our own work.

**Table 1.1 Summary of Capability and Performance of GPU-based HF and DFT implementation**

| Authors | Software | $l_{max}$ | ERIs | J | K | XC | Gradient | Speedup[a] |
|---|---|---|---|---|---|---|---|---|
| Ufimtsev and Martinez[9-11] | TeraChem | d | Yes | Yes | Yes | Yes | Yes | 100-1000 |
| Yasuda[8] | | p | Yes | Yes | No | No | No | 10 |
| Asadchev et al.[19.31] | GAMESS | g | Yes | Yes | Yes | Yes | No | 17.5 |
| Miao and Merz | QUICK | f | Yes | Yes | Yes | Yes | Yes[b] | 10-100 |

a. Compared to single core CPU. b. Only supports up to d orbital

Yasuda[8] is the pioneer in porting ERI calculation to GPUs in 2008. In his work, he addressed the major problem for ERI evaluation on GPUs and presented the results for the Coulomb contribution to the KS matrix and the HF matrix with s and p type basis functions. Even tough the algorithm is not the most efficient due the Rys quadrature scheme[20] he chose with low memory requirement, his program maximized the load balance of GPU's SMs. At that moment in time, GPU's only supported single precision, so a mixed-precision combined with CPUs and GPUs was introduced so that large ERIs, prescreened by the Schwarz cutoff up bound, were calculated in the CPU with double-precision while, relatively small ERIs were evaluated on the GPU in single precision. This algorithm resulted in accurate DFT and SCF energies with errors within $10^{-3}$ kcal mol $^{-1}$ while full-GPU calculation produced an error of 1 kcal mol -1. The contribution to the Coulomb matrix was directly computed from uncontracted ERIs to avoid one of the major bottlenecks of GPU computing, namely, data-transfer between CPU and GPU. If all ERIs are taken by the GPU, the resultant speedup using NVIDIA's Geforce 8800 GTX is around one order of magnitude for the formation of the Coulomb matrix for a system of over one hundred atoms (such as valinomycin, 168 atoms) with the 6-31G basis set when compared to a single core CPU. However, if mixed-precision was employed, the speedup drops to three-fold. Yasuda's method only supported the Coulomb matrix and only supported low-angular momentum functions, and precision issues due to the hardware available at the time limited what he could do. However, he opened the door for researchers to explore the potential of GPUs for acceleration of ERI calculation.

Ufimtsev and Martinez[9-11] later on published a series of papers developing a CUDA kernel for ERI evaluation and Fock matrix assembly. They wrapped and commercialized their code into a software package called TeraChem, and GPU computing is the biggest selling-point of this program. Their early paper supported s and p type basis functions and gradients, and later on support for d type and its gradient was reported[29]. Both HF and KS-DFT methods are able to execute on a GPU, and was coupled with AMBER for *ab initio* QM/MM calculations for small systems. They implement the McMurchie-Davidson[21] scheme for its relatively small footprint throughout ERI evaluation, resulting in low memory requirements, which is similar to the Rys quadrature scheme. Three different mapping were tested and the results showed that the optimal strategy is to calculate each primitive ERI batch on one thread. In order to maximize the load balance and reduce thread divergence, a pre-sorting strategy was introduced, which treated the $N^4$ integrals as an $N^2$ * $N^2$ matrix which was sorted along the $N^2$ dimension using different criteria. As in Yasuda's work, the Fock matrix elements are directly computed on the GPU.

HF calculation with the 3-21G and 6-31G basis sets running on a NVIDIA GTX 280 card realized more than a 100-fold speedup compared with the quantum chemistry program package GAMESS. For small and medium size molecules, the Fock matrix time is the most time consuming part but for large molecules, such as Olestra (453 atoms), linear algebra required for the SCF solution emerged as a bottleneck since the Fock matrix formation time on the GPU is close to that of the linear algebra part. This algorithm places the s and p type functions in small integral blocks that can be treated entirely in shared memory to achieve high performance, however, this situation changed for basis functions with higher angular momentum quantum numbers, such as d-type functions. Moreover, the Rys quadrature used by GAMESS in their comparisons is not an optimal ERI evaluation algorithm and much more efficient algorithms do exist and less favorable GPU speedup should be expected. Moreover, the speedup observed is mostly based on the comparison between single-precision GPU and double-precision CPU results, so the errors in the SCF energies quickly exceeded $10^{-3}$ au for mid-size molecule. However, for larger molecules computation for larger ERIs in DP will be required. Similar to Yasuda's procedure, large ERIs in DP will be computed in double-precision, while smaller ones were carried out in SP. This mixed precision[22] model is a compromise between computation speed and accuracy by providing 2-4-fold speedup over full double-precision computing but reducing the error to about $10^{-6}$ au on Tesla C1060 and C2050 cards.

TeraChem also implemented the calculation of the analytical HF energy gradient with s- and p type basis function[9-11], and later on supported d type basis function[29]. Using the 3-21G basis set, GPUs achieved a speedup of 6-100 for small to larger molecules on NVIDIA 295 cards. Using the mixed precision model described above, the RMS (root mean square) error in the forces is about $10^{-5}$ au, which is close to the typical convergence threshold for geometry optimizations. With the aid of energy and gradient calculations on GPUs, small *ab inito* molecule dynamics simulations was feasible. For example, a MD simulation of the $H_3O^+(H_2O)_{30}$ Cluster with the 6-31G basis set in the microcanonical ensemble with a time step of 0.5 fs was realized. An energy drift of 0.022

kcal mol$^{-1}$ ps$^{-1}$ was observed over a simulation time of 20 ps. It is worth mentioning that TeraChem can run over multiple GPUs with a parallel efficiency of over 60% on three NVIDIA GeFrorce 8800GTX cards.

To support d orbitals for both the energy and gradient[29], as mentioned above, a meta-programming strategy that leverages the computer algebra system to generate correct and efficient code was employed. Since the capability of a compiler to identify the best code transformation and permutations of data accesses is limited by the need to carry out source code to machine code translation efficiently, full code optimization needs to be dealt with at the source code level. Therefore, their code-generator, translated both the mathematical formalism of Coulomb and exchange integral computation to modern programming language such as Fortran and C, but also simplified the algebraic expressions by factoring out and eliminating common sub-expressions and by divying expressions into groups of intermediates. By using meta-programming, not only were d-type basis sets supported, but s- and p- type basis sets were re-written and further optimized. Moreover, the resultant CPU code also tokk advantage of meta-programming optimization. Similar to previous publications, the realized speedup on GPUs involving d-type basis functions was a factor of 10-100 using a NVIDIA GTX 580 card with an error of $10^{-3}$-$10^{-2}$ kcal mol$^{-1}$ when compared to the corresponding CPU calculations.

Asadchev *et al*. presented algorithms and CUDA implementations for uncontracted ERI evaluation up to g-type functions based on the GAMESS package[19,31]. The Rys quadrature was chosen for its low memory requirement and efficiency for higher order angular momentum. Rys quadrature has complex memory access patterns especially for high angular momentum. (ff|ff) for example, requires 5376 FLOPS for intermediate quantities which are reused and $10^4$ FLOPS for the final ERIs. So with DP, the memory requirement is larger than the device memory so that slow global memory access is mandatory. Therefore, the author re-arranged the parallel calculation of the ERIs to minimize memory loads. The code also processes ERIs, such as the Fock Matrix formation, on the GPU device to avoid CPU-GPU communication. Moreover, similar to TeraChem, the complex code must be machine generated to ensure correctness and efficiency. The authors therefor selected Python Cheetah[30] to adopt a template-based approach to generate automated code. As was seen for the TeraChem implementation, machine-generated code also boosts the performance of the CPU code.

The improved CPU and GPU code was tested on NVIDIA GTX 275's and Tesla T10's and was then compared with the performance of ERI evaluation with the original Rys quadrature implemented in GAMESS. More than a 30% improvement was gained using the new C++ Rys quadrature code on the CPU, even though some of test cases showed other ERI evaluation algorithms in GAMESS outperformed this approach. The GPU implementation, on the other hand, observed a speed-up of up to 17.5-fold when compared with the new C++ Rys quadrature code on a CPU with double-precision. Their code not only can execute in parallel on multiple-CPUs, but also on a multi-CPU-single-GPU hybrid platform.

# REFERENCES

# REFERENCES

(1) NVIDIA. Compute Unified Device Architecture (CUDA). http://docs.nvidia.com/cuda/index.html (accessed July, 2014)

(2) NVIDIA. The NVIDIA CUDA Fast Fourier Transform library. http://developer.nvidia.com/cufft (accessed July, 2014)

(3) NVIDIA. The NVIDIA CUDA Basic Linear Algebra Subroutines (cuBLAS) library. http://developer.nvidia.com/cublas (accessed July, 2014)

(4) Götz, A. W.; Wölfle, T.; Walker, R. C. In *Annual Reports in Computational Chemistry*; Ralph, A. W., Ed. 2010; Vol. 6, p 21-35.

(5) D.A. Case; T.A. Darden; T.E. Cheatham, I.; C.L. Simmerling; J. Wang; R.E. Duke; R. Luo; R.C. Walker; W. Zhang; K.M. Merz; B.P. Roberts; B. Wang; S. Hayik; A. Roitberg; G. Seabra; I. Kolossváry; K.F. Wong; F. Paesani; J. Vanicek; J. Liu; X. Wu; S.R. Brozell; T. Steinbrecher; H. Gohlke; Q. Cai; X. Ye; J. Wang; M.-J. Hsieh; G. Cui; D.R. Roe; D.H. Mathews; M.G. Seetin; C. Sagui; V. Babin; T. Luchko; S. Gusarov; A. Kovalenko; Kollman, P. A. *AMBER11,University of California, San Francisco*. **2010**.

(6) Case, D. A.; Cheatham, T. E.; Darden, T.; Gohlke, H.; Luo, R.; Merz, K. M.; Onufriev, A.; Simmerling, C.; Wang, B.; Woods, R. J. *J. Comput. Chem.* **2005**, *26*, 1668-1688.

(7) Anderson, J. A.; Lorenz, C. D.; Travesset, A. *J. Comput. Phys.* **2008**, *227*, 5342-5359.

(8) Yasuda, K. *J. Comput. Chem.* **2008**, *29*, 334-342.

(9) Ufimtsev, I. S.; Martinez, T. J. *J. Chem. Theory Comput.* **2008**, *4*, 222-231.

(10) Ufimtsev, I. S.; Martinez, T. J. *J. Chem. Theory Comput.* **2009**, *5*, 1004-1015.

(11) Ufimtsev, I. S.; Martinez, T. J. *J. Chem. Theory Comput.* **2009**, *5*, 2619-2628.

(12) Vogt, L.; Olivares-Amaya, R.; Kermes, S.; Shao, Y.; Amador-Bedolla, C.; Aspuru-Guzik, A. *J. Phys. Chem. A* **2008**, *112*, 2049-2057.

(13) Helgaker, T., Jørgensen, P., Olsen, J. Molecular Electronic-Structure Theory, Wiley, West Sussex, England, **2000**.

(14) Kirk, D.B., Hwu, W.W. Programming Massively Parallel Processors, Morgan Kaufmann Publish- ers, Burlington, MA, **2010**.

(15) Kohn, W., Sham, L. *Phys. Rev.* **1965**, *140*, A1133-8.

(16) Parr, R.G., Yang, W. Density-Functional Theory of Atoms and Molecules, Oxford University Press, Oxford, **1989**.

(17) Salek, P., Hos, S., Thogersen, L., Jorgensen, P., Manninen, P., Olsen, J., Jansik, B. *J. Chem. Phys.* **2007**, *126*, 114110.

(18) NVIDIA. The NVIDIA CUDA Sparse Matrix library (cuSPARSE) library. https://developer.nvidia.com/cuSPARSE (accessed July, 2014)

(19) Asadchev, A.; Allada, V.; Felder, J.; Bode, B. M.; Gordon, M. S.; Windus, T. L. *J. Chem. Theory Comput.* **2010**, *6*, 696-704.

(20) Rys, J.; Dupuis, M.; King, H. F. *J. Comput. Chem.* **1983**, *4*, 154-157.

(21) McMurchie, L. E.; Davidson, E. R. *J. Comput. Phys.* **1978**, *26*, 218-231.

(22) Luehr, N.; Ufimtsev, I. S.; Martinez, T. J. *J. Chem. Theory Comput.* **2011**, *7*, 949-954.

(23) Kulik, H. J.; Martinez, T. J. *Abstr Pap Am Chem S* 2012, *244*.

(24) Kulik, H. J.; Luehr, N.; Ufimtsev, I. S.; Martinez, T. J. *J Phys Chem B* 2012, *116*, 12501-12509.

(25) Isborn, C. M.; Gotz, A. W.; Clark, M. A.; Walker, R. C.; Martinez, T. J. *J. Chem. Theory Comput.* 2012, *8*, 5092-5106.

(26) Ufimtsev, I. S.; Luehr, N.; Titov, A.; Martinez, T. *Abstr Pap Am Chem S* 2011, *242*.

(27) Ufimtsev, I. S.; Luehr, N.; Martinez, T. J. *J Phys Chem Lett* 2011, *2*, 1789-1793.

(28) Kulik, H. J.; Isborn, C. M.; Luehr, N.; Ufimtsev, I.; Martinez, T. J. *Abstr Pap Am Chem S* 2011, *242*.

(29) Titov, A. V.; Ufimtsev, I. S.; Luehr, N.; Martinez, T. J. *J. Chem. Theory Comput.* 2013, *9*, 213-221.

(30) Cheetah - the Python-Powered template engine. http://www. cheetahtemplate.org/ (accessed Jan, 2015).

(31) Asadchev, A.; Gordon, M. S. *J. Chem. Theory Comput.* 2012, *8*, 4166-4176.

# CHAPTER 2. PROJECT OBJECTIVES

The main objectives of this work were:

A. Develop a novel algorithm to execute *ab initio* quantum chemistry methods including energy calculation using the Hartree-Fock method and Density Functional Theory on GPUs using the CUDA platform. As described above, the peak FLOPs count and bandwidth of GPUs are about 10x of that of a CPU, so we expected significant speedup could be realized. For most of these calculations, s, p and d orbitals are sufficient for second row elements that are essential in organic chemistry and biochemistry; hence, the support of s, p and d orbital calculations was initially focused on.

B. Develop novel algorithms to execute the gradient calculation of *ab initio* quantum chemistry methods including HF and DFT methods. Support for s, p and d orbitals is the target, but to support the d orbital gradient, we needed f orbital energies as well. Hence, we also aimed to support f orbital energy calculations as well.

C. Integrate the above-mentioned main features into our quantum chemistry software package, Quick, and further GPU-ize the software by introducing CUDA library calls into linear algebra calculations to further speedup HF and DFT calculations.

D. Benchmark the software for a series of representative molecules and calculations including energy, gradient, and geometry optimization calculations on GPUs and CPUs and then compare and discuss the result.

E. Develop GPU-based WHAM (The Weighted Histogram Analysis Method), a Molecular Dynamics trajectory analysis software. The one-dimensional WHAM is very fast, but multi-dimensional WHAM is time-consuming for some real tasks in our group, therefore, a GPU-ized WHAM accelerated our research.

# CHAPTER 3. GPU ACCELERATION ON ERI EVALUATION

## 3.1. INTRODUCTION

Quantum theory has been utilized in many roles, including interpreting chemical phenomena and predicting new molecules with novel functions. To achieve this, computational and theoretical chemists constantly make compromises between accuracy and computational expense. However, we are witnessing a new era in computational quantum chemistry, sparked by an interest in harnessing the capabilities of heterogeneous computing, especially modern graphics processing units (GPUs), which afford impressive price *versus* performance characteristics. Early GPUs were not widely accepted by the computational chemistry community because of limited precision and programming difficulties. Recently, though, these difficulties have been largely remedied by the development of the latest generation of GPU cards from NVIDIA, which support up to 64-bit floating-point arithmetic, and through the introduction of the Compute Unified Device Architecture (CUDA)[1], which is a simple interface extension based on the standard C/C++ language.

So in this chapter, we will focus on the acceleration on ERI evaluation using GPU so that the calculation of HF energy can be speedup. This chapter is organized as following. First, we briefly describe the ERI evaluation algorithm employed, which uses vertical and horizontal recurrence relations and is one of the most efficient methods for ERI evaluation. Next we describe a general approach for GPU evaluation of ERIs using recurrence relations, and describe the details regarding Fock matrix assembly in our direct SCF implementation. To study the efficiency of our direct SCF scheme, we compare it to a conventional SCF implementation on a GPU. In the next section, we provide more detailed benchmarks in order to compare our GPU and CPU implementations. With double-precision ERI evaluation, the accuracy of the SCF calculation is $10^{-7}$ a.u. or better for moderate systems with thousands of basis functions, suitable for most applications. We have applied these ideas in a series of typical proteins (up to 4000 basis functions) with high-angular momentum Gaussian basis sets and show performance increases of up to 100 fold, but more typical speedups are approximately 10 to 20 fold. Memory usage is shown in this section, where we show that systems with up to ~10000 basis function are feasible with a typical GPU. Finally, we conclude the chapter with a brief discussion and conclusions.

## 3.2. ELECTRON REPULSION INTEGRALS

In computational chemistry field, the Hatree-Fock(HF) method is one of the most popular *ab initio* quantum chemistry method, which is based on approximation for the determination of wave functions and energy in a stationary state. By assuming N-Body wave function of a system can be approximated expressed by a single Slater determinant, HF method invokes the variational method that minimizes N-body HF energy to be close to real energy. A solution yielding HF wave function and energy, can be derived from a series of equation called HF equation as an approximate solution of Schrödinger equations and the solution of HF equation required to be a self-consistent mean charge field. Most HF calculation can be solved iteratively although iteration algorithm does not guarantee to converge and advanced method can remedy the issue that is not in the scope of this chapter's discussion.

On the other hand, Density functional theory (DFT) is a computational quantum mechanical modeling method that also based on an approximation of Schrödinger equation for N-body. With this theory, the properties of N-body system are determined and described by the spatially electron density of the N-body using functional. For most modern DFT method, iteratively solution is necessary based on Kohn-Sham equation in a self-consistent fashion.

For *ab initio* including HF and DFT methods, one of the major time-consuming steps is the evaluation of a large number of two-electron repulsion integrals that have the form:

$$(\mu\nu|\kappa\lambda) = \iint \varphi_\mu(\vec{r})\varphi_\nu(\vec{r})\frac{1}{r-r'}\varphi_\kappa(\vec{r'})\varphi_\lambda(\vec{r'})d\vec{r}d\vec{r'} \qquad (1)$$

where the $\varphi_\mu$ are one-electron basis functions and **r** refers to the coordinates of the electron. In practice, linearly combined contracted Gaussian functions made up of primitive atom-centered Cartesian Gaussian functions of the form:

$$\varphi_\mu(\vec{r}) = \sum_{p=1}^{N} c_{\mu p}\chi_p(\vec{r}) \qquad (2)$$

are used to represent the one-electron basis functions. The contribution of the primitive p to the contracted function is denoted by the coefficient $c_{\mu p}$. An unnormalized primitive Cartesian Gaussian function centered at $\vec{A} = (A_x + A_y + A_z)$ with exponent $\alpha$ is given by

$$\chi_p(r) = (x - A_x)^{a_x}(y - A_y)^{a_y}(z - A_z)^{a_z}e^{-\alpha(\vec{r}-\vec{A})^2} \qquad (3)$$

where

$$\vec{a} = (a_x, a_y, a_z) \qquad (4)$$

and p is a set of integers indicating angular momentum and the direction of the Gaussian function. Therefore, the contracted two-electron integral is constructed from integrals over primitive functions and coefficients. A primitive ERI will be denoted by [pq|rs], and a contracted ERI will be distinguished from a primitive ERI by:

$$(\mu\nu|\kappa\lambda) = \sum_{pqrs} c_{\mu a} c_{\nu b} c_{\kappa c} c_{\lambda d} [ab|cd] \qquad (5)$$

ERIs possess eight-fold symmetry ($(\mu\nu|\kappa\lambda) = (\nu\mu|\kappa\lambda) = (\mu\nu|\lambda\kappa)$, *etc*.), meaning one can be computed and used for the remainder, providing a reduction in computational effort by roughly 6-fold. It would be more efficient to compute all of the primitive ERIs involving four shells for which $\vec{a}$ has the same $a_i$. For example, 3 [ps|ss] type integrals can be computed at the same time where $\vec{p} = (1,0,0), (0,1,0)$ and $(0,0,1)$. The ERI bottleneck formally has $N^4$ scaling, but many approaches have been devised to reduce scaling to $N^{<3}$, however, even with these algorithms, ERI computation can still consume the majority of the computational time (typically 80-90% of the overall CPU time for systems with less than a thousand basis functions). Once contracted ERIs are generated as given in equation (1), it is straightforward to form the Fock matrix with

$$F_{\mu\nu} = H_{\mu\nu}^{core} + \sum_{\lambda\sigma} P_{\lambda\sigma}[(\mu\nu|\sigma\lambda) - \frac{1}{2}(\mu\lambda|\sigma\nu)] = H_{\mu\nu}^{core} + J_{\mu\nu} - \frac{1}{2}K_{\mu\nu} \qquad (6)$$

where P is density matrix and J and K are the Coulomb and exchange matrices respectively.

### 3.2.1. RECURRENCE RELATIONS FOR ERI EVALUATION

The primitive integrals can be evaluated in many ways and most of them are based on Boys' seminal work[13]. Recurrence relation is one of the most efficient and most widely used methods. Obara and Saika(OS)[14] first derived a recurrence relation for ERI evaluation which related a given ERI to other integrals. Head-Gordon and Pople (HGP)[15] optimized this method by storing some common ERIs to reduce floating point operation counts. In their derivation, the vertical recurrence relation (VRR) is given as:

$$\begin{aligned}[(a+1_i)b|cd]^{(m)} &= (P_i - A_i)[ab|cd]^{(m)} + (W_i - P_i)[ab|cd]^{(m+1)} \\ &+ \frac{a_i}{2\zeta}\left([(a-1_i)b|cd]^{(m)} - \frac{\eta}{\eta+\zeta}[(a-1_i)b|cd]^{(m+1)}\right) \\ &+ \frac{b_i}{2\zeta}\left([a(b-1_i)|cd]^{(m)} - \frac{\eta}{\eta+\zeta}[a(b-1_i)|cd]^{(m+1)}\right) \\ &+ \frac{c_i}{2(\eta+\zeta)}[ab|(c-1_i)d]^{(m+1)} + \frac{d_i}{2(\eta+\zeta)}[ab|c(d-1_i)]^{(m+1)}\end{aligned} \tag{7}$$

where i is x, y or z, and

$$1_i = (\delta_{ix}, \delta_{iy}, \delta_{iz}), \tag{8}$$

$$\zeta = \alpha + \beta, \eta = \gamma + \delta, \tag{9}$$

and $\alpha, \beta, \gamma, \delta$ are exponents of a, b, c and d respectively.

$$P_i = \frac{\alpha A_i + \beta B_i}{\alpha+\beta}, Q_i = \frac{\gamma C_i + \delta D_i}{\gamma+\delta}, W_i = \frac{\zeta P_i + \eta Q_i}{\zeta+\eta}. \tag{10}$$

The superscript index (m) in eq (7) is an auxiliary index, with m=0 yielding true ERIs. The VRR shows that primitive ERIs of higher angular momentum are linear combinations of lower angular momentum ERIs ultimately depending on s type ERIs. The quantities requiring evaluation are $[ss|ss]^{(0)}$ to $[ss|ss]^{(n)}$, where the max n value is:

$$n = \sum_{i=1}^{3}(a_i + b_i + c_i + d_i) \tag{11}$$

We can evaluate $[ss|ss]^{(m)}$ with analytical formality:

$$[ss|ss]^{(m)} = \frac{1}{\sqrt{\eta+\zeta}} K_{AB} K_{CD} F_m(T) \tag{12}$$

where

$$T = \frac{\zeta\eta}{\zeta+\eta}(\vec{P} - \vec{Q})^2 \qquad (13)$$

$$F_m(T) = \int_0^1 t^{2m} e^{-Tt^2} dt \qquad (14)$$

$$K_{AB} = \sqrt{2}\frac{\pi^{\frac{5}{4}}}{\alpha+\beta}\exp\left[-\frac{\alpha\beta}{\alpha+\beta}\left(\vec{A} - \vec{B}\right)^2\right] \qquad (15)$$

and $K_{CD}$ is analogous to $K_{AB}$. HGP noted further that

$$[a(b + 1_i)|cd]^{(m)} = [(a + 1_i)b|cd]^{(m)} + (A_i - B_i)[ab|cd]^{(m)} \qquad (16)$$

This equation relates one integral with another of the same total angular momentum but with a shifted position from the first to the second, and is termed as a horizontal recurrence relation (HRR). Since this relation does not involve the exponent partial, it may be applied to contracted ERIs.

$$(a(b + 1_i)|cd)^{(m)} = ((a + 1_i)b|cd)^{(m)} + (A_i - B_i)(ab|cd)^{(m)} \qquad (17)$$

For example, using the previous equation to evaluate (ab|cd), integrals from (a0|b0) to ((a+b)0|(c+d)0) are constructed using VRR and then (ab|cd) are evaluated using HRR. This algorithm greatly reduces floating-point operation counts, especially for basis functions with high angular momentum. In practice, temporary integrals, with auxiliary index m = 0, are used in other ERI evaluations, and should be stored in memory temporarily for higher efficiency. For double zeta Gaussian basis sets, hybrid functions, sp for example, are used, and to treat this type of function, primitive ERIs such as [a0|b0] to [(a+b)0|(c+d)0] can be stored instead of contracted ERIs because s and p share the same primitive ERI exponent values, this treatment speeds up calculations, but requires large memory resources in exchange. Because of their different architectures, the required memory is attainable for CPUs but is too large to implement efficiently for GPUs. We will return to this issue in the coming sections.

## 3.3. IMPLEMENTATION

### 3.3.1. COMPUTE UNIFIED DEVICE ARCHITECTURE

GPU technology, typically used to handle computation for computer graphics and video gaming, has been adapted to perform computations for applications that are traditionally handled by CPUs. This is termed General-Purpose computing on Graphics Processing Units or GPGPU. GPU is a good example of a massive parallel stream-processing architecture that uses the single-instruction multiple data (SIMD) model. Currently, as we described above, the most widely used language environment for GPGPU technology is NVIDIA Compute Unified Device Architecture (CUDA). CUDA is a programming model for graphics as well as general-purpose computation using a relatively simple extension of the standard C language to develop scalable and efficient parallel programs.

The CUDA device architecture has a scalable array of streaming multiprocessors that consist of 8 scalar processors. Note that all technical specifications referred to in this paragraph are for CUDA 2.x unless otherwise noted. With this design, GPUs are especially well suited for compute-intensive and highly parallel computations.

As introduced in chapter 1, within the CUDA framework, a batch of threads is hierarchically arranged into a one-, two- or three-dimensional grid of blocks up to 65535 blocks, and each block of threads further consists of one-, two-, or three-dimensional grids (shown in 1.6). The number of threads in a block cannot exceed 512 and should be specified explicitly in the code. The CUDA framework introduces the built-in variables *threadIdx* and *blockIdx* to identify a thread in a block. The best performance is obtained when the number of threads in one block is a multiple of 32. Each 32 threads are bundled as a warp, so that thread in the same warp will execute the same instruction that may result divergence when occur branches. Divergence results in performance degradation, and should be avoided in order to maximize parallel performance, an issue that will be touched on later. The thread scheduler switches active warps to balance the load so that the overall performance will be maximized. Although only one thread block can be executed at any given time on an SM (two for the Fermi architecture), multiple thread blocks can be active. For the memory hierarchies (as shown in 1.7), each thread has access to its local register on the processor, and shared data in a block is visible to threads in this block via a parallel data cache with medium data latency but limited resources. Furthermore, all threads have access to the relatively large global memory (also known as dynamic random-access memory, DRAM) but with high data access latencies. Also, GPUs provide fast and high visibility but limited and read-only constant and texture memories. The global, constant, and texture memory spaces are persistent across kernel launches by the same application. Because of the architecture of the CUDA platform, main memory has high latency, on the order of hundreds of cycles. Therefore, in order to achieve high bandwidth, coalesced memory access to DRAM is recommended. Coalescing ensures consecutive threads access consecutive memory addresses so that memory requests can be served simultaneously.  Multiple active threads can hide memory latency by overlapping their computations.

When the GPU kernel function is called in the CUDA implementation, the CPU waits until the kernel function completes and is returned. Most GPU kernel functions only access GPU memory, therefore the CPU must copy the required data from the CPU memory to the GPU memory. Because of the relatively slow 2.0 Gb/s transfer speeds between the CPU and GPU memories, it is important to avoid large amounts of CPU-GPU data transfer.  We will discuss a scheme to reduce CPU-GPU data transfer below.

In addition, subroutine libraries are available that provide common solutions for problems in quantum chemistry and solid-state physics such as the Fourier transform (CUFFT)[16] and linear algebra (CUBLAS)[17] operations that are used in our implementation. One of the most notable accelerations these libraries afford is in matrix-multiplication, providing a six- to ten-fold speedup for large GEMMs over the INTEL MKL library.

### 3.3.2. RELATED WORK

As introduction chapter mentioned, Yasuda[6] was the first to evaluate ERIs on GPUs for s and p functions in single precision. He chose the Rys quadrature scheme[18] for its low memory requirements even though it is not the most efficient algorithm for low angular momentum quantum numbers. A mixed-precision (MP) scheme was introduced to calculate the largest ERIs in double precision (DP) on the CPU and the remainder in single-precision (SP) on the GPU. This scheme led to SCF energy errors of the order of

$10^{-3}$ au. If all ERIs are computed on the GPU, speedups of ~1 order of magnitude for molecules as big as valinomycin with 6-31G basis set (168 atom, 882 basis sets) have been realized. Asadchev[19,20] further adopted the Rys quadrature scheme on GPUs for ERI evaluation in GAMESS[21] and extended it up to g functions. Their implementation on NVIDIA Tesla T10 cards showed speedups of around 25 times for DP and 50 for SP compared with a single CPU; however, no timings were given for data transfer between the GPU DRAM and CPU main memory which may take several times longer than the actual execution time of the ERI kernel.

Ufimtsev and Martinez[7-9] have developed a CUDA-based program for ERI evaluation and Fock matrix formation involving s and p basis functions on GPUs, using the McMurchie-Davidson[22] scheme for its relatively few intermediate steps per integral and its memory requirements, similar to the Rys quadrature[18]. HF SCF calculations with 3-21G and 6-31G basis sets using their implementation and a NVIDIA GTX280 card[7-9] was, in some cases, more than 100 times faster than the quantum chemistry software GAMESS[21] on a CPU. However, most of their comparisons were between single-precision GPU result and double-precision CPU results so the SCF energy error observed with UM's code quickly exceeded $10^{-3}$ au. Later, a mixed SP/DP approach was described as a compromise between accuracy and speed[23].

### 3.3.3. IMPLEMENTATION DESIGN

In our work, in order to implement the ERI evaluation in the GPU, code was created using our code generator to express the recurrence relation given in eq (7). Current GPU code will use registers to store the auxiliary integrals and automatically delete and flag them as available once they are not needed in the subsequent steps, increasing register usage efficiency by keeping as many registers busy as possible.

As discussed above, GPU architecture is very different from that of CPU, and one of the most important concerns is thread divergence, that is to say, every thread in a warp will execute the same instruction sets and, therefore, threads may be idle when they do not go into the same branching or conditional statement as other threads in the warp. So it is necessary to classify ERI type and subsequently reorder them so that threads in one warp will likely undergo the same ERI or loop (or all of them bypass ERI evaluation because of the Schwarz upper bound cutoff[24,25]). We treat the ERIs in the direct SCF procedure as an $N^2*N^2$ integral matrix problem with row and column indices as half ERIs in the form of bra ( $[ab|$ ) and ket ( $|cd]$ ), where an ERI corresponds to an element in this matrix. $[ab|cd]$ evaluation can be skipped if index $a$ is greater than index c due to ERI symmetry. We find that by sorting bra and ket (including tiebreakers) a well-ordered integral grid can be obtained that leads to generally optimal performance for most systems. Figure 3.1 illustrates our approach using a water cluster with 4 water molecules employing the cc-pVDZ[26] basis set with 100 basis functions. We use coloration to denote the magnitude of the ERI estimation value. First, sorting in $|ab]$ half ERIs type, for instance, the highest angular momentum in this system with cc-pVDZ basis set is a d-orbital, so we have nine combinations as shown in figure 3.1. We identify the angular momentum criterion as the most important. Second, half ERIs are sorted by Primitive Gaussian function number. The cc-pVDZ basis set has a maximum of seven primitive

Gaussian functions for Oxygen used to construct the contracted basis set. Contracted ERIs of type (ss|ss), for example, require various numbers of primitive ERIs [ss|ss] from 2401 to 1 (from $7^4$ to $1^4$), so if it is not sorted, it is quite possible that one thread takes 2401 loops and others in the warp take significantly fewer loops and wait in an idle state. Therefore, sorting by primitive Gaussian function number in the same ERI type will greatly improve thread usage percentage. Correlation-consistent basis sets benefit most from this sorting criterion as our benchmark results show below. The third sorting criterion is the Schwarz cutoff, or upper bound. The size of an individual ERI can be estimated using Cauchy-Schwarz inequality[24,25],

$$[ab|cd] \leq \sqrt{[ab|ab][cd|cd]} \tag{18}$$

With this cutoff bound sorting, sparse populated regions are gathered to certain areas so that threads in a warp that are responsible for this area could possibly skip ERI evaluation simultaneously and set as available for future computation. These three sorting orders create a well-balanced grid as shown in Figure 3.1 and guarantee that threads are kept busy and at or near maximum usage.

Most streaming-type architectures, such as CUDA, do not provide efficient tools for inter-thread data communication except for threads that are in the same block. We find that the most efficient mapping strategy is to have one thread working on one contracted ERI. This mapping strategy is efficient based on our well-sorted scheduling combined with thread divergence avoidance in order to keep threads under dense computation. Variables in eqs. (9), (10) and (15) ( $\zeta = \alpha + \beta$ , $P_i = \frac{\alpha A_i + \beta B_i}{\alpha + \beta}$ and

$K_{AB} = \sqrt{2}\frac{\pi^{\frac{5}{4}}}{\alpha+\beta}\exp\,[-\frac{\alpha\beta}{\alpha+\beta}\left(\vec{A}-\vec{B}\right)^2]$ ) are all pre-calculated and stored in GPU memory. They will occupy most of the DRAM in the GPU with a usage scale of O($N^2$), trading memory usage with speed. With these pre-calculated values, both DRAM data access count and floating-pointing count is reduced significantly.

Similar to the CPU implementation, once a thread generates a series of integrals two strategies are available. One is to store integrals to disk so that they will not be re-calculated in further iterations, which is traditionally called a conventional SCF. However, the drawback of this strategy is the slow CPU-GPU data-transfer speed and its high scale. It has been proven for large systems with a huge number of integrals that data-transfer time is greater than GPU kernel time[7] and similar validation will be shown. Another algorithm is direct SCF, which uses calculated integrals immediately and recalculates those integrals on the fly. It is clearly desirable to implement the SCF entirely in the GPU. In our implementation, once an integral is available, it is used to assemble the Fock matrix. Traditionally, matrix-element-to-block mapping is adopted for Fock matrix formation, however, that implementation produces 2-8 times more redundant integrals, leading to low efficiency. In our code, the atomic function is introduced to assemble the Fock matrix. The atomic function is a feature offered by NVIDIA since CUDA 1.1, which performs a lock-and-set atomic operation on words residing in global or shared memory (since CUDA 1.2). These operations are named atomic in the sense that it is guaranteed to perform without interference from any other threads. After one thread locks a memory address, no other thread has access privilege until the operation is completed. This feature is used in our implementation: once a unique integral is evaluated in a thread, it will be added to the corresponding Fock matrix elements forming the Coulomb and exchange matrices simultaneously. A unique integral is usually required by Fock matrix elements multiple times, depending on ERI symmetry. For example, because of symmetry relations for ERIs (ij|kl)=(ji|kl)=(kl|ij)=etc., (ij|kl) can be reused 8 times if i, j ,k, l are not equal to each other.

One concern for the atomic function is it only supports long integer operations in the current version while the double-precision atomic operator is not available (due to hardware limitations). This concern arise because, for integral arithmetic, there is no stipulation about the order in which threads perform their operations. Hence, floating-point arithmetic is not associative for the rounding of intermediate results. For instance, (A+B)+C equals A+(B+C) if A, B, C are integers, but this is generally not true for floating-point. Therefore, early hardware does not support floating-point and the newest version only supports single-precision. A compromise approach is masking the double precision type as a long-long integer and eventually unmasking the formed Fock matrix when all ERIs are evaluated.  However, this step reduces the accuracy of the Fock matrix even though ERIs are evaluated using double-precision. Hence, we say this step is an accuracy bottleneck and the main source of error. Another concern beyond accuracy is efficiency.

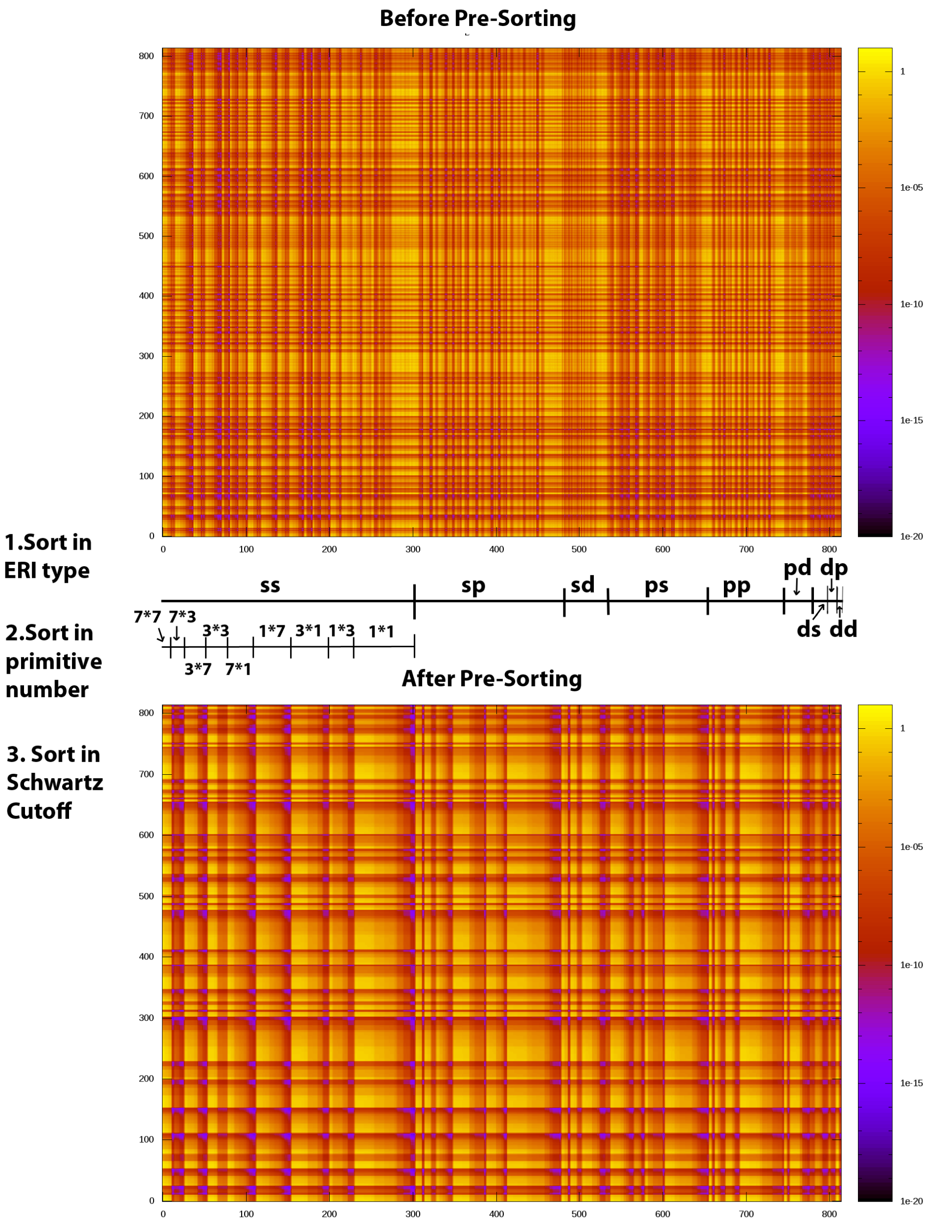


**Figure 3.1.** Pre-sorting scheme for half ERIs. Row and column indices correspond to bra and ket pairs of contracted integrals, and the colors reflect the estimated magnitude of the ERI value. Sorting order is described in the text.

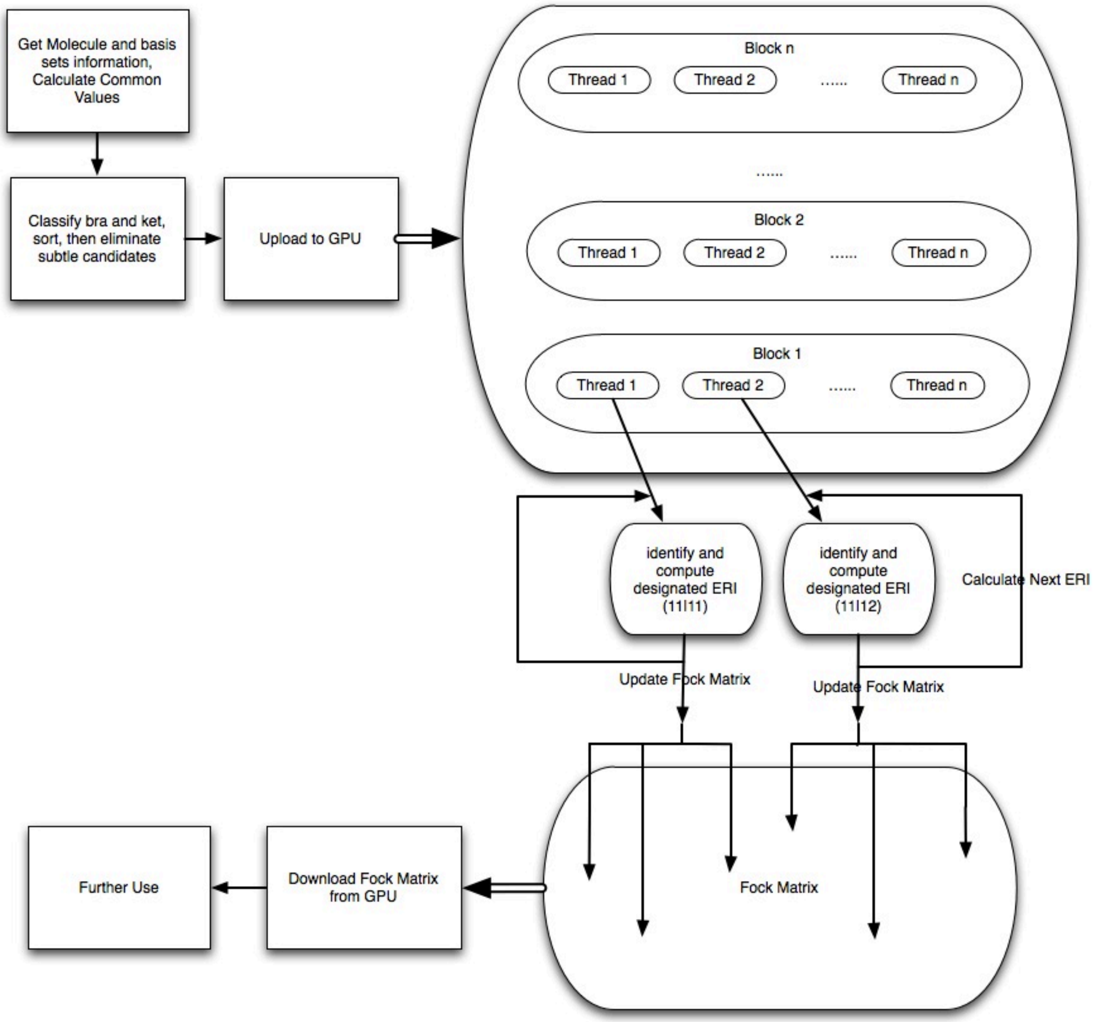

Get Molecule and basis sets information, Calculate Common Values
Classify bra and ket, sort, then eliminate subtle candidates
Upload to GPU
Block n
Thread 1
Thread 2
......
Thread n
......
Block 2
Thread 1
Thread 2
......
Thread n
Block 1
Thread 1
Thread 2
......
Thread n
identify and compute designated ERI (11|11)
identify and compute designated ERI (11|12)
Calculate Next ERI
Update Fock Matrix
Update Fock Matrix
Fock Matrix
Download Fock Matrix from GPU
Further Use

**Figure 3.2.** Flowchart for GPU implementation of ERI evaluation within an SCF cycle

If a thread is supposed to access an address that is locked by another thread that wants to access the same memory, that thread will hold until the occupying thread unlocks the memory address. In that case, the more threads that are idle and waiting to write to one identical memory address in sequence, the lower the efficiency will be. This will be a factor preventing better efficiency for small systems with crowded memory access. However, in our implementation, a typical launch bound for a CUDA 2.0 card is a GPU with 16 blocks and 512 threads, if one thread generates one integral and that integral is used 8 times at maximum, then 8192 integrals are calculated in one loop and a maximum value of 65536 memory addresses are requested to be modified. The Fock matrix has a N*N elements where N is number of basis functions. For a system with 1000 basis functions, only 6.6% of the memory addresses are on the list to be updated. So it is less likely that two threads will want to write to the same address, especially for larger systems, and actually only a few of the threads are active while others are waiting for global memory access due to the memory latency or inactive due to lack of registers.

In conclusion, atomic operations will not lead to significant system inefficiency besides what is expected for normal memory access. To demonstrate this, we compared two ways to deal with ERIs: (1) form the Fock matrix in the CPU after transferring the ERIs from the GPU (labeled as conventional SCF) and (2) form the Fock matrix in the GPU using atomic operations (labeled as direct SCF). The model we used is an N * N hydrogen-mesh with neighboring distance of 1 Å. The systems used here are "toy" models because there are only s-orbitals required which exclude the overhead brought by thread convergence.  Table 3.1 shows the timing results produced by the two methods, conventional SCF and direct SCF. All those results are based on 64-bit double-precision computation on an NVIDIA M2090 graphic processing unit card. For our conventional SCF code, we have modified it to avoid any disk I/O operations that may seriously reduce computation efficiency so that the two methods can be compared fairly. Kernel time includes floating-point calculation and data-fetch latency, and the majority of CPU-GPU transfer time is ERI transfer for conventional SCF and the Fock matrix for direct SCF. The first notable result in table 3.1 is the impressive speedup by GPU computation, and both ways provide significant improvement compared to a CPU no matter the kernel or wall time. If only kernel time is considered, direct SCF provides about a 40-fold speedup while the conventional method provides a 150-fold increase. The 150-fold speedup can be ruled as the maximum speedup that GPU can achieve in an ideal situation, without any overhead such as data latency and thread convergence. However, the data transfer time of conventional SCF is as much as the kernel time while it is less than 0.5% wall time and negligible for direct SCF. The kernel time of direct SCF is about 3 times more than conventional SCF, except it includes the Fock matrix assembly time, one of the reasons is the penalty from the atomic operators. However, even the overall time of conventional SCF is close to that of direct SCF in this case, data-transfer time and memory requirement scale $N^4$ for conventional SCF but $N^2$ for direct SCF, therefore, memory will be quickly exhausted for conventional SCF (the maximum number of basis functions is <300 with a 6 GB GPU Memory) and takes more time for data transfer (via a PCI Express slot, 2.0GB/s) which makes calculations on larger systems impossible. So, as noted before, it is necessary to assemble the Fock matrix in the GPU and abandon sophisticated code optimized for CPUs over years to avoid expensive copy operations.

**Table 3.1.** Timing for two SCF schemes for Hydrogen Atom System ERI Evaluation [a]

| Hydrogen Number | | Direct SCF | | Conventional SCF | | CPU Time/s |
|---|---|---|---|---|---|---|
| | | Kernel Time/s | GPU-CPU transfer/s | Kernel Time/s | GPU-CPU transfer/s | |
| 8*8 | STO-3G | 0.183 | 0.062E-3 | 0.051 | 0.008 | 6.30 |
| | 6-31G | 0.443 | 0.229E-3 | 0.127 | 0.086 | 14.97 |
| 10*10 | STO-3G | 0.454 | 0.049E-3 | 0.121 | 0.032 | 19.00 |
| | 6-31G | 1.178 | 0.289E-3 | 0.340 | 0.447 | 47.40 |
| 12*12 | STO-3G | 0.918 | 0.101E-3 | 0.274 | 0.139 | 46.18 |
| | 6-31G | 2.630 | 0.789E-3 | 0.770 | 1.942 | 121.83 |

[a] GPU timing refer to kernel time and GPU-CPU data transfer time. CPU time refers to the Fock matrix formation time. Unit is second.

## 3.4. RESULTS AND DISCUSSION

Before we began tests on our GPU code, we implemented the same recurrence relations for a CPU on QUICK[27]. Our test indicates it has competitive efficiency with GAMESS as shown in Table 3.2. The GAMESS version is August 11, 2011 R1 64-bit under Linux with Intel FORTRAN Compiler 10.1.15(shown as GAMESS/intel) and the GNU FORTRAN compiler 4.1.2(shown as GAMESS/gnu) using the default configuration, while QUICK is compiled using the same Intel compiler with optimization option level 3 (-O3). The test cases are Hartree-Fock SCF computation on sets of representative molecules of small and medium size, such as a water cluster with 16 water molecules, an (Alanine)$_3$ chain, and a (Glycine)$_{12}$ chain using the 6-31G, 6-31G* and 6-31G** basis sets. In the calculations most setups are default except the direct option is selected for all systems and the integral cutoff is set as $10^{-9}$ ensuring a fair comparison. The benchmark is performed on the same machine with an AMD Opteron Processor 2427 CPU, and we list their first iteration times in Table 3.2. QUICK performs competitively in most categories, therefore, we conclude that QUICK's implementation is as efficient as a standard quantum chemistry package.

In the GPU version, all primitive ERIs are calculated on-the-fly due to register shortage as previously described. In contrast, in the CPU version, because of the considerably larger amount of memory provided, primitive ERIs are saved in memory temporarily to form contracted ERIs, which leads to about a two- to four-fold speedup for split Gaussian basis sets. In the following benchmark test, both CPU and GPU codes use the direct SCF procedure with the 2-electron integral cutoff set as starting from $10^{-9}$ that will be changed to tighter criteria after reach convergence threshold. A Schwarz upper bound cutoff is used to pre-screen small ERI blocks.

**Table 3.2.** Time Comparisons Between QUICK and GAMESS[a]

| | | QUICK | GAMESS/gnu | GAMESS/intel |
|---|---|---|---|---|
| 6-31G | $(H_2O)_{16}$ | 4.7 | 7.5 | 6.4 |
| | $(\text{Alanine})_3$ | 6.2 | 5.2 | 4.4 |
| | $(\text{Glycine})_{12}$ | 52.4 | 51.0 | 43.3 |
| 6-31G* | $(H_2O)_{16}$ | 11.3 | 17.4 | 14.9 |
| | $(\text{Alanine})_3$ | 21.4 | 17.3 | 15.0 |
| | $(\text{Glycine})_{12}$ | 175.6 | 183.2 | 154.1 |
| 6-31G** | $(H_2O)_{16}$ | 21.5 | 33.4 | 29.4 |
| | $(\text{Alanine})_3$ | 26.3 | 27.7 | 23.8 |
| | $(\text{Glycine})_{12}$ | 225.0 | 264.3 | 224.9 |

a. The number in the table indicates CPU time in the first iteration for the Fock matrix formation time. All values are reported in seconds. Compiler and platform information is described in the text.

The QUICK GPU code is rewritten in C++, using the Intel C++ Compiler 10.1.15 and CUDA compiler 4.0 v0.2.1221 with optimization option level 3(-O3). In the CUDA compiler, the fast math library option is on (-use_fast_math), CUBLAS 4.0 is used for linear algebra especially matrix-matrix multiplication. Most efficient comparison-based sorting algorithms have a complexity of $O(n^2 * \log n)$, and in our case, n=N2, where N is the number of unique shells, therefore, the scale of sorting is $O(N^2 * \log N)$, which is smaller than ERI scale and linear algebra; therefore, sorting will not be a bottleneck in terms of scale. In the benchmark, we compare performance on one typical CPU processor used in high-performance computation (AMD Opteron™ Processor 2427) and one of the state-of-the-art GPU(TESLA M2090, with ECC off). An M2090 card has a specification of 6 Gigabytes memory size, with 16 Streaming Multiprocessor and 512 CUDA cores. We set the block number equal to the number of stream multiprocessors in a device, 16 in this case. Timing is set for the Fock matrix formation time including CPU-GPU data transfer except where mentioned otherwise.

The first set of test cases were linear (Alanine)$_n$ chains. We chose different Gaussian split type basis sets varied from 3-21G to 6-311G**, and the chain length is from n = 1 to n = 30 ($C_{90}N_{30}H_{152}O_{30}$) with up to 3762 basis functions. The speedup comparison for GPU and CPU to form the Fock Matrix in the first iteration is presented in Figure 3.3, while we compare the energy deviation of the GPU relative to the CPU in the first iteration in Figure 3.4. From Figure 3.3, an obvious trend is that the speed up increase is directly related to the system size, for example, for the 6-31G* basis set, GPU calculation ranges from 7.47 times faster for a single Alanine to as much as 27.40 times faster for 30 Alanine residues. This tendency can be explained by three factors. First, with larger systems and larger basis sets, the threads in the GPU are better balanced, and therefore less thread divergence occurs, which is important to avoid in CUDA programming. Second, the density matrix is quadratic with respect to the number of basis functions while the thread number is fixed in GPU, so if a larger number of basis functions are used, there is less chance that two threads modify the same address when they finish ERI evaluation.  In contrast, threads will take atomic operations “crowdedly” if the density matrix size is small, which leads some threads to adopt idle states that slows calculation speed. The third reason is because of the better pre-screen strategy including the Schwarz cutoff used in GPU and skipping a large amount of memory read requests or atomic write requests, which are the bottleneck in GPU computation. The CPU also uses a similar cutoff scheme, and its memory access is fast compared with the GPU memory access, so the increased speedup is observed if more ERIs are treated as negligible by the cutoff.

An interesting observation is double-polarized basis sets are generally most accelerated and their speedup increases fastest, while non-polarized basis sets are the least accelerated but the speedup increase is second fastest. There is less of a speedup for small molecules but a greater speedup for large molecules compared with single-polarized basis sets. This is because polarized basis sets will have fewer integrals cutoff but possess a large number of basis functions, which take advantage of reason 2 above but goes against reasons 1 and 3. So for double-polarized basis sets, reason 2 dominates reasons 1 and 3, while for single-polarized basis set this is switched.

In Figure 3.4, energy differences are shown. Since all of the calculations are double precision except for atomic operations, the only error beyond numeric error is brought on by atomic operations. We found all the absolute errors are within $10^{-10}$ Hartrees for a system with a $-10^{5}$ Hartree electronic energy, so the relative error is on the order of $10^{-16}$, which is approximately double-precision accuracy magnitude. This error increases with the increase of system size and is due to error accumulation as expected. We analyze the error growth and find the growth scale is about $n^{2.5\sim3.0}$, particularly for 6-311G* and 6-311G**, two big basis sets with significant enough error to provide a meaningful fit, the scale is $n^{3.01\pm0.05}$and $n^{2.77\pm0.05}$ (both ignoring the first ten points). The error growth scale is quite close to the ERI growth scale. Therefore, in conclusion, the ERI error is essentially zero because they are calculated as double precision, and the main error is from the atomic error. This is because to work with atomic operations, ERIs are masked as long-long integers leading to approximate 10 decimal digits of accuracy which is less than double-precision (~15.9), but more than single-precision (~7.2). These calculations are far better than single-precision GPU ERI evaluation, which only has a $10^{-3}$ a.u. accuracy because of error accumulation. The error order we achieve, $10^{-10}$ Hartrees, is accurate enough for most chemistry calculations especially considering the accompanying speedup.

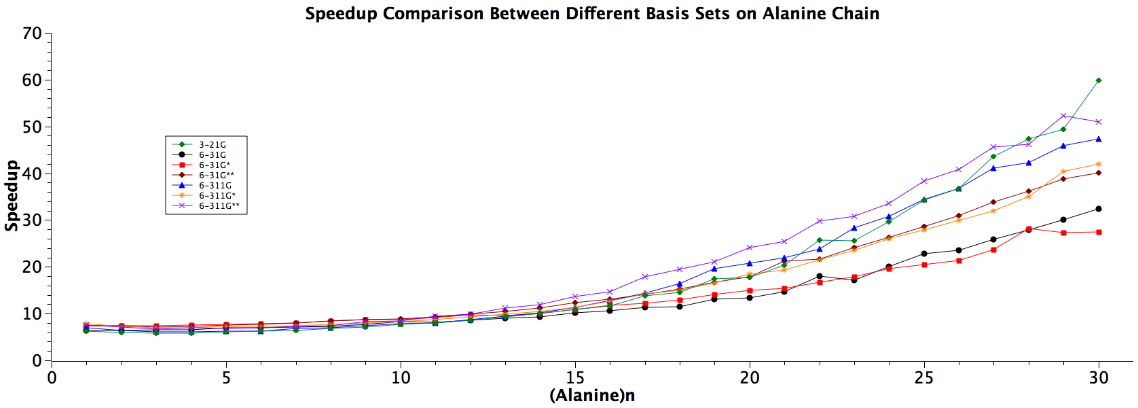


**Figure 3.3.** Speedup comparisons between different basis sets on Alanine chain series. Timing for CPU and GPU are their first iteration Fock matrix formation time including data transfer time. Platform and software details are described in text.

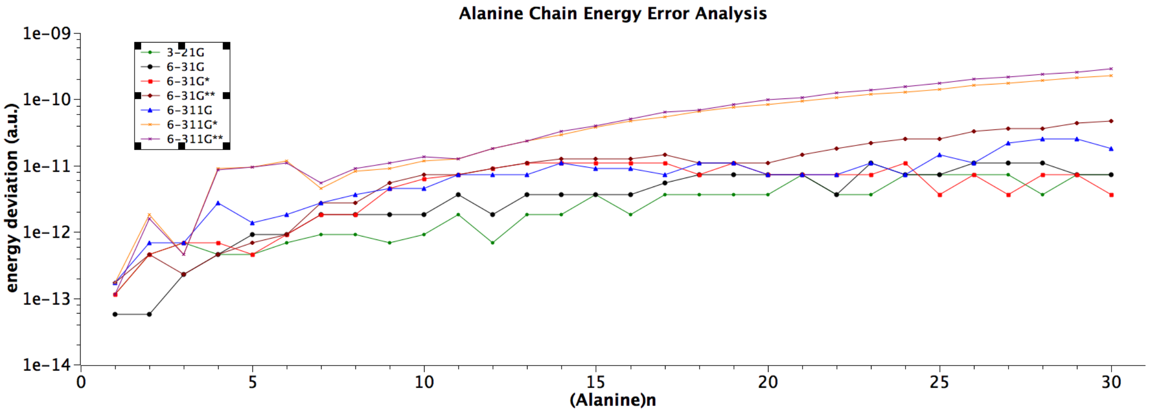


**Figure 3.4.** Energy deviation comparisons between different basis sets on Alanine chain series. Logarithmic scale Y-axis is used.

Further tests are full SCF calculations on some prototypical systems such as the $3_{10}$-helix acetyl(ala)$_{18}$NH$_2$, the α-helix acetyl(ala)$_{18}$NH$_2$, the β-strand acetyl(ala)$_{18}$NH$_2$, an ice crystal structure and a water-cluster together with some mid-size systems like taxol, valinomycin, olestra and proteins with PDB[28] ID of 1M2C(α -conotoxin mii), 1OMG(ω-conotoxin mviia), 1VTP(vacuolar targeting peptide). The 6-31G*, 6-31G** and 6-311G basis sets were used for the large systems and for some of the small systems the 6-311, 6-311G** and cc-pVDZ basis sets were used. The system sizes range from 110 atoms to 453 atoms with up to 4015 basis functions. The geometries of the above-mentioned molecules were taken from the literature or constructed and optimized in-house. In these calculations, in addition to the calculation settings used in the Alanine chain series, we turned on the DIIS (direct inversion in the iterative subspace) SCF[29] option in order to accelerate SCF convergence. The convergence criteria were set to a density matrix RMS (root mean square) difference within $10^{-7}$, and an energy change of less than $10^{-9}$ Hartrees. The Fock matrix will be calculated using differences from the previous iteration, which minimizes the number of Fock matrix elements that need to be updated, especially for late iterations. A staged integral cutoff strategy was also used to save time in the early stages of the SCF.

The results are presented in Table 3.3. We list first iteration time, last iteration time and compare the converged total energy deviation between the CPU and GPU result. From the table, we observe that a single GPU can speedup a SCF calculation by up to 130-fold compared to a single CPU. For relatively small systems, GPU direct SCF calculations are about 10 times faster than CPU calculations. The speedup, as discussed above, depends on the basis set type and the number of basis functions: large basis sets and weakly-interacting basis sets which lead to more integrals being cutoff will provide a higher speedup. Moreover, if we compare the three different helix conformations, we note that the speedup of the β-strand calculation outperforms that of the α-helix and $3_{10}$-helix calculations when same basis sets are used. This is because the α-helix structure is more compact, while the β-strand structure is less compact benefitting from a higher amount of integrals being cutoff. We also find that the olestra molecule and water cluster $(H_2O)_{100}$ yields impressive speedups, about 100 time faster for 6-31G, due to the large amount of hydrogen atoms, where a large portion of the integrals are of the (ss|ss) type, which requires the least computational cost and memory access. According to Table 3.1, for a hydrogen mesh with the 6-31G basis set, in the conventional SCF scheme, the kernel time in GPU versus CPU can reaches a 150 times speedup, which is the maximum speedup a GPU of the type employed can achieve. Therefore, the speedup achieved for olestra and water clusters are close to the ideal kernel efficiency with negligible atomic operation penalties. The energy error including ERI error and CUBLAS error will accumulate with iterations, therefore, for a full SCF calculation, the magnitude of energy error increases to $10^{-7}$ after error accumulation for a large system with more than 3000 basis functions.

However, since time used in the linear algebra subroutines, especially matrix-matrix multiplication (we use the DGEMM subroutine from NVIDIA CUBLAS) and the diagonalization routine (we do this calculation on the CPU), scale cubically with the number of the basis functions (these calculations scale larger than does ERI computation) for large systems, indicates that linear algebra plays a significant role in the total calculation time. For example, in the SCF calculation for olestra with the 6-31G** basis set, in the first iteration, the time for ERI evaluation is 371.51s (37.7% of Fock matrix formation time) while diagonalization is 323.68 seconds (32.9% of Fock matrix formation time) and the DIIS time (excluding the diagonalization time) is 289.45 seconds (29.38% of Fock matrix formation time), while the ERI evaluation and DIIS times in CPU were 50009.94 seconds and 16079.17 seconds. Hence, the GPU achieves a 134.6 and a 55.6 times speedup respectively, while the diagonalization time is almost unchanged, and, indeed, it is almost as much as the ERI evaluation time on a GPU. Thus, because of the diagonalization routine the maximum achievable speedup will not be realized.

In this series of benchmark computations, the largest peak DRAM usage is 1.25 Gigabytes, which is for olestra using 6-31G**. Peak memory usages for some larger systems were tested as well, and the results are listed in Table 3.5. Memory usage is in a trade off with speed as described above by storing pre-calculated values and they, together with necessary molecule basis set and electron structure information such as one-electron operator matrix and density matrix, represent most of the memory usage. Ideally, memory usage is quadratic with the number of unique shells, so for an M2090 card with 6.0 Gigabytes, the maximum systems are about 10000 basis sets because of well-designed pre-screening strategies. We could avoid the pre-calculation step to reduce memory usage but this will sacrifice speed for large systems.

In addition, we did more test cases using different devices, and we find the speed up is not significantly different (~10-20%) when lower level cards were used (M2070, GTX580 for example), as shown in Table 3.4. For some small basis sets, 6-31G for instance, about a 20% advantage over GTX 580 is realized using a M2090, but for some larger basis sets, such as 6-311G**, the advantage is less obvious. This is because the bottleneck for GPU calculation is memory bandwidth rather than floating-point calculation, and in terms of registers and shared memory, the Tesla M2090 card does not have a significant advantage over GTX GPUs, and so the test result matches our expectations. But we do recommend Telsa or better cards for their stability and fault tolerance.

## Table 3.3. Accuracy and Performance Comparison Between CPU and GPU calculation [a]

| Molecule<br>(atom number) | Basis sets<br>(function number) | 1st iteration/s | | | Last iteration/s | | | Energy | |
|---|---|---|---|---|---|---|---|---|---|
| | | CPU | GPU | speedup | CPU | GPU | speedup | CPU/a.u. | GPU/*10$^{-9}$ a.u. |
| Taxol(110) | 6-311G(940) | 513.26 | 70.74 | **7.26** | 823.86 | 76.53 | **10.77** | -2909.149312868 | -2 |
| | 6-311G**(1453) | 1789.80 | 203.97 | **8.77** | 3611.05 | 277.49 | **13.01** | -2910.445038007 | -1 |
| | cc-pvDZ (1160) | 1201.54 | 116.87 | **10.28** | 2694.33 | 174.99 | **15.40** | -2910.042418906 | 1 |
| Valinomycin(168) | 6-311G(1284) | 1343.53 | 153.67 | **8.74** | 2031.90 | 182.15 | **11.16** | -3770.243737052 | -4 |
| | 6-311G**(2022) | 4694.71 | 393.55 | **11.93** | 8601.67 | 612.14 | **14.05** | -3772.058214363 | -5 |
| | cc-pvDZ (1620) | 3451.95 | 340.87 | **10.13** | 6089.49 | 458.61 | **13.28** | -3771.439946125 | 6 |
| Ice-like $(H_2O)_{80}$(240) | 6-311G(1520) | 3073.54 | 109.57 | **28.05** | 3153.74 | 109.48 | **28.81** | -6082.305528935 | 5 |
| | 6-311G**(2480) | 9953.00 | 246.84 | **40.32** | 10241.20 | 300.18 | **34.12** | -6084.520235828 | 3 |
| | cc-pvDZ (2000) | 6776.11 | 93.51 | **72.46** | 10684.27 | 295.63 | **36.14** | -6083.023206911 | -1 |
| Ice-like $(H_2O)_{96}$(288) | 6-311G(1824) | 6639.24 | 165.18 | **40.19** | 9518.88 | 325.39 | **29.25** | -7298.891219499 | 7 |
| | 6-311G**(2976) | 20754.41 | 366.43 | **56.64** | 29402.37 | 925.64 | **31.76** | -7301.528543939 | 6 |
| | cc-pvDZ (2400) | 14534.58 | 136.77 | **106.27** | 20173.25 | 449.68 | **44.86** | -7299.732249179 | -2 |
| $(H_2O)_{32}$(96) | 6-311G(608) | 134.77 | 14.40 | **9.36** | 128.77 | 10.78 | **11.95** | -2431.716629896 | 2 |
| | 6-311G**(992) | 379.50 | 34.05 | **11.15** | 957.65 | 66.14 | **14.48** | -2432.703288503 | 1 |
| | cc-pvDZ (800) | 254.58 | 14.21 | **17.92** | 674.14 | 32.37 | **20.83** | -2432.055391595 | 2 |
| $(H_2O)_{100}$(300) | 6-311G(1900) | 7157.31 | 104.32 | **68.61** | 8728.88 | 141.65 | **61.62** | -7602.094350092 | -1 |
| | 6-311G**(3100) | 25236.82 | 252.80 | **99.83** | 31204.21 | 486.81 | **64.10** | -7605.175710929 | 2 |
| | cc-pvDZ (2500) | 16049.72 | 100.04 | **160.43** | 20052.53 | 200.23 | **100.15** | -7603.361888329 | 22 |

Table 3.3. (cont'd)

| | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|
| $3_{10}$-helix acetyl(ala)$_{18}$NH$_2$(189) | 6-311G(1507) | 1923.62 | 173.75 | **11.07** | 2842.48 | 227.14 | **12.51** | -4632.003799593 | -17 |
| | 6-311G**(2356) | 6639.98 | 472.74 | **14.05** | 11533.48 | 799.10 | **14.43** | -4634.169869157 | -7 |
| | cc-pvDZ(1885) | 5800.82 | 541.90 | **10.70** | 11243.90 | 806.39 | **13.96** | -4633.502073111 | 11 |
| α-helix acetyl(ala)$_{18}$NH$_2$(189) | 6-311G(1507) | 2053.30 | 194.58 | **10.55** | 3164.16 | 233.21 | **13.57** | -4632.315977523 | -7 |
| | 6-311G**(2356) | 7083.52 | 507.06 | **13.97** | 11889.02 | 822.49 | **14.45** | -4634.476339611 | -13 |
| | cc-pvDZ(1885) | 6004.15 | 541.96 | **11.08** | 10230.45 | 726.92 | **14.07** | -4633.816772135 | 3 |
| β-strand acetyl(ala)$_{18}$NH$_2$ (189) | 6-311G(1507) | 1499.20 | 89.95 | **16.67** | 2101.61 | 123.06 | **17.08** | -4632.231275219 | -6 |
| | 6-311G**(2356) | 5572.93 | 263.14 | **21.18** | 9102.65 | 499.50 | **18.22** | -4634.420919368 | 2 |
| | cc-pvDZ(1885) | 4970.84 | 440.33 | **11.29** | 8287.50 | 575.00 | **14.41** | -4633.750359062 | 147 |
| α -conotoxin mii<br>(PDBID: 1M2C, 3+1-)(220) | 6-31G*(1964) | 3978.10 | 345.70 | **11.51** | 13309.37 | 973.75 | **13.67** | -7106.869678943 | 15 |
| | 6-31G**(2276) | 5453.77 | 409.36 | **13.32** | 18046.87 | 1299.04 | **13.89** | -7107.108358427 | 27 |
| | 6-311G(1852) | 4485.98 | 359.74 | **12.47** | 11311.33 | 902.68 | **12.53** | -7105.597849172 | 3 |
| ω-conotoxin mviia<br>(PDB ID: 1OMG, 7+2-) (353) | 6-31G*(3035) | 17473.01 | 730.71 | **23.91** | 23381.07 | 1081.33 | **21.62** | -11122.046450828 | 5 |
| | 6-31G**(3563) | 26247.84 | 869.72 | **30.18** | 33451.28 | 1439.35 | **23.24** | -11122.460728457 | 16 |
| | 6-311G(2885) | 28709.47 | 812.89 | **35.32** | 30734.31 | 1171.02 | **26.25** | -11120.201936663 | -13 |
| Olestra(453) | 6-31G*(3181) | 22626.95 | 306.09 | **73.92** | 25706.94 | 539.59 | **47.64** | -7491.143229057 | -165 |
| | 6-31G**(4015) | 53992.79 | 402.68 | **134.08** | 58037.78 | 853.27 | **68.02** | -7491.573825553 | -250 |
| | 6-311G(3109) | 51047.04 | 358.01 | **142.59** | 59152.48 | 1035.48 | **57.13** | -7489.506716913 | 1 |

Table 3.3. (cont'd)

| | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|
| vacuolar targeting peptide (PDB ID: 1VTP, 4+7-) (396) | 6-31G*(3418) | 24196.66 | 1041.92 | **23.22** | 31079.79 | 1382.36 | **22.48** | -10014.756057438 | 16 |
| | 6-31G**(4000) | 38460.44 | 941.06 | **40.87** | 51952.78 | 1849.08 | **28.10** | -10015.181384177 | 23 |
| | 6-311G(3208) | 34865.17 | 895.09 | **38.95** | 51292.98 | 1305.71 | **39.28** | -10012.680303881 | -15 |

[a] GPU energy column lists relative energies with respect to corresponding CPU calculation. Platform and software details are described in text. [b] PDB ID and number of positive and negative charge center included in parentheses.

**Table 3.4. Timing for GPU version calculation using different devices [a]**

| | (Glycine)12 Chain | | | | | | Taxol | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | 6-31G | 6-31G* | 6-31G** | 6-311G | 6-311G* | 6-311G** | 6-31G | 6-31G* | 6-31G** | 6-311G | 6-311G* | 6-311G** |
| **M2090(6.0 GB)[b]** | 7.92 | 24.70 | 29.48 | 18.30 | 47.93 | 61.48 | 28.63 | 81.22 | 96.91 | 70.73 | 171.37 | 215.64 |
| **M2070(6.0 GB)** | 9.16 | 28.50 | 33.89 | 21.03 | 55.19 | 65.55 | 32.82 | 94.09 | 112.22 | 80.99 | 194.59 | 232.30 |
| **GTX580(1.5 GB)** | 10.00 | 26.06 | 30.82 | 22.25 | 51.08 | 60.48 | 35.10 | 87.00 | 102.61 | 85.54 | 183.57 | 217.39 |
| **CPU[c]** | 73.33 | 275.22 | 342.79 | 175.16 | 495.83 | 628.43 | 257.80 | 828.51 | 987.13 | 582.52 | 1622.83 | 1389.97 |

a. Number in the table shows the Fock Matrix formation time in GPU in seconds. b. global memory size(also know as DRAM) included in parentheses. c. CPU calculation is on AMD Opteron$^{tm}$ Processor 2427.

**Table 3.5. Peak Memory Usage Comparison [a]**

| Molecule | Number of Atoms | 6-31G | | 6-31G* | | 6-31G** | | 6-311G | | 6-311G* | | 6-311G** | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | | Number of Basis Functions | Peak Memory Usage | Number of Basis Functions | Peak Memory Usage | Number of Basis Functions | Peak Memory Usage | Number of Basis Functions | Peak Memory Usage | Number of Basis Functions | Peak Memory Usage | Number of Basis Functions | Peak Memory Usage |
| 1VTP | 396 | 2206 | 674MB | 3418 | 880MB | 4000 | 1049MB | 4420 | 963MB | 5002 | 1224MB | 4000 | 1427MB |
| Crambin | 642 | 3597 | 1796MB | 5559 | 2338MB | 6509 | 2785MB | 5244 | 2589MB | 7206 | 3279MB | 8151 | 3818MB |
| 1BQ9 | 765 | 4318 | 2570MB | 6694 | 3357MB | 7801 | 3985MB | 6287 | 3687MB | 8663 | 4686MB | 9770 | 5441MB |

a. All calculations are on Telsa M2090 6.0 GB GPU.

## 3.5. CONCLUSIONS

In this chapter, we evaluate ERIs on a GPU using recurrence relations and form the Fock matrix entirely in the GPU. Our full SCF benchmark calculations (along with earlier work described in the introduction) demonstrate that GPU ERI implementations achieve impressive speedups compared to traditional CPU architectures. The energy error associated with double-precision calculations in our implementation is on the order of $10^{-7}$ a.u. compared to the CPU result, which meets most chemical calculation accuracy requirements. A well-sorted integral grid that reduces thread divergence and provides an optimized memory access pattern boosts the performance in terms of accuracy and efficiency. This speedup is also achieved by optimizing the Fock matrix formation scheme by introducing the atomic-operation to significantly reduce data transfer from $N^4$ to $N^2$, which is one of the most time-consuming steps in conventional GPU SCF programming. Moreover, this approach also reduces redundant ERI calculation by reusing ERI data. Our benchmarks show the speedup increases with increasing system size, and our code now is applicable to s, p and d orbital functions which are in most most organic or biochemistry calculations.

Even with the observed performance increases there are several avenues for improvement. In full SCF calculations, diagonalization is executed on the CPU largely because it is hard to efficiently implement on a GPU. In some systems, this dominates computation time, so it is necessary to introduce highly efficient GPU-based diagonalization routines. Moreover, like most other GPU ERI evaluation schemes, register shortage is a crucial factor that limits GPU speedup, and this can be improved by even more aggressive memory caching and shared memory usage. As shown in our benchmarks, atomic operators bring considerable penalties especially for small molecules. Also it brings along a major energy error because of the limited function support.

Our future work will focus on two aspects. First is the ERI derivative with which geometry optimization or molecular dynamics can be implemented and we are working to integrate this work with the AMBER MD package so that large-scale *ab initio* QM/MM is readily available. Another direction we are working on is post-HF methods such as MP2 and Coupled-Cluster methods. However, most post-HF methods reuse ERIs in different stages so a proper strategy is to store ERIs in external files, but in GPU implementation this treatment of ERIs will inevitably transfer calculated ERIs from the GPU to CPU, which as mentioned above, is very slow. So how to create efficient post-HF methods entirely in GPUs is still an open question. In addition, higher angular momentum function ERI evaluation is under development. We will refer to these questions in the next chapter.

# APPENDIX

# MANUAL AND SAMPLE INPUT FILE OF QUICK

The input file of QUICK follows the philosophy “Simple is better”, so software designer tried to minimize user’s learning cost.

## Installation

To install QUICK, first you may need to configure make.in file. Intel Fortran compiler is recommanded.

*! Copy corresponding make.in file from makein folder and rename it to make.in.*

Then type

*make quick*

If you want to install GPU QUICK, NVIDIA CUDA COMPILER is required, which uses make.in.gnu.cuda. You may test 'nvcc --version'. Copy make.in.gnu.cuda to from makein folder to your directory and remane it to make.in

*cp ./makein/make.in.gnu.cuda ./make.in*

Before compiler, you must set CUDA_HOME in make.in file if not set in bashfile, otherwise, CUBLAS and other libraries can not been compiled or linked. Notice, if you want to support f orbital, set CUDA_SPDF=y in make.in file you copied from make.in.gpu.cuda, which will take a little bit long time and memory to compile, otherwise, set CUDA_SPDF=n for s, p and d orbital. (Default is not)

*! Modify CUDA_HOME=(your cuda home) in make.in file*

*! Modify CUDA_SPDF=y or n in make.in file if you want or do not want to support gpu f function*

Then type

```
make quick.cuda
```

in ./bin directory, you can find executable files.

## Usage and input file

We suggest you to export install directory into PATH. Edit ~/.bashrc, then add this line

```
export PATH=(YOUR QUICK DIR)/bin:$PATH
```

and add basis set path

```
export QUICK_BASIS=(YOUR QUICK DIR)/basis
```

where (YOUR QUICK DIR) is your directory.

In the input file, the first line is calculation card. For example

```
HF ncyc=3 energy BASIS=6-31gs denserms=1.0e-7
```

This means HF calculation is wanted, and after 3 cycle, we only calculate difference of Fock matrix, basis set is 6-31g**, and the convergence criteria is 1.0E-7. If you don't know anything about quantum chemistry, this card is recommended. After first line, you input your molecule geometry after a empty line. For example

“

```
HF ncyc=3 energy BASIS=6-31gs denserms=1.0e-7


C 4.5916 2.6127 4.3145
H 6.8049 4.5138 -2.5775
H 8.3134 7.7325 -1.0688
H 4.3547 2.9911 3.3201
H 4.5176 3.5187 3.7129
```

”

which is the element and x,y,z coordinates respectively. Save the file as *CH4.in* for example, next, type

*quick CH4.in*

or GPU version

*quick.cuda CH4.in*

or MPI version

*quick.MPI CH4.in*

These will generate CH4.out as output file (to master node if MPI version is used).

## More comments about GPU version

Currently, QUICK support CUDA version 2.0 and up GPU, that supporting double-precision. We tested on GTX580, M2090, K20 and K40 on minimum Linux system. We can't guarantee our program is flawless, and it is experimental now. If you have any question feel free to email the author.

# REFERENCES

# REFERENCES

(1) NVIDIA. Compute Unified Device Architecture(CUDA). http://developer.download.nvidia.com/compute/DevZone/docs/html/C/doc/CUDA_C_Programming_Guide.pdf (accessed October 31, 2012)

(2) Götz, A. W.; Wölfle, T.; Walker, R. C. In *Annual Reports in Computational Chemistry*; Ralph, A. W., Ed. 2010; Vol. 6, p 21-35.

(3) D.A. Case; T.A. Darden; T.E. Cheatham, I.; C.L. Simmerling; J. Wang; R.E. Duke; R. Luo; R.C. Walker; W. Zhang; K.M. Merz; B.P. Roberts; B. Wang; S. Hayik; A. Roitberg; G. Seabra; I. Kolossváry; K.F. Wong; F. Paesani; J. Vanicek; J. Liu; X. Wu; S.R. Brozell; T. Steinbrecher; H. Gohlke; Q. Cai; X. Ye; J. Wang; M.-J. Hsieh; G. Cui; D.R. Roe; D.H. Mathews; M.G. Seetin; C. Sagui; V. Babin; T. Luchko; S. Gusarov; A. Kovalenko; Kollman, P. A. *AMBER11,University of California, San Francisco*. **2010**.

(4) Case, D. A.; Cheatham, T. E.; Darden, T.; Gohlke, H.; Luo, R.; Merz, K. M.; Onufriev, A.; Simmerling, C.; Wang, B.; Woods, R. J. *J. Comput. Chem.* **2005**, *26*, 1668-1688.

(5) Anderson, J. A.; Lorenz, C. D.; Travesset, A. *J. Comput. Phys.* **2008**, *227*, 5342-5359.

(6) Yasuda, K. *J. Comput. Chem.* **2008**, *29*, 334-342.

(7) Ufimtsev, I. S.; Martinez, T. J. *J. Chem. Theory Comput.* **2008**, *4*, 222-231.

(8) Ufimtsev, I. S.; Martinez, T. J. *J. Chem. Theory Comput.* **2009**, *5*, 1004-1015.

(9) Ufimtsev, I. S.; Martinez, T. J. *J. Chem. Theory Comput.* **2009**, *5*, 2619-2628.

(10) Vogt, L.; Olivares-Amaya, R.; Kermes, S.; Shao, Y.; Amador-Bedolla, C.; Aspuru-Guzik, A. *J. Phys. Chem. A* **2008**, *112*, 2049-2057.

(11) Olivares-Amaya, R.; Watson, M. A.; Edgar, R. G.; Vogt, L.; Shao, Y.; Aspuru-Guzik, A. *J. Chem. Theory Comput.* **2010**, *6*, 135-144.

(12) Ma, W.; Krishnamoorthy, S.; Villa, O.; Kowalski, K. *J. Chem. Theory Comput.* **2011**, *7*, 1316-1327.

(13) Boys, S. F. *Proceedings of the Royal Society of London. Series A. Mathematical and Physical Sciences* **1950**, *200*, 542-554.

(14) Obara, S.; Saika, A. *J. Chem. Phys* **1988**, *89*, 1540-1559.

(15) Head-Gordon, M.; Pople, J. A. *J. Chem. Phys* **1988**, *89*, 5777-5786.

(16) NVIDIA. The NVIDIA CUDA Fast Fourier Transform library. http://developer.nvidia.com/cufft (accessed October 31, 2012)

(17) NVIDIA. The NVIDIA CUDA Basic Linear Algebra Subroutines (cuBLAS) library. http://developer.nvidia.com/cublas (accessed October 31, 2012)

(18) Rys, J.; Dupuis, M.; King, H. F. *J. Comput. Chem.* **1983**, *4*, 154-157.

(19) Asadchev, A.; Allada, V.; Felder, J.; Bode, B. M.; Gordon, M. S.; Windus, T. L. *J. Chem. Theory Comput.* **2010**, *6*, 696-704.

(20) Wilkinson, K. A.; Sherwood, P.; Guest, M. F.; Naidoo, K. J. *J. Comput. Chem.* **2011**, *32*, 2313-2318.

(21) Schmidt, M. W.; Baldridge, K. K.; Boatz, J. A.; Elbert, S. T.; Gordon, M. S.; Jensen, J. H.; Koseki, S.; Matsunaga, N.; Nguyen, K. A.; Su, S.; Windus, T. L.; Dupuis, M.; Montgomery, J. A. *J. Comput. Chem.* **1993**, *14*, 1347-1363.

(22) McMurchie, L. E.; Davidson, E. R. *J. Comput. Phys.* **1978**, *26*, 218-231.

(23) Luehr, N.; Ufimtsev, I. S.; Martinez, T. J. *J. Chem. Theory Comput.* **2011**, *7*, 949-954.

(24) Almlof, J. *Lecture Notes in Quantum Chemistry II, European Summer School in Quantum Chemistry* **1994**, 1–90.

(25) Strout, D. L.; Scuseria, G. E. *J. Chem. Phys* **1995**, *102*, 8448-8452.

(26) Dunning, T. H. *J. Chem. Phys.* **1989**, *90*, 1007-1023.

(27) Y. Miao, X. He, K. Ayers, Ed. Brothers, K.Merz, QUICK(version 0.09_120306), University of Florida, Gainesville, FL, 2011

(28) Berman, H. M.; Westbrook, J.; Feng, Z.; Gilliland, G.; Bhat, T. N.; Weissig, H.; Shindyalov, I. N.; Bourne, P. E. *Nucleic Acids Res* **2000**, *28*, 235-242.

(29) Pulay, P. *Chem Phys Lett* **1980**, *73*, 393-398.

# CHAPTER 4. GPU ACCELERATION ON ERI DERIVATIVE EVALUATION

## 4.1. INTRODUCTION

As we shown last chapter, Graphical Processing Units (GPUs) provide fast and accurate computational performance for a wide range of problems at a reasonable cost. Environments for developing general purpose computing on graphical processing units (GPGPU), such as the Compute Unified Device Architecture (CUDA), facilitate the creation of high-performance software for a wide range of applications even though GPUs were originally designed to render images on a monitor. Meanwhile, traditional quantum chemistry methods, such as the Hartree-Fock Self-Consistent field method (HF-SCF) and Density Functional Theory method (DFT) are widely used to rationalize the behavior of molecular systems. However, the broad application of these methods, to large systems containing 1,000s of atoms, has been limited by their high computational requirements using traditional CPUs.

GPUs, on the other hand, have recently become widely available as a general purpose-processing platform. With GPU technology, it is possible to bring supercomputing power to the desktop and achieve trillions of peak floating point operations per second (FLOPS) outperforming desktop CPUs by over an order of a magnitude. Another key factor in GPU's is becoming widely used in scientific areas is the release of NVIDIA's Computer Unified Device Architecture (CUDA) platform that eases the coding burden for GPUs with an extension of the standard C/C++ language. Several papers describing the use of GPU Computing, especially with CUDA, have demonstrated impressive speedups for a range of computational chemistry applications, including *ab intio* quantum chemistry[1-12] and its application to the simulation of biochemical reactions[13-18], post-HF methods[19-23] as well as empirical force-field and ab initio molecular dynamics[24-26] simulations.

The applications of *ab initio* computational chemistry methods are not limited to the study of the single point energetics or electronic structure of large molecular systems, but can also be used in molecular geometry optimization and molecular dynamics simulations which help scientists to simulate chemical reactions *in silico* rather than in the laboratory. However, for HF and DFT methods, not only is the single point energy required but also the computation of the energy gradients requiring significant computational resources especially in the study of large biochemical systems. Gradient calculation requires approximately the same computational effort as the SCF calculation, hence, GPU acceleration of this step will further benefit to computational chemistry community in its ongoing effort to study larger chemical systems.

In last chapter, we reported the GPU acceleration of the Electron Repulsion Integral (ERI) evaluation for s, p and d orbitals[8] to compute the HF and DFT single point energy, which further demonstrated the potential of GPU use in *ab initio* quantum chemistry methods. The realized speedup was 10~20-fold (with excellent accuracy) compared with a single core CPU for moderately sized molecular systems including proteins. However, two more challenging problems still need to be tackled for broader application of GPUs in quantum chemistry: efficient implementation of electron-correlation methods and efficient gradient computation for geometry optimization and MD simulation. GPU application to address electron-correlation has already been reported using MP2[22] and

Coupled-Cluster method[19,20,27-29] with impressive overall performance. Herein, we focus on the latter issue.

For energy computation the evaluation of ERIs with higher-angular momentum functions such as f orbitals (and beyond) becomes necessary. Due to the complexity of high-angular momentum ERIs, GPU code to compute them is especially difficult to create and fine tune because of the difference in the architectures of GPUs relative to CPUs. Another challenge is the calculation of ERI derivatives, the bottleneck of HF gradient computing, that also depend on high-angular momentum ERIs, which is the problem faced for large basis set computations, but with a memory access pattern different from that of normal ERI computations. Therefore, these challenges and the ever increasing demands from computational chemists provide the main motivation for the present work, which is an implementation and calculation of high-angular momentum ERIs and ERI derivatives on GPUs. Efforts aimed at computing high-angular momentum functions and their associated gradients on GPUs have been reported using different ERI algorithms[6,30]. In this chapter, we describe a new algorithm to overcome issues surrounding the computation of higher-angular momentum functions and their gradients using recurrence relationships carried out on current-generation GPUs with negligible loss of efficiency.

In this chapter, we first describe the algorithm we applied to evaluate ERIs, which is based on recurrence relations, one of the fastest ERI evaluation algorithms available especially for ERIs with high-angular momentum functions, and then its extension to analytical gradient computation will be introduced. In the next section, we will briefly describe how to implement ERIs with high-angular momentum functions on GPUs and apply this to direct HF and DFT calculations. Moreover, a new strategy to assemble the gradient within the HF framework is introduced. In the last section, detailed benchmarks will be presented. We performed a series of small, medium and large molecule calculations to profile the speed and accuracy performance for problems requiring thousands of Gaussian-type basis functions. Finally we conclude the chapter with a brief discussion and conclusions.

## 4.2. THEORY

Within the framework of Hartree-Fock theory, the total energy of a closed-shell system can be expressed as the sum of electron-nucleus interactions (the first term), electron-electron interactions (the second term) and nucleus-nucleus interactions (the third term) within the Born–Oppenheimer approximation

$$E_{tot} = \sum_{\mu\nu} P_{\nu\mu} H_{\mu\nu}^{core} + \frac{1}{2} \sum_{\mu\nu\lambda\sigma} P_{\nu\mu} P_{\lambda\sigma} (\mu\nu||\lambda\sigma) + V_{NN} \qquad (1)$$

Here we use the notation $(\mu\nu||\lambda\sigma) = (\mu\nu|\lambda\sigma) - (\mu\lambda|\sigma\nu)$ to describe the electron repulsion integrals (ERIs), which represent the most expensive part in equation 1 contributing to both the Coulomb and Exchange ERI terms. Four-centered ERIs are given by:

$$(\mu\nu|\kappa\lambda) = \iint \varphi_\mu(\vec{r})\varphi_\nu(\vec{r}) \frac{1}{r-r'} \varphi_\kappa(\overrightarrow{r'})\varphi_\lambda(\overrightarrow{r'}) d\vec{r} d\overrightarrow{r'} \qquad (2)$$

and the derivative of the total energy with respect to nuclear coordinate $X_A$ can be written as:

$$\frac{\partial E_{tot}}{\partial X_A} = \sum_{\mu\nu} P_{\nu\mu} \frac{\partial H_{\mu\nu}^{core}}{\partial X_A} + \frac{1}{2} \sum_{\mu\nu\lambda\sigma} P_{\nu\mu} P_{\lambda\sigma} \frac{\partial(\mu\nu||\lambda\sigma)}{\partial X_A} - \sum_{\mu\nu} Q_{\nu\mu} \frac{\partial S_{\mu\nu}}{\partial X_A} + \frac{\partial V_{NN}}{\partial X_A} \quad (3)$$

Where the density matrix is $P_{\mu\nu} = 2\sum_a^{N/2} C_{\mu a} C_{\nu a}$ and $Q_{\nu\mu} \equiv 2\sum_a^{N/2} \varepsilon_a C_{\mu a} C_{\nu a}$. As in electronic energy calculations, the major bottleneck in equation 3 is the evaluation of the ERI derivative. These can also be efficiently computed using recurrence relations in an analogous way to ERI computation as the following sections describe.

### 4.2.1. ELECTRON REPULSION INTEGRALS AND RECURRENCE RELATIONS

To evaluate ERIs, many different and efficient algorithms have been developed[31,32], for example, Dupuis, Rys and King (DRK)[33] developed the first algorithm for integral evaluation involving high-angular momentum functions, McMurchie and Davidson(MD) [34] later on used Hermite polynomials to efficiently evaluate integrals over Gaussians-type basis functions, Obara and Saika (OS)[35] and Head-Gordon and Pople (HGP)[36] developed new recursion relationships with fewer terms by focusing on shifting work outside of the contractions loops. In this work, we employed an adapted OS and HGP algorithm to generate a general algorithm that is applicable to a wide range of integral types and offered an efficient implementation on GPUs.

We represent ERIs using a linear combination of contracted Gaussian functions because of their well-known mathematical advantages.

$$\varphi_\mu(\vec{r}) = \sum_{p=1}^{N} c_{\mu p} \chi_p(\vec{r}) \quad (4)$$

A primitive Cartesian Gaussian function centered at $\vec{A} = (A_x + A_y + A_z)$ with exponent $\alpha$ is given by

$$\chi_{\vec{a}}(r) = (x - A_x)^{a_x}(y - A_y)^{a_y}(z - A_z)^{a_z}e^{-\alpha(\vec{r}-\vec{A})^2} \tag{5}$$

and the contracted two-electron integrals are constructed from primitive ERIs by:

$$(\mu\nu|\kappa\lambda) = \sum_{pqrs} c_{\mu a}c_{\nu b}c_{\kappa c}c_{\lambda d}[ab|cd] \tag{6}$$

in equation 5, $\vec{a} = (a_x, a_y, a_z)$, and $a_x$, $a_y$ and $a_z$ are a set of integers indicating angular momentum and the direction of the Gaussian function. These sums are restricted to functions with the same quantum number. It would be more efficient to compute all of the primitive ERIs involving four shells for which $\vec{a}$ has the same $a_i$. For example, 3 [ps|ss] type integrals can be computed at the same time where $\vec{a} = (1,0,0), (0,1,0)$ and $(0,0,1)$. This will lead to complex conditions for high-angular momentum ERIs, for example, [dd|dd] will have 1296 kinds of different ERIs because each index can have 6 different integer combinations to satisfy the pre-condition that the sum of these three direction integers equals 2. Therefore, the calculation complexity grows dramatically with the introduction of high-angular momentum functions such as f orbitals.

To efficiently evaluate the value of ERIs, Head-Gordon and Pople (HGP)[36] optimized the recurrence relation algorithm described by Obara and Saika (OS)[35] to reduced the floating point operation count. It is based on the recurrence equation:

$$\begin{aligned}
[(a+1_i)b|cd]^{(m)} = {} & (P_i - A_i)[ab|cd]^{(m)} + (W_i - P_i)[ab|cd]^{(m+1)} \\
& + \frac{a_i}{2\zeta}\left([(a-1_i)b|cd]^{(m)} - \frac{\eta}{\eta+\zeta}[(a-1_i)b|cd]^{(m+1)}\right) \\
& + \frac{b_i}{2\zeta}\left([a(b-1_i)|cd]^{(m)} - \frac{\eta}{\eta+\zeta}[a(b-1_i)|cd]^{(m+1)}\right) \\
& + \frac{c_i}{2(\eta+\zeta)}[ab|(c-1_i)d]^{(m+1)} + \frac{d_i}{2(\eta+\zeta)}[ab|c(d-1_i)]^{(m+1)}
\end{aligned} \tag{7}$$

Where i is x, y or z, and

$$1_i = (\delta_{ix}, \delta_{iy}, \delta_{iz}), \tag{8}$$

$$\zeta = \alpha + \beta, \eta = \gamma + \delta, \tag{9}$$

$$P_i = \frac{\alpha A_i + \beta B_i}{\alpha + \beta}, Q_i = \frac{\gamma C_i + \delta D_i}{\gamma + \delta}, W_i = \frac{\zeta P_i + \eta Q_i}{\zeta + \eta}. \tag{10}$$

and $\alpha, \beta, \gamma, \delta$ are exponents of a, b, c and d respectively. Technically, all the integrals can be ultimately computed from [ss|ss] integrals, which can be analytically evaluated efficiently[35,37].

$$[ss|ss]^{(m)} = \frac{1}{\sqrt{\eta + \zeta}} K_{AB} K_{CD} F_m(T) \tag{11}$$

where

$$T = \frac{\zeta\eta}{\zeta + \eta}(\vec{P} - \vec{Q})^2 \tag{12}$$

$$F_m(T) = \int_0^1 t^{2m} e^{-Tt^2} dt \tag{13}$$

$$K_{AB} = \sqrt{2} \frac{\pi^{\frac{5}{4}}}{\alpha + \beta} \exp\left[-\frac{\alpha\beta}{\alpha + \beta}\left(\vec{A} - \vec{B}\right)^2\right] \tag{14}$$

In, equation 7, the summation of the four indices decreases as ERIs with higher angular momentum functions are constructed from lower ones, so it is termed a vertical recurrence relation (VRR). Moreover, a horizontal recurrence relation (HRR) is also applicable for Gaussian-type ERIs

$$[a(b + 1_i)|cd]^{(m)} = [(a + 1_i)b|cd]^{(m)} + (A_i - B_i)[ab|cd]^{(m)} \tag{15}$$

and can be applied to contracted ERIs,

$$(a(b + 1_i)|cd)^{(m)} = \left((a + 1_i)b\middle|cd\right)^{(m)} + (A_i - B_i)(ab|cd)^{(m)}$$

$$= \sum_{a \le k \le a+b+1, c \le l \le c+d} c_{kl} (k0|l0)^{(m)} \tag{16}$$

Which hints that an ERI can always be expressed as a linear combination of (k0|l0) type ERIs. Therefore, a general strategy for evaluation of (ab|cd) is to calculate them from a set of integrals from (a0|b0) to ((a+b)0|(c+d)0). For example, for an ERI with a four d orbital index, (dd|dd), first we compute (ds|ds) (36 total), (fs|fs) (100 total) and (gs|gs)

(225 total) via a VRR and then assemble (dd|dd) using an HRR. Therefore, for the VRR step, only the b = 0 and d = 0 situation will be considered in equation (7), and we can obtain the (half) coefficient analytically as well from equation (16).

$$
\begin{aligned}
&[(a+1_i)0|c0]^{(m)} \\
&\quad = (P_i - A_i)[a0|c0]^{(m)} + (W_i - P_i)[a0|c0]^{(m+1)} + \frac{a_i}{2\zeta}\left([(a-1_i)0|c0]^{(m)} - \frac{\eta}{\eta+\zeta}[(a-1_i)0|c0]^{(m+1)}\right) \\
&\qquad + \frac{c_i}{2(\eta+\zeta)}[a0|(c-1_i)0]^{(m+1)}
\end{aligned}
$$

$$
(a_n b_{n'}| = \sum_{i=0}^{n'_x}\sum_{j=0}^{n'_y}\sum_{k=0}^{n'_z}\binom{n'_x}{i}\binom{n'_y}{j}\binom{n'_z}{k}(A_x - B_x)^{n'_x - i}\left(A_y - B_y\right)^{n'_y - j}(A_z - B_z)^{n'_z - k}\ \left(a_{(n_x+i)(n_y+j)(n_z+k)}0\right| \tag{17}
$$

### 4.2.2. CALCULATION OF DERIVATIVE ERIS

The derivative of a primitive Gaussian function $\chi_a(r)$ is a linear combination of a higher and a lower angular momentum Gaussian function:

$$
\frac{\partial \chi_a}{\partial \mathrm{A}_i} = 2\,\alpha\left(\chi_{a+1_i}\right) - a_i(\chi_{a-1_i}) \tag{18}
$$

Here A is the function center and i can be x, y or z. Similarly, the first derivative of a primitive ERI is also a combination of a higher and a lower primitive ERIs.

$$
\frac{\partial}{\partial \mathrm{A}_i}[\mathrm{ab}|\mathrm{cd}] = 2\,\alpha[(a+1_i)b|cd] - a_i[(a-1_i)b|cd] \tag{19}
$$

The recurrence algorithm described in the last section can be adapted to this equation except higher angular momentum ERIs are needed to compute the first derivative. For example, the first derivative of [dd|ss] requires [ps|ss] and [hs|ss] along with [ds|ss], [fs|ss] and [gs|ss] where the latter three are needed in the ERI evaluation as well. It is worth noticing that, to evaluate [dd|dd] type derivatives, for instance, [gs|gs], the integral with the most expensive computing cost, does not need to be evaluated because the first and third indices do not require higher ERIs simultaneously.

Also, because of translational invariance[38], the sum of the gradients of four indices equals to 0:

$$\left(\frac{\partial}{\partial \mathrm{A}_i}+\frac{\partial}{\partial \mathrm{B}_i}+\frac{\partial}{\partial \mathrm{C}_i}+\frac{\partial}{\partial \mathrm{D}_i}\right)(\mathrm{ab}|\mathrm{cd}) = 0 \tag{20}$$

So, in the worst case, only three rather four centers must be evaluated, and thus, we can skip the one with the largest estimated computational resource requirement to optimize the calculation. For the most optimal case, the contraction step can be applied to equation (19) using

$$\frac{\partial}{\partial \mathrm{A}_i}(\mathrm{ab}|\mathrm{cd}) = \big((a+1_i)b\big|cd\big)_\alpha - a_i((a-1_i)b|cd) \tag{21}$$

where the subscript $\alpha$ in the first term of the RHS of above equation indicates that it has been formed with a contraction coefficients scaled by two. However, this strategy may be limited by the fact that the GPU may not have sufficient registers or/and memory to hold auxiliary integrals for high angular momentum ERIs. For equation 19, if three centers are evaluated simultaneously, only one set of temporary ERIs is needed, which is of [ab|cd] type, while equation 21 requires 4 sets ($(\mathrm{ab}|\mathrm{cd})_a$, $(\mathrm{ab}|\mathrm{cd})_b$, $(\mathrm{ab}|\mathrm{cd})_c$ and (ab|cd)). Therefore, efficiency has to be sacrificed in this situation, and we will return to this issue and discuss the details in the next section.

## 4.3. IMPLEMENTATION

### 4.3.1. CUDA

GPUs offer a tremendous amount of computing power in terms of FLOPS while being relatively efficient in terms of heat produced and overall energy costs. However, increased complexity and reduced flexibility are the shortcomings of GPUs when compared with traditional CPU platforms. A GPU is an example of a massively parallel stream-processing architecture using a single-instruction multiple data (SIMD) model. GPUs process threads in blocks, with 16 to 1024 threads per block, which a programmer can specify in their code and the GPU executes the threads in warps. In the current generation of GPUs the warp size is 32. Threads in one warp must execute the same instruction in the same clock cycle due to the fact each streaming multi-processor executes in a Single Instruction Multiple Thread (SIMT) fashion. Therefore, branching must be addressed to avoid instruction divergence, which significantly affects overall performance. These thread blocks logically map to streaming multiprocessors (SM) in the GPUs; for example, 16 SMs for a NVIDIA M2090 (Fermi architecture) and 15 SMs for a NVIDIA K40 (Kepler architecture). These specifications are useful for optimum performance for code vectorization. Moreover, the Fermi architecture has compute capability of CUDA 2.x, which features fast double-precision computing and atomic operations, while Kepler has CUDA 3.x, which features even faster double-precision computing, in-warp communication and dynamic parallelism.

The memory hierarchy of GPUs originates from its graphics lineage. Main memory, known as dynamic random-access memory (DRAM) or global memory, is visible to all threads in all multiprocessors, is relatively large (for example, 6GB for a M2090, 12 GB for a K40c) but comparatively slow such that that frequent data retrieval from DRAM should be avoided. Shared memory is accessible to all threads within a multiprocessor, but it is small (typically 48kB), but faster than DRAM, hence, it plays a critical role for thread communication or memory buffering when frequently fetching from DRAM is necessary. Besides this type of memory, all threads have a small amount of private and fast registers, and in the recent Kepler platform, threads can gain access to registers of other threads in a same warp and avoid thread communication via shared memory. To maximize the potential of GPU computing, some further suggestions include (1) coalesced memory to access DRAM is recommended, (2) the chance of thread divergence should be minimized utilizing shared memory to store intermediate results, and (3) threads can hide memory latency by overlapping computations. Moreover, the CPU and GPU are in different physical address spaces and DRAM is the only way to synchronize the data between the host (mostly CPU) and the attached devices (GPUs), but this comes with a significant performance penalty due to the slow PCIe bus. Thus, CPU and GPU communication should be avoid if not necessary. In the following sections, we will describe an implementation based on the GPU programming philosophy described above.

### 4.3.2. MACHINE GENERATED ERI CODE

The code to evaluate ERIs, especially with high-angular momentum values, is extremely complicated with many different scenarios and possible coding tricks for optimization, so it is very difficult or even impossible to write ERI code by hand efficiently. In this case automated code generation is almost a requirement[6,9]. Our major focus is on the VRR step, which is based on equation 17, while the HRR step, which is given in equation 16, is relatively easy to write because the branching condition only depends on one index although it requires different subroutines for ERI evaluation and ERI gradient evaluation.

First of all, meta-classes were written by a code-generator that describes how an ERI class is related to up to 5 other classes via the determination of the necessary coefficients (using equation 17). The class description is presented in Figure 4.1. The class, named $C_{ij}$ , or relation class, represents the relation between the ERI class $[i0|j0]^{(m)}$ and ERI classes with lower i and j values and higher m values with explicit i and j values but an implicit m value. The functions, $F_{ij}$, evaluates the integral $[i0|j0]^{(0)}$ starting from a set of starting integrals ranging from $[00|00]^{(0)}$ to $[00|00]^{(i+j)}$, and the [00|00] type integrals or $C_{00}$ class can be analytically evaluated from equation 11 (the $C_{00}$ class and the $F_{00}$ function are actually assigned values in our code). The path from starting integrals to integral $[i0|j0]^{(0)}$ can be easily patterned using a breath-first search by knowing the tree-like vertical recurrence relation as expressed in equation 17. Our code traces the required relation class $C_{ij}$ starting from $F_{ij}$ with a starting queue with $C_{ij}$ and an empty pool, then pop $C_{ij}$ into the queue and adding a dependent relation classes of $C_{ij}$ into the pool, after that, it repeatedly pushes all relation classes in the pool into the queue and pops them all while adding their dependent relation classes into the pool until only the $C_{00}$ classes remains in the pool. As noted, in class $C_{ij}$ of Figure 4.1, the j index is downgraded while the i value is constant except for $[(i-1)0|(j-1)0]^{(m+1)}$, however, we can also downgrade the i index and produce the same ERI value, so the final selection depends on which one has the fewest FLOPs and the same selection is made for $F_{ij}$. These selections are made empirically. We hide some parameters such as P, Q, W and Gaussian function information in Figure 4.1 for brevity.

The last, but quite important step, is the final optimization. We simplified the code by eliminating variables that are only used once by replacing the variables with their expressions and factoring out common sub-expression. For CUDA, eliminating variables may offer a greater benefit because it reduces register usage which otherwise could utilize slow global memory and jeopardize performance. Therefore, factorization may have a deleterious effect according to our tests.

Using the procedure outlined above we generate an efficient ERI evaluation code for [i0|j0] with i and j values up to 6 respectively. This is sufficient for [ff|ff] ERIs and [dd|dd] gradient ERIs. These subroutines are relatively complicated in terms of numbers of lines, for example, $F_{44}$, which calculates [h0|h0], has 344 lines and $F_{66}$ has 8568 lines, which illustrates the necessity of using machine-generated code.

**Class $C_{i,j}$**

**Member**: $A \times B$, where $\vec{a}$ in A and $\vec{b}$ in B satisfy

$\vec{a} = (a_x, a_y, a_z)$ that $a_x + a_y + a_z = i$

and $\vec{b} = (b_x, b_y, b_z)$ that $b_x + b_y + b_z = j$

**Constructor**:

Input parameters:

$C_{i,(j-1)}$ $(i, j-1)^{m+1}$ $C_{i,(j-1)}$ $(i, j-1)^{m}$ $C_{i,(j-2)}$ $(i, j-2)^{m+1}$

$C_{i,(j-2)}$ $(i, j-2)^{m}$ $C_{(i-1),(j-1)}$ $(i-1, j-1)^{m+1}$

$\zeta, \eta, \vec{P}, \vec{Q}$ and $\overrightarrow{W}$

Content: using input parameters to express each member of class $C_{i,j}$

**Function $F_{i,j}$**

**$C_{0,1}$ $(0,1)^0$ :** $C_{0,0}$ $(0,0)^0$ , $C_{0,0}$ $(0,0)^1$ // express $(0,1)^0$ constructs from $(0,0)^0$ , $(0,0)^1$

**$C_{0,1}$ $(0,1)^1$ :** $C_{0,0}$ $(0,0)^1$ , $C_{0,0}$ $(0,0)^2$ // $\zeta, \eta, \vec{P}, \vec{Q}$ and $\overrightarrow{W}$ should also include but not listed here

…

**Figure 4.1.** Class and Function description for the ERI code. $(i,j)^m$ represent an ERI class for $[i0|j0]^{(m)}$ and class $C_{i,j}$ is built to express its relation with other classes with lower i, j, same m and higher m values The goal of Function $F_{i,j}$ is to generate $[i0|j0]^{(0)}$ from $[00|00]^{(m)}$ where the m value varies from 0 to i+j by using classes to express its path.

### 4.3.3. ERI EVALUATION FOR LOW-ANGULAR MOMENTUM VALUES

It is relatively straightforward to evaluate ERIs with quantum numbers less than 2. Technically, $F_{00}$ to $F_{44}$ ($F_{00}$ is not a subroutine but a value) are all that is needed to calculate the most complex integral [dd|dd], and it is possible to combine all of these 24 subroutines to one large subroutine, and NVIDIA's CUDA Compiler (NVCC) can successfully compile and execute this subroutine. In last chapter[8], we showed the speedup and accuracy of this unoptimized subroutine, and below we illustrate the performance with optimized code. Moreover, this subroutine can calculate the gradient of s and p orbital integrals. We provide benchmark details below.

### 4.3.4. ERI EVALUATION FOR HIGH-ANGULAR MOMENTUM VALUES

For ERIs with higher-angular momentum, for example ERIs with f orbitals and ERI gradients of d orbitals, large numbers of registers are required and NVCC can neither compile nor execute if we include subroutines beyond $F_{44}$. However, since we have equation 16, which suggests ERIs can be expressed as a linear combination of [k0|l0] or (k0|l0), we can divide these combination into several parts, for example,

$$[a\mathrm{b}|cd]^{(m)} = \sum_{k=a}^{a+b} \sum_{l=c}^{c+d} c_{kl}[k0|l0]^{(m)} = \sum_{Z} \sum_{k,l\in Z} c_{kl}[k0|l0]^{(m)} \qquad (22)$$

Which implies the additivity of ERIs. The subscript Z indicates that these ERIs are only contributing to VRR ERIs within zone Z. For example, we can manually setup two zones for [a0|c0] that $Z_1$={a<3 and c<3}, $Z_2$={a >=3 or c >= 3}. Then, $Z_1$ includes s and p orbital only ERIs and $Z_2$ contains ERIs involved with d and higher angular momentum values. Moreover, it does not involve the exponent part so the contraction step can be applied.

$$(a(b+1_i)|cd)^{(m)} = \sum_{Z} \sum_{k,l\in Z} c_{kl}(k0|l0)^{(m)}$$

$$= \textstyle\sum_Z(((a+1_i)b|cd)^{(m)}{}_Z + (A_i - B_i)(ab|cd)^{(m)}{}_Z) \qquad (23)$$

The previous equations demonstrate that if we separate the VRR step into several parts and count contributions from each part rather than compute all VRR ERIs as a batch, we can still produce the same results without introducing extra FLOPs into the VRR and a just a few extra FLOPs into the HRR. In addition, the derivatives of ERIs can be computed in this fashion as well, which we will use later.

$$\frac{\partial}{\partial \mathrm{A}_i}[\mathrm{ab|cd}] = \textstyle\sum_Z(2\ \alpha[(a+1_i)b|cd]_Z - a_i[(a-1_i)b|cd]_Z) \qquad (24)$$

and if a contraction step is feasible on the GPU, then

$$\frac{\partial}{\partial \mathrm{A}_i}(\mathrm{ab|cd}) = \textstyle\sum_Z((\big((a+1_i)b\big|cd\big)_Z)_\alpha - a_i\big((a-1_i)b\big|cd\big)_Z) \qquad (25)$$

With these equations, we can further consider a partition scheme. For now, we only consider F00 to F66, which computes ERIs up to [fflff]. All of these 48 subroutines (again, F00 is a value rather than a subroutine) can be merged into one subroutine. However, this fails in the compilation stage as described previously. The basic principal of our partition scheme is to try to bind as many subroutines into one subroutine as possible. Therefore, we developed the partition design presented in Figure 4.2. This scheme is the best fit for device of CUDA compute capability 3.x (3.0 and above) and 2.x(2.0 and above but 3.0 below), within which, 9 and 5 subroutines or zones are separated from the global subroutine respectively, and functions that fall within the same zone will be warped into the same subroutine corresponding to one kernel during the

VRR step. Addition of other subroutines to a zone (combining zone 3 and zone 1 in Figure 4.2(a) for example) cannot compiled nor executed on current generation GPUs. These series of subroutines can be further optimized at the HRR step by pre-excluding select ERIs.

As we described in last chapter and, indeed, most GPU-based SCF implantations, a pre-sorting algorithm developed by Ufimtsev and Martinez[1,2] is used prior to ERI evaluation to ensure that threads can execute the same or similar instructions with their neighboring threads within a warp. We again employed pre-sorting in our implementation, and we only describe the outline of this treatment with the full details being given in our previous publication. Since it is impossible to sort or vectorize $N^4$ ERIs but it is possible to do so for $N^2$ half ERIs, a feasible strategy is to replace the four-index ERI with a $N^2$*$N^2$ matrix problem with two dimensions represented by a bra (such as [ij|) and ket (such as |kl]). And once we have this type of ERI matrix, we can “thread walk” - searching from one element of the matrix to another, calculating the assigned ERI and continue to work on the next one until all elements have been evaluated. A thread will evaluate a contracted ERI rather than a primitive ERI since it has been shown that this strategy is very effective[3]. Each dimension, bra or ket, of the ERI matrix can be rearrange by three kinds of criteria, ERI type is the criteria with highest priority, with the primitive Gaussian function number and Schwarz cutoff as the second and third criteria, respectively. To try to minimize thread divergence, the selection of these three criteria was based on 1) different ERI types call different ERI subroutines which will take the majority of the calculation time, and is the innermost loop in the ERI subroutine. 2) The primitive

Gaussian function number determines the number of loops calling the ERI subroutine, and is the second innermost loop. 3) Sorting by the Schwarz cutoff maximizes the possibility of an all-cutoff or all-pass scenario for the ERIs evaluated in a warp, which is the outermost loop**.** The Schwarz cutoff is a method to estimate the upper bound of an ERI value by using a Cauchy-Schwarz inequality[39]

$$[ab|cd] \leq \sqrt{[ab|ab][cd|cd]} \tag{26}$$

Herein, we improve this treatment in two ways: First, before sorting by ERI types, we split the bra or ket into two parts, one with a high Schwarz value (dense region, expecting large integral values) and another with a low value (sparse region, expecting small integral values). The cutoff can then be manually chosen, and we select a value of $10^{-4}$ because we found that it gives the best balance between efficiency and accuracy. For the dense region, the ERI values are typically too big to be ignored and the dominant factors are its ERI types and primitive Gaussian function numbers if a contraction step is feasible on the GPU. A hidden filter arises because of the symmetry of the ERI, [ij|kl]=[ji|kl]=[ij|lk] and so on, so only ERIs with i<=j and i<=k and k<=l need to be calculated. Therefore, if the Schwarz cutoff is an upper bound criteria, this does not impact the dense regions of the ERI matrix (see Figure 4.3). Via this, the order of the half matrix will be randomized resulting in instances where i>j which will be ignored by symmetry (i≤j are retained). If this conditional check is not made with the Schwartz cutoff divergence may occur. Without the Schwarz cutoff i≤j is always true which eliminates the need for the conditional check. So in the dense region, the Schwartz cutoff is not needed. On the other hand, for the sparse region ERIs will have a greater chance to be small enough to encounter the Schwarz cutoff so that once threads in one warp fall into this region, it is very likely that all threads in the same warp will skip the ERI evaluation. Hence, in the sparse region, after sorting by ERI types, sorting the half ERI by the Schwarz cutoff upper bound is an efficient strategy.

This strategy is illustrated in Figure 4.3, using an example of a water cluster with 4 molecules 6 Å from each other such that both the inter and intra molecular interactions are considerable but neither very strong nor very weak. This clearly splits the half ERIs into dense and sparse regions. This system includes 212 basis functions, 64 shells and 1187 eligible half ERIs. We used a color scheme to illustrate the magnitude of the ERI upper bound. It is comparatively easy to discern the dense*dense and sparse*sparse region and the other two areas (dense*sparse) in between in Figure 4.3. The sparse*sparse area, mostly colored with purple and black regions are crowded with ERIs with a small upper bound while the dense*dense area is filled with orange that represents ERIs expected to have large values. However, the boundary for ERI types **is** hard to tell for the dense area because the Schwarz cutoff is not introduced in that area, however, the boundary for ERIs in the sparse area can be easily identified. The table next to the ERI matrix in Figure 4.3 indicates the boundaries for the half ERIs.

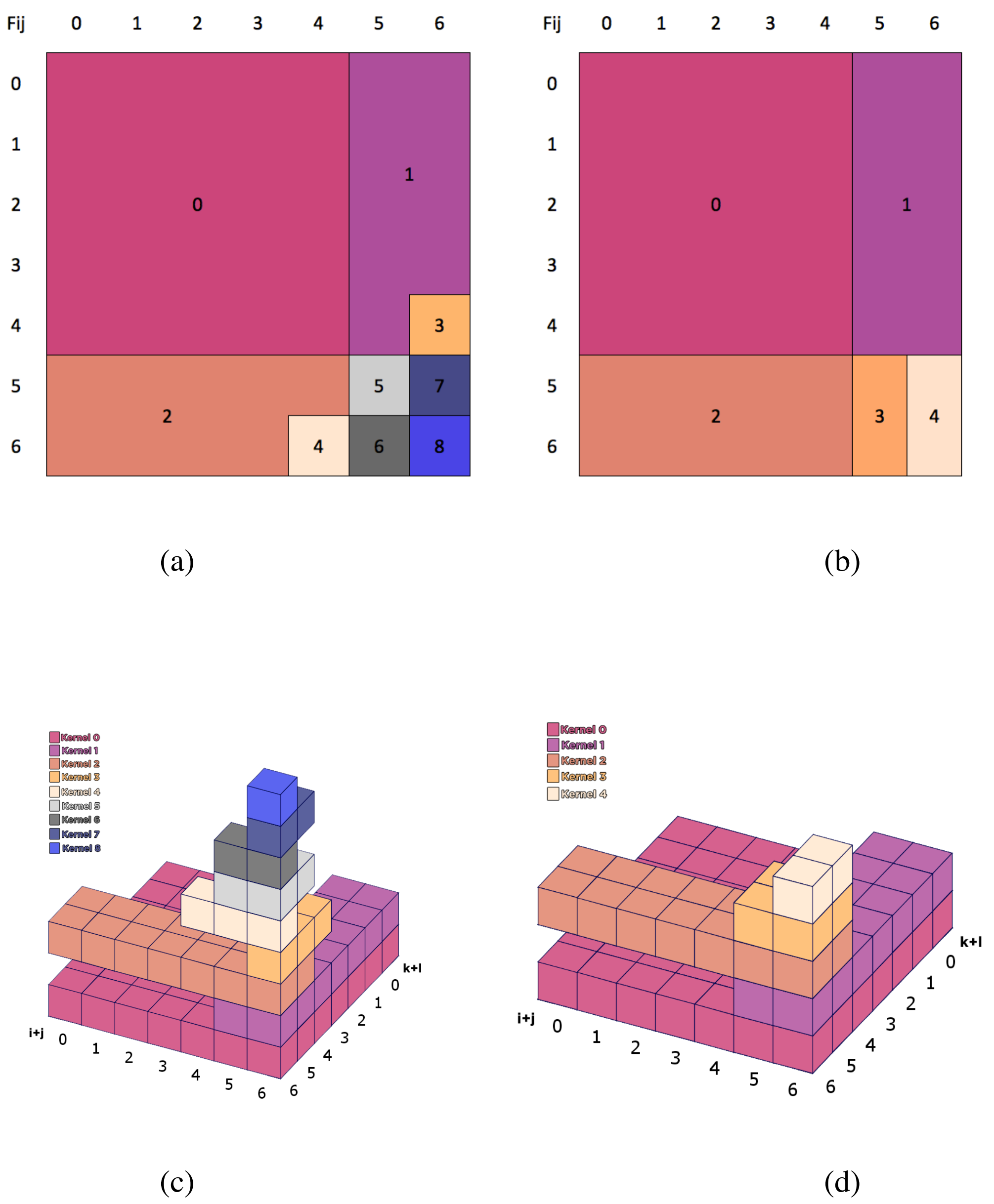
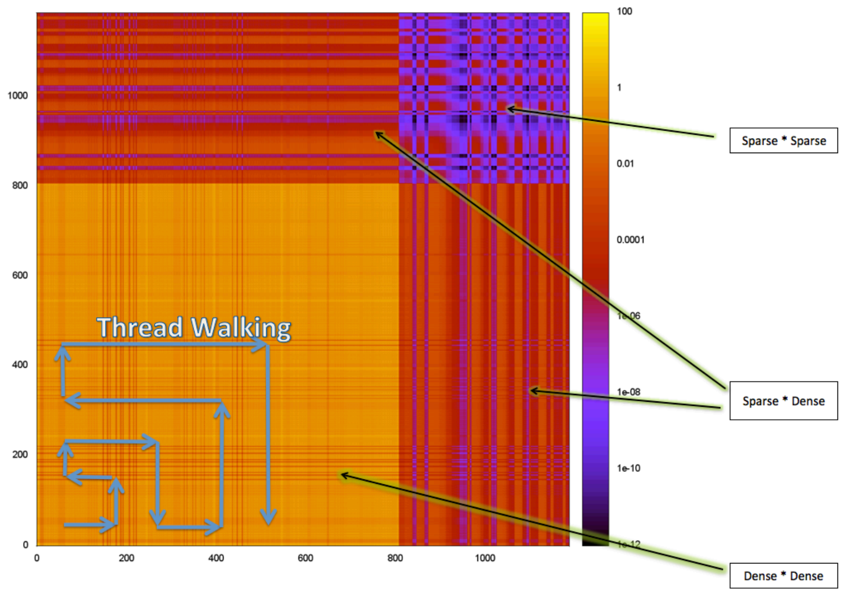


**Figure 4.2.** Optimized subroutine partition pattern for (a) CUDA 3.0 and (b) CUDA 2.0 architectures. All $F_{ij}$ functions that are within one zone will be warped in one subroutine and called in a kernel. (c) (d) Kernel call pattern for ERI type [ij|kl] with the partition strategy (a) for CUDA 3.0 and (b) for CUDA 2.0. See text for details.

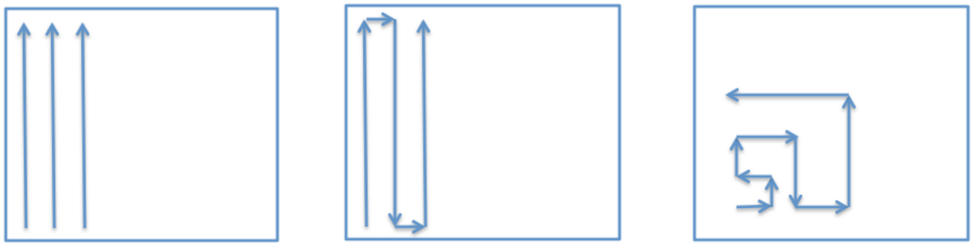


(Water)4 / 6-311G(df,pd)

| Half ERI | i+j | Starting | Ending |
|---|---|---|---|
| Dense | 0 | 0 | 243 |
|  | 1 | 244 | 467 |
|  | 2 | 468 | 647 |
|  | 3 | 648 | 747 |
|  | 4 | 748 | 791 |
|  | 5 | 792 | 803 |
|  | 6 | 804 | 808 |
| Sparse | 0 | 808 | 950 |
|  | 1 | 951 | 1064 |
|  | 2 | 1065 | 1145 |
|  | 3 | 1146 | 1181 |
|  | 4 | 1182 | 1187 |
|  | 5 | - | - |
|  | 6 | - | - |

**Figure 4.3**. Pre-sorting and thread walking illustration. The test case is a water cluster with 4 water molecules. The color scheme indicates the upper bound of ERIs, dense*dense, sparse*dense and sparse*sparse areas can be visually identify in the ERI matrix. The table on the right is the boundary for different ERI types. See details in the text.

Besides this change, a second change is the thread walk was altered from a linear search to a circular search as shown in Figure 4.3. This adaptation is especially suitable to the first change because the large-valued and small-valued regions are most likely distributed circularly at the origin or at the edge of the ERI matrix. So at the beginning, the thread walk ensures almost every ERI is not cutoff and at the last steps of the walk, almost every visited ERI will be smaller than the cutoff criteria. Both changes yield a ~30% improvement for small systems and ~15% for large systems (typically for system with more than 1000 basis functions) because of the smaller possibility of thread divergence. We compare other walk strategies, like linear and snake, on alanine chains with lengths of 1, 5, 9 and 13 using the 6-31g** basis set in Figure 4. The circular is faster for the smaller systems, but for larger all methods explored performed similarly. Other thread walk strategies might offer significant improvements as well but generally are not as good as the circular search and suffer more significant degradations when applied to large systems as well. Another note is that most SCF calculations will only evaluate the difference between iterations so in the final several iterations, the ERI matrix will be much more sparse than in the initial iterations. Thus, when applying the thread walking changes, the time needed to calculate later iterations will be slightly increased when compared with the original thread walking strategy. Nonetheless, the overall time required after these modifications is better than when these computational tricks are not employed.



(a) Linear (b) Snake (c) Circular

Performance comparison between different thread walk strategies

| Molecule Name(Atom Number) | α-helix acetyl(ala)$_{18}$NH$_2$ (189) | | β-strand acetyl(ala)18NH2 (189) | | Taxol (110) | | Valinomycin (168) | |
|---|---|---|---|---|---|---|---|---|
| Basis Set | 6-31g | 6-31g** | 6-31g | 6-31g** | 6-31g | 6-31g** | 6-31g | 6-31g** |
| (a) Linear | 144.7 | 435.1 | 64.7 | 203.4 | 49.3 | 149.9 | 106.8 | 312.1 |
| (b) Snake | 142.0 | 433.2 | 63.1 | 201.2 | 49.0 | 148.2 | 105.8 | 310.7 |
| (c) Circular | 135.2 | 424.1 | 57.0 | 191.8 | 45.2 | 145.2 | 98.7 | 301.4 |
| Improvement % | 6.6 | 2.5 | 11.9 | 5.7 | 8.2 | 3.1 | 7.6 | 3.4 |

**Figure 4.4.** Thread walk strategy comparison. Three different thread walk strategies are presented for GPU calculations. Time used to calculate the 2$^{nd}$ iteration for four examples of alanine chains of different lengths using 6-31g** are reported. Unit is seconds. Improvement % row indicates performance improvement of circular strategy over linear strategy.

We visualize the kernel call pattern in Figure 4.2(c) and (d) for an ERI type [ij|kl] with two axes with i+j value and j+k values, respectively. We show the partition strategies for CUDA capability of 3.x and 2.x in Figure 4.2(a) and (b), respectively. Each cube corresponds to a kernel run for this type. For example, if i+j equals 5 and k+l equals 6, kernel 0,1,2,4,5 and 7 are necessary to compute its value. From a kernel perspective, a kernel does not contribute to every ERI type but only for certain types. For example, for subroutines in zone 1 warped as kernel 1 in Figure 4.2(a)  (designed for the device of CUDA compute capability 3.x) the kernel containing subroutines $F_{(5\sim6)(0\sim6)}$, is needed by ERI types [ij|kl] that fulfill the condition $i + j \geq 5$, and similarly, for zone 3, threads only need to search $i + j \geq 5$ and $k + l \geq 5$. Therefore, only going through those regions that may produce these types of ERIs can minimize redundant thread walks and further improve performance. These calls are statically determined at the compilation stage to avoid unnecessary branching during runtime.

### 4.3.5. IMPLEMENTATION OF ERI DERIVATIVE EVALUATION

Besides the single point energy, the Hartree-Fock method can provide analytical expression for the energy gradient. Similar to the single point energy, the bottleneck is, as shown in equation 3, the evaluation of the ERI derivatives, which accounts for more than 80%~90% of the CPU depending upon system size. To implement the HF gradient calculation on GPUs, two general steps are necessary: the ERI derivative evaluation and final gradient assembly.

For ERI derivative evaluation, similar to normal ERI evaluation, a recurrence relation is used in our implementation. The difference between normal ERI evaluation and the derivative evaluation, as discussed above (see equation 19), is the presence of two extra classes of ERIs one with higher index and another with a lower one. If, for example, we want the d orbital gradient, the derivative of [dd|dd] requires a series of subroutines $F_{(0\sim5)(0\sim5)}$ but $F_{55}$ is not needed. However, as described in the last section, the compiler cannot handle all subroutines at once, therefore, a similar, but simpler, partition strategy was developed as shown in Figure 4.5, which represents a subset of the normal ERI evaluation process. As Figure 4.5 shows, three zones are indicated, and this partition is suitable for both the CUDA compute capability 2.x and 3.x architectures. The kernel call pattern is quite similar to that of the CUDA compute capability 2.x ERI evaluation but simpler so we do not present it herein. The thread walk is similar to that used in the normal ERI evaluation. Moreover, the HRR step for zone 1 and 2 are simpler than zone 0. For zone 0, the most efficient algorithm is to evaluate twelve derivatives classes (four centers and three directions, but nine classes need to be computed (see equation 20) simultaneously once the value of the auxiliary ERIs from [00|00] to [(i+j+1)0|(k+l)0] and [(i+j)0|(k+l+1)0] are obtained. For the derivative of a primitive ERI, the HRR is applied as much as 18 times because there are 9 classes and each class has two ERIs (a higher and lower function). However, if the contraction step is possible on the GPU significant computational savings can be achieved. Zone 1, for example, is called when (1) (i+j) equals 4, (2) for the derivative of center i or j and (3) only when an ERI with a higher function is involved, hence, only 6 and 3 HRR computes are required at most for zone 1 and zone 2.

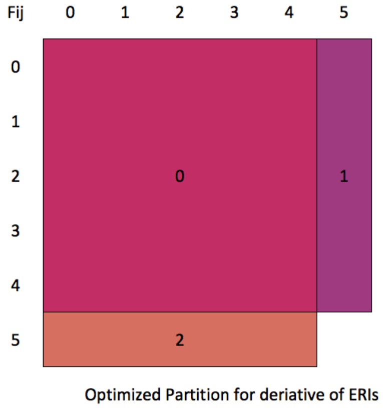


**Figure 4.5**. Optimized Subroutine Partition for ERI Derivatives. This partition strategy works for both CUDA 3.0 and CUDA 2.0. The kernel call mapping is similar to Figure 4.2 but simplified so it is not shown here.

However, the contraction step, sacrificing space usage for reduced FLOP counts, given by equation 25 cannot be applied to zones 1 and 2 which are wrapped as two separate kernels. This is caused by insufficient register and memory resources available on the GPU. If the contraction step is applied, as given in equation 21, an efficient way to implement this is to store 4 sets of auxiliary classes whose class size increases exponentially with the growth in quantum number: For example, because the degeneracy number of s,p,d,f,g,h,i orbitals are 1,3,6,10,15,21,28, so for [ddl, L=4, we have 1+3+6+10+15 = 35, therefore, [ddldd] requires 35×35 for each class and 56×56 for the derivative; [fflff] requires 84×84 if each thread computes one contracted ERI, which is infeasible for ERIs with higher-angular momentum functions even though one set is possible as discussed in the last section regarding [ff|ff] evaluation. So for zone 1 and zone 2, the contraction step is skipped because it is computationally infeasible. In this way CUDA executes the subroutines and each thread will work on a primitive ERI instead of a contracted one. We show below that with or without the contraction step we can achieve impressive performance.

Another stage is assembling the gradient. In the HF energy implementation, we used atomic operations, a set of lock-set-unlock operations to void thread conflicts and assemble the Fock matrix in global memory. However, the drawback of atomic operations in our hands was their speed and accuracy. The clock cycle of atomic operations was generally two-fold less than that of registers and one-fold with respect to shared memory, even though this penalty could be reduced by coalescing memory access and thread access.

Atomic operations are not particularly fast on Fermi GPUs but for Kepler GPUs they are ~3x faster than the previous generation of cards. Therefore, we expect a similar accuracy using atomic operations for the SCF energy and gradient but with increased performance from the Kepler GPUs. Compared to the HF energy calculation, the number of atomic operations is significantly less for the HF gradient. This is because for the HF energy, an ERI class contains many individual ERIs (*e.g.,* (pp|pp) contains 81 ERIs) and each ERI contributes to up to six elements to the density matrix via atomic operations (*e.g.,* the (pp|pp) class, in the worst case scenario, requests up to 486 atomic operator calls). But for assembling the gradient, an ERI class shares common centers which means they contribute to the same gradient elements, thus, the contribution from the ERI derivative can be assembled locally at the thread level. The gradient is then updated using a single atomic operator call regardless if the contraction step is introduced, so a thread calls at most 12 atomic operators at once for zone 0, 1 and 2. We use a presorting step for the energy computation and reutilize this for the gradient computation. We could have created a pre-sorting strategy for the gradient computation as well, but this added overhead would eliminate the realized computational gains.

## 4.4. BENCHMARK RESULTS AND DISCUSSION

We implemented and benchmarked our ERI and ERI derivative code in our quantum chemistry package QUICK[40] for both CPUs and GPUs. The QUICK CPU code was originally written in FORTRAN and the GPU code is rewritten in C++ together with the machine-generated code described above. We built the CPU version with the INTEL FORTRAN Compiler (version 12.1.5 20120612) with optimization option level 3 (-O3). For the GPU version, the Intel C++ Compiler (version 12.1.5 20120612) and CUDA compiler 4.2 V0.2.1221 with optimization option level 3(-O3) was used. Moreover, the fast math library option was used (-use_fast_math) and all calculations used double precision unless otherwise indicated. In our benchmarks, the CPU we used was a single core 3.07GHz Intel Xeon CPU X5675. The GPU test platform was a NVIDIA TESLA K40. This GPU card features 12 Gb of global memory, 15 streaming multiprocessors and 2880 CUDA cores built based on the KEPLER architecture. In addition, we turned the

ECC off and clocked to 875MHz to maximize the performance. The K40 card has compute capability 3.5, so the kernel partition and kernel call pattern will follow Figures 2(a) and (c).

First, we present the tests we carried out on SCF energy calculations involving f orbital ERI contributions. In the SCF energy GPU implementation, all primitive ERIs were calculated on-the-fly and the Fock matrix was assembled in global memory, while for the CPU version, the primitive ERIs can be saved in memory to form contracted ERIs for higher efficiency which cannot be done on GPUs due to a shortage of register space. Both the CPU and GPU versions used the direct SCF procedure, which has been shown[1,2,7,8] to be suitable for GPU-based SCF calculations. Any integrals smaller than $10^{-9}$ were neglected both in the CPU and GPU benchmark studies. ERI evaluation with f orbital contributions on GPUs will be treated with several kernel runs as discussed above, while for the CPU we ran in a traditional manner.

We studied linear alanine chains with lengths ranging from 1 to 26 with the 6-31G(df, pd) basis set (with 280 to 4702 basis functions). The results are summarized in Figure 4.6. We plotted the time used for the second iteration to form the Fock Matrix with the superposition of atomic densities (SAD) initial guess.[41,42] We exclude the one-electron contribution in the plot, which is relatively small and does not need to be calculated every iteration. Besides the time to form the Fock Matrix, diagnalization is another critical step for each iteration, and while it is relatively small for small systems it will ultimately dominate for very large systems because it scales as $O(N^3)$ which is larger than the scaling dependence of direct SCF calculations. We use the second iteration for our benchmark studies because comparing with first iteration timings can lead to misstating the overall GPU performance. Meanwhile, besides the time to run a single iteration, the time cost associated with different kernels is plotted as well. As observed in Figure 4.6, a GPU yields an 8-18 times speedup when compared to a single core of a CPU with the speedup increasing with an increase in system size. This tendency was observed in last chapter[8] and in the work of other groups[9]. This tendency is due to three reasons: (a) larger systems means larger ERI matrices that lead to less thread divergence; (b) larger systems produce more ERIs that are small enough to be cutoff and (c) reduction in chance collisions of thread requests to the same density matrix element in larger systems, which reduces the atomic operation penalty. The most time consuming kernel is kernel 0 (see Figure 4.6), which is not surprising since every ERI including ERIs with f-type Gaussian function contributions calls this kernel. Besides kernel 0, kernels 1-8 are called when the ERI has an f Gaussian function contribution with the accumulated time taking roughly 30% of the total time.

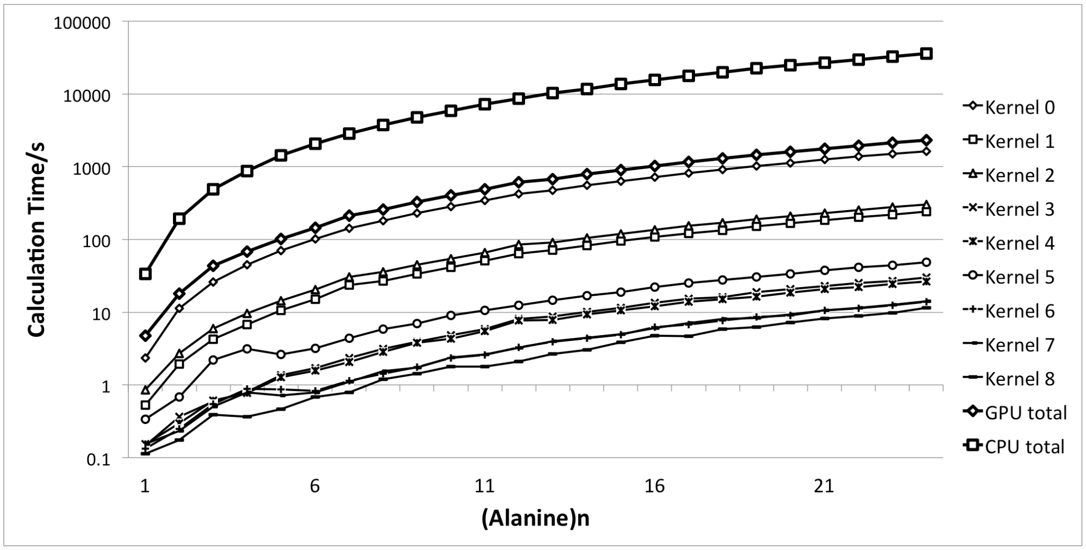


**Figure 4.6**. Calculation Time profiles for the CPU and GPU calculations on alanine chains. Timings for the CPU and GPU kernel time (top two curves) are based on the Fock matrix formation excluding the one-electron contribution. Timings for kernel 0 to kernel 8 are given as well and the kernel call mapping corresponds to Figure 4.2. Platform and software details are described in the text. A logarithmic scale is used for the Y axis.

The SCF energy error between the CPU and GPU calculation using double-precision is within $10^{-7}$ a.u. for the alanine chain test set which is quite accurate given the different execution order of the floating point operations for the GPU and CPU implementations. To further analyze our implementation, we examined several systems with different basis sets with (6-31G(df,pd) and 6-31G(2df,2pd)) and without f orbital contributions (6-

31G**). The systems we tested include relatively small molecules such as taxol, valinomycin, and small proteins containing around 200 atoms including the PDB[43] ID's of 1AKG, 1CNL, 1M2C and 1PEN. The results are presented in Table 4.1. The number of basis functions varies from more than a thousand (taxol at the 6-31G** level) to more than five thousand (1M2C at the 6-31G(2df, 2pd) level). The realized speedup is11-13 times and varies slightly with the basis set employed. The listed times used for kernel 1-8 accounts for approximately 9.6%, 12.0%, 1.2%, 1.1%, 1.7%, 0.6%, 0.6%, and 1.0% of the total time and ~27% overall. According to Table 4.1, the performance of f orbital computing with a GPU is not significantly impacted by our partition treatment. Actually, 6-31G(df,pd) and 6-31G(2df,2pd) calculations benefit from GPU usage because of the "large basis set effect" described above and along with our partition treatment (which goes against some conventional thinking on this topic[44]). The partition scheme might also be applied to kernel 0, where we separate kernel 0 into several smaller kernels, to potentially gain better performance. However, we found this idea did not work because these subroutines are relatively small so the overhead penalty of GPU kernel initialization, repeated ERI screening calculations and extra calculations to select the corresponding subroutine reduces the benefit brought about by our partitioning scheme. In sum, to evaluate ERIs with f Gaussian functions, we need slightly more time for the calculations, but the overall speedup, even considering the extra time used, is still impressive, demonstrating the accuracy, feasibility and efficiency of our treatment of ERIs with contributions from f functions.

The next benchmark is focused on SCF gradient calculation using the same test systems (polyalanines) as before. The GPU is used only to accelerate the most time-consuming part, which is the derivative of 4-center ERIs while other contributions such as the one-electron integral derivatives are calculated on the CPU since they only take a minor portion of the overall computation time. In this test, for both the CPU and GPU gradient calculations, the energy calculation itself was performed on the GPU to ensure we are using the same density matrix. The speedup of the GPU over the CPU calculation is illustrated in Figure 4.7(a) by comparing the time used to evaluate the derivative ERI part of the gradient calculation. As Figure 4.7(b) indicates, the accuracy of the GPU gradient compared to the CPU as indicated by the root mean squared deviation (RMSD) of calculated gradients and the max difference (bolded) is quite good. The first 6 points for 6-31G are not presented because the difference is smaller than $10^{-12}$. In terms of accuracy, the RMSD of the gradient difference between the CPU and GPU is less than $10^{-10}$ and $10^{-11}$ a.u. for the 6-31G** and 6-31G basis sets while the maximum difference is within $10^{-9}$ and $10^{-10}$ a.u. respectively. According to Figure 4.7(a), we find that our GPU implementation speeds up the gradient calculation by as much as 19 times ((alanine)$_{30}$ with the 6-31G** basis set and 3010 basis functions). When compared to the GPU speedup of the SCF energy, the observed speedup here is slightly better due to the reduced number of atomic operations.

To further analyze the differences in kernel time usage, we used the same set of systems used in the analysis of the energy calculation. The results are summarized in Table 4.2. Generally, the extra calculation (kernel 1 and kernel2) arising from the partition strategy represents about 12% of the total time (previously it was ~30% on average). Overall the speedup of the gradient calculation on the GPU relative to the CPU was between 12.9-16.1 times, which, as expected due to the reduction in atomic operations, is slightly better than what we observed for the realized speedups in the HF energy calculation on a GPU.

One important application of the energy gradients is geometry optimization. To test our implementation we optimized the $(\text{glycine})_{12}$ chain with the 6-31G and 6-31G** basis set using the L-BFGS algorithm and profiled the performance and measured the accuracy of the geometries obtained using the CPU or GPU code base. These calculations involve 87 atoms and 514 and 925 basis functions, respectively. The starting geometry was initially optimized with the STO-3G basis set using the GPU code. In order to compare the accuracy of the GPU gradient along with the GPU energy, three sets of computations were carried out. These include CPU (energy) + CPU (gradient), GPU (energy) + CPU (gradient), and GPU (energy) + GPU (gradient). We present the geometry RMSDs of the GPU+CPU and GPU+GPU relative to the CPU+CPU result from the same step in Figure 4.8. As expected the error observed using the GPU energy and gradient calculation increases as the number of steps increases. The error in the 6-31G** result is slightly larger than that of 6-31G due to the larger number of basis functions employed. At geometry convergence, the RMSD is mostly within the $10^{-4}$ magnitude for both the GPU+CPU and GPU+GPU calculations with 6-31G and 6-31G** basis sets. The energy differences obtained for the GPU+CPU and GPU+GPU optimizations relative to the CPU+CPU result (in atomic units) are $1.80\times10^{-6}$ and $3.21\times10^{-6}$ for 6-31G and $7.46 \times10^{-6}$ and $4.28\times10^{-7}$ for 6-31G**.

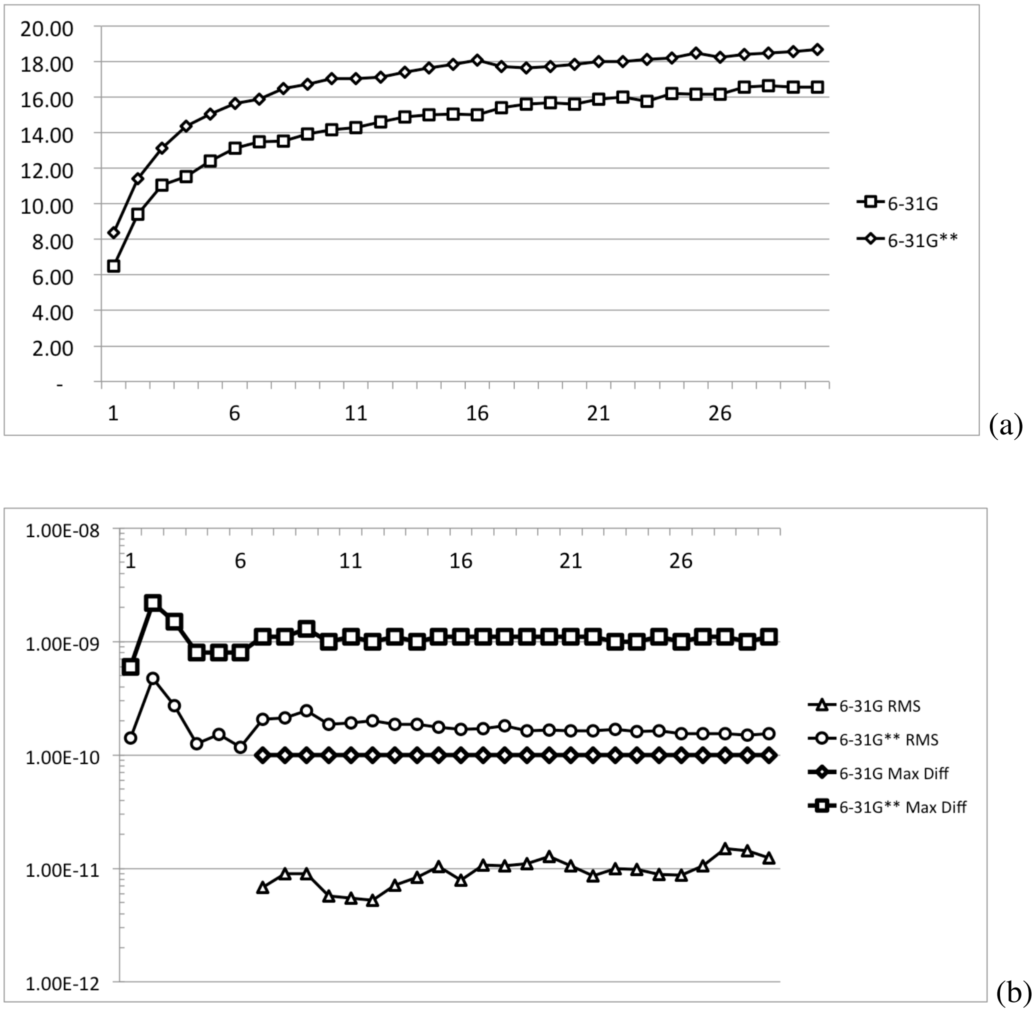


**Figure 4.7**. (a) Speedup comparison and (b) gradient deviation comparison between CPU and GPU calculations with different basis sets for alanine chain lengths from 1 to 30. Timing for CPU and GPU are based on the gradient generation time including data transfer but excludes the one-electron integral contribution. Platform and software details are described in text. For (b) RMSD and maximum difference (bolded) between the GPU and CPU result are presented. First 6 points are not presented because the difference is smaller than $10^{-12}$. A logarithmic scale is used for the Y-axis.

We also explored the memory usage of the GPU calculations. ERI evaluation with high-angular momentum functions does not allocate extra memory, and therefore, the peak memory usage is the same as in our earlier efforts[8]. For the gradient calculation on the GPU, a relatively small amount of memory is required to store the gradient value and it scales linearly with the number of atoms. For the K40 card with 12 gigabytes, for example, calculation of the (alanine)$_{28}$ chain using the 6-311G(2df, 2pd) basis set with 6780 basis functions uses almost every bit of global memory, which is one of the largest systems that the K40 is able to handle in our hands. Hence, with Quick, GPU global memory is the limiting factor for larger systems currently. This limit, together with bandwidth and the number of registers that limit calculation speed and size will be mitigated with incoming GPU technologies such as NVLink which will boost bandwidth and available memory.

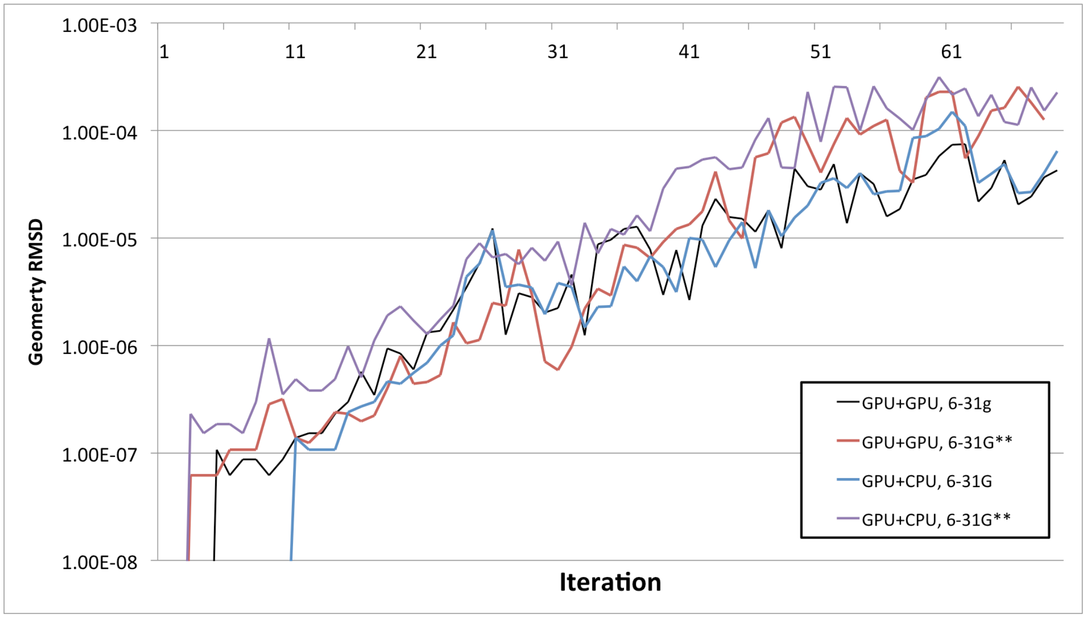


**Figure 4.8**. Accuracy of geometry optimization using a GPU *versus* a CPU. The test molecule was (Glycine)$_{12}$ with the 6-31G and 6-31G** basis sets. Three sets of computations were executed including CPU+CPU, GPU+CPU, GPU+GPU. The curves indicate geometry RMSD relative to the CPU+CPU results at the same step.

### Table 4.1. Performance Comparison between CPU and GPU SCF Calculation[a,b]

| Molecule/ Atom Number | Basis Sets/ No. of Basis Function | Kernel Time/s | | | | | | | | | GPU total/s | CPU total/s | Speedu p |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | | | |
| taxol (110) | 6-31G**/1160 | 132.11 | - | - | - | - | - | - | - | - | 132.11 | 1735.13 | **13.1** |
| | 6-31G(df,dp)/2064 | 607.57 | 86.23 | 114.41 | 10.79 | 10.63 | 17.22 | 4.78 | 5.25 | 4.68 | 861.55 | 13178.28 | **15.3** |
| | 6-31G(2df,2dp)/2577 | 1488.60 | 221.32 | 283.74 | 20.05 | 20.22 | 38.49 | 8.95 | 9.28 | 9.05 | 2099.70 | 28856.43 | **13.8** |
| valinomyc in (168) | 6-31G**/1620 | 276.29 | - | - | - | - | - | - | - | - | 276.29 | 3397.98 | **12.3** |
| | 6-31G(df,dp)/2940 | 1206.36 | 158.77 | 205.08 | 21.29 | 19.24 | 29.98 | 10.56 | 11.00 | 16.77 | 1679.05 | 24905.01 | **14.9** |
| | 6-31G(2df,2dp)/3678 | 3014.18 | 411.14 | 513.33 | 41.63 | 37.88 | 68.08 | 21.46 | 21.73 | 36.03 | 4165.46 | 56135.19 | **13.4** |
| 1AKG (209) | 6-31G**/2171 | 616.04 | - | - | - | - | - | - | - | - | 616.04 | 7290.80 | **11.8** |
| | 6-31G(df,dp)/3839 | 2874.78 | 370.15 | 469.47 | 53.39 | 47.25 | 68.28 | 28.20 | 28.47 | 47.67 | 3987.67 | 49457.74 | **12.4** |
| | 6-31G(2df,2dp)/4829 | 6220.36 | 841.31 | 1055.04 | 90.57 | 81.01 | 141.26 | 48.86 | 48.28 | 86.58 | 8613.27 | 107771.61 | **12.5** |
| 1CNL (169) | 6-31G**/1771 | 424.36 | - | - | - | - | - | - | - | - | 424.36 | 5054.76 | **11.9** |
| | 6-31G(df,dp)/3149 | 2033.50 | 263.78 | 335.74 | 35.90 | 32.47 | 48.86 | 17.67 | 18.57 | 26.28 | 2812.77 | 34725.38 | **12.4** |
| | 6-31G(2df,2dp)/3929 | 4405.82 | 597.59 | 743.98 | 62.04 | 56.40 | 97.97 | 29.60 | 30.72 | 49.02 | 6073.13 | 75738.14 | **12.5** |
| 1M2C (220) | 6-31G**/2276 | 704.04 | - | - | - | - | - | - | - | - | 704.04 | 7969.27 | **11.3** |
| | 6-31G(df,dp)/4060 | 3245.16 | 410.62 | 511.75 | 61.66 | 52.59 | 76.42 | 33.30 | 34.26 | 60.80 | 4486.57 | 53311.56 | **11.9** |
| | 6-31G(2df,2dp)/5068 | 7055.35 | 937.31 | 1158.67 | 107.66 | 95.43 | 156.81 | 57.60 | 57.60 | 110.93 | 9737.35 | 119672.64 | **12.3** |
| 1PEN (203) | 6-31G**/2131 | 607.46 | - | - | - | - | - | - | - | - | 607.46 | 7537.26 | **12.4** |
| | 6-31G(df,dp)/3789 | 2812.73 | 365.32 | 464.97 | 54.55 | 48.75 | 69.25 | 28.03 | 28.10 | 46.13 | 3917.83 | 48023.12 | **12.4** |
| | 6-31G(2df,2dp)/4728 | 6101.39 | 829.10 | 1034.71 | 88.97 | 80.81 | 138.65 | 46.94 | 46.85 | 82.34 | 8449.76 | 106282.32 | **12.6** |

a. Time listed in the table indicates the time used to form the Fock Matrix in the 2nd iteration of the SCF calculation excluding the one-electron contribution and the diagonalization time.

b. Platform and software details are described in the text.

**Table 4.2. Performance Comparison between CPU and GPU HF-Gradient Calculation** [a,b]

| Molecule/ Atom Number | Basis Sets/ No. of Basis Function | Kernel Time/s | | | GPU total/s | CPU total/s | Speedup |
|---|---|---|---|---|---|---|---|
| | | 0 | 1 | 2 | | | |
| taxol | 6-31G/647 | 98.00 | - | - | 98.00 | 1345.82 | **13.7** |
| (110) | 6-31G**/1160 | 379.70 | 18.65 | 33.78 | 432.13 | 6939.49 | **16.1** |
| valinomycin | 6-31G/882 | 183.47 | - | - | 183.47 | 2584.33 | **14.1** |
| (168) | 6-31G**/1620 | 691.03 | 35.75 | 57.62 | 784.40 | 12538.55 | **16.0** |
| 1AKG | 6-31G/1211 | 444.66 | - | - | 444.66 | 5729.57 | **12.9** |
| (209) | 6-31G**/2171 | 1522.28 | 75.22 | 132.96 | 1730.46 | 25422.03 | **14.7** |
| 1CNL | 6-31G/991 | 303.96 | - | - | 303.96 | 3914.50 | **12.9** |
| (169) | 6-31G**/1771 | 1078.59 | 49.77 | 90.32 | 1218.67 | 17861.00 | **14.7** |
| 1M2C | 6-31G/1268 | 477.74 | - | - | 477.74 | 6238.65 | **13.1** |
| (220) | 6-31G**/2276 | 1677.16 | 86.20 | 146.89 | 1910.26 | 27997.59 | **14.7** |
| 1PEN | 6-31G/1192 | 423.68 | - | - | 423.68 | 5616.09 | **13.3** |
| (203) | 6-31G**/2131 | 1500.19 | 74.27 | 132.99 | 1707.45 | 25548.42 | **15.0** |

a. Time listed in the table indicates the time used to calculate the gradient using the HF method excluding the one-electron contribution

b. Platform and software details are described in the text.

## 4.5. CONCLUSIONS

In this chapter, we implemented the evaluation of ERIs up to f orbitals and ERI derivatives up to d orbitals using GPU technology and ERI recurrence relations. Our SCF and gradient calculations demonstrate the efficiency of GPUs where speedups of 10~20 times faster are expected over modern CPUs. A partition strategy is introduced to solve the difficulties encountered in computing ERIs including f orbitals and was further applied to d orbital gradient calculation. Importantly, we observed a very limited efficiency decrease after employing this strategy. A well-sorted pre-sorting strategy and several other improvements boost overall GPU performance and atomic operations are used to reduce data transfer which is particularly effective on a Kepler GPU. Moreover, like other GPU-enabled quantum chemistry software[4,14], GPU-based DFT calculation is also available in QUICK although we did not touch on this topic. For pure DFT methods, such as BLYP and LDA, the difficulty of GPU coding is significantly easier than encountered in HF while the speedup that a GPU can bring is at the same level seen for HF. Now with the help of ERI acceleration, hybrid DFT methods, such as B3LYP, is available by simply calculating the HF and BLYP contributions to exchange-correlation part, respectively, using the GPU.

Even with the observed performance increase using GPUs, there is still room to improve. Our long-term goal is to efficiently and accurately investigate complex biological systems with GPUs and most calculations will involve s, p and d orbitals and f orbitals when large basis sets or calculations involving metal ions are employed. Hence, f orbital gradients is currently the next piece of the puzzle. However, even though we have generated the $F_{(0\sim7)(0\sim7)}$ code with an automatic code generator, which is sufficient for the f orbital gradient calculation, the difficulty is that even if the partition strategy is used as well, a kernel that only calls the subroutine $F_{76}$, for example, will fail at the compilation stage because the memory requirements are beyond the current generation of GPU cards. We developed a compromise approach to deal with this dilemma that split a given subroutine into several kernels but find that the overall performance is quite poor. So we currently run this part of the calculation on the CPU. However, GPU cards and software continues to improve so solutions to this problem are likely to be available in the near future.

Our future work will focus on following aspects. First we are integrating our source code with the AMBER MD package to further enable *ab initio* QM/MM simulations[15]. Second is to develop GPU enabled correlated *ab initio* methods such as MP2 and coupled-cluster methods. Moreover, GPU enabled second order derivatives would also be helpful for scientists who want to calculate frequencies or other properties. Additionally, a multi-GPU implementation and implementation of the code on INTEL PHI platform is on going in order to have another option to accelerate *ab initio* calculation[1,9].

# APPENDIX

# SAMPLE OF MACHINE GENERATED CODE

The goal of the machine-generated code is to generate a function such as

```
        void h_2_2(QUICKDouble* store, arguments)
```

where *store* is an double-precision float(type name QUICKDouble, a redefined double type) array that saved ERIs. The purpose of above function is to evaluate (20|20) or (ds|ds) integrals and save them to the ERI container *store*. *Arguments* are some constant parameters that describes molecular and basis sets information.

Within *h_2_2*,

```
__device__ __inline__ void h_2_2(QUICKDouble* store,arguments)
{
        // call for L = 0 B = 1
        f_0_1_t f_0_1_0 ( VY(0), VY(1), arguments);
        // call for L = 0 B = 1
        f_0_1_t f_0_1_1 ( VY(1), VY(2), arguments);
        // call for L = 0 B = 2
        f_0_2_t f_0_2_0 ( f_0_1_0,f_0_1_1, VY(0), VY(1), arguments);
        // call for L = 0 B = 1
        f_0_1_t f_0_1_2 ( VY(2), VY(3), arguments);
        ….
         // call for L = 2 B = 2
        f_2_2_t f_2_2_0 ( f_1_2_0,f_1_2_1,f_0_2_0,f_0_2_1,f_1_1_1, arguments);
        // WRITE LAST FOR I = 2 J= 2

        LOC2(store, 4, 4, STOREDIM, STOREDIM) += f_2_2_0.x_4_4 ;

        LOC2(store, 4, 5, STOREDIM, STOREDIM) += f_2_2_0.x_4_5 ;

        …

}
```

Here, *VY(i)* stands for F00 classes with m value equals to i because they are a set of value rather than classes. *f_[i]_[j]_t* class is a ERI (i0|j0) type with each member of the class is one combination of the ERI, for example,

```
// Class for L = 0 B = 2

class f_0_2_t {    // (ss|sd) ERI type

public: QUICKDouble x_0_4 ;         // (s=0, s = 0, s=0, d = xx)

        QUICKDouble x_0_5 ;         // (s=0, s = 0, s=0, d = yy)

        QUICKDouble x_0_6 ;         // (s=0, s = 0, s=0, d = zz)

        QUICKDouble x_0_7 ;         // (s=0, s = 0, s=0, d = xy)

        QUICKDouble x_0_8 ;         // (s=0, s = 0, s=0, d = xz)

        QUICKDouble x_0_9 ;         // (s=0, s = 0, s=0, d = yz)

        __device__ __inline__ f_0_2_t( f_0_1_t t_0_1_0, f_0_1_t t_0_1_1,

QUICKDouble t_0_0_0, QUICKDouble t_0_0_1, arguments);

};
```

Size of this class is enormous for ERIs with higher angular momentum value. And the inline function, expresses the relation between one ERI type and ERIs with lower angular momentum value. For example,

```
__device__ __inline__ f_0_2_t :: f_0_2_t ( f_0_1_t t_0_1_0, f_0_1_t t_0_1_1,
QUICKDouble t_0_0_0, QUICKDouble t_0_0_1, arguments)
{
        x_0_4 = Qtempx * t_0_1_0.x_0_2 + WQtempx * t_0_1_1.x_0_2 ;
        x_0_5 = Qtempy * t_0_1_0.x_0_3 + WQtempy * t_0_1_1.x_0_3 ;
        x_0_6 = Qtempx * t_0_1_0.x_0_3 + WQtempx * t_0_1_1.x_0_3 ;
        x_0_7 = Qtempx * t_0_1_0.x_0_1 + WQtempx * t_0_1_1.x_0_1 + CDtemp * (
t_0_0_0 - ABcom * t_0_0_1 ) ;
        x_0_8 = Qtempy * t_0_1_0.x_0_2 + WQtempy * t_0_1_1.x_0_2 + CDtemp * (
t_0_0_0 - ABcom * t_0_0_1 ) ;
        x_0_9 = Qtempz * t_0_1_0.x_0_3 + WQtempz * t_0_1_1.x_0_3 + CDtemp * (
t_0_0_0 - ABcom * t_0_0_1 ) ;
}
```

In sum, function h_[i]_[j] is to calculate ERI type (i0|j0) by the assistant of sets of f_[i]_[j]_t classes to evaluate the value from VY(i), a set of F00 or $(00|00)^{(m)}$ values. The code is extremely long and complicated, for example, 788 lines for f_6_6_t, and 124 f_[i]_[j]_t is constructed and computed as auxiliary ERIs for h_6_6 to calculate (60|60) ERI type containing 28*28 = 784 elements.

# REFERENCES

# REFERENCES

(1) Ufimtsev, I. S.; Martinez, T. J. *J. Chem. Theory Comput.* **2009**, *5*, 1004-1015.

(2) Ufimtsev, I. S.; Martinez, T. J. *J. Chem. Theory Comput.* **2009**, *5*, 2619-2628.

(3) Ufimtsev, I. S.; Martinez, T. J. *J. Chem. Theory Comput.* **2008**, *4*, 222-231.

(4) Yasuda, K. *J. Chem. Theory Comput.* **2008**, *4*, 1230-1236.

(5) Yasuda, K. *J. Comput. Chem.* **2008**, *29*, 334-342.

(6) Asadchev, A.; Allada, V.; Felder, J.; Bode, B. M.; Gordon, M. S.; Windus, T. L. *J. Chem. Theory Comput.* **2010**, *6*, 696-704.

(7) Asadchev, A.; Gordon, M. S. *J. Chem. Theory Comput.* **2012**, *8*, 4166-4176.

(8) Miao, Y. P.; Merz, K. M. *J. Chem. Theory Comput.* **2013**, *9*, 965-976.

(9) Titov, A. V.; Ufimtsev, I. S.; Luehr, N.; Martinez, T. J. *J. Chem. Theory Comput.* **2013**, *9*, 213-221.

(10) Luehr, N.; Ufimtsev, I. S.; Martinez, T. J. *J. Chem. Theory Comput.* **2011**, *7*, 949-954.

(11) Wilkinson, K. A.; Sherwood, P.; Guest, M. F.; Naidoo, K. J. *J. Comput. Chem.* **2011**, *32*, 2313-2318.

(12) Götz, A. W.; Wölfle, T.; Walker, R. C. In *Annual Reports in Computational Chemistry*; Ralph, A. W., Ed. 2010; Vol. 6, p 21-35.

(13) Kulik, H. J.; Martinez, T. J. *Abstr Pap Am Chem S* **2012**, *244*.

(14) Kulik, H. J.; Luehr, N.; Ufimtsev, I. S.; Martinez, T. J. *J Phys Chem B* **2012**, *116*, 12501-12509.

(15) Isborn, C. M.; Gotz, A. W.; Clark, M. A.; Walker, R. C.; Martinez, T. J. *J. Chem. Theory Comput.* **2012**, *8*, 5092-5106.

(16) Ufimtsev, I. S.; Luehr, N.; Titov, A.; Martinez, T. *Abstr Pap Am Chem S* **2011**, *242*.

(17) Ufimtsev, I. S.; Luehr, N.; Martinez, T. J. *J Phys Chem Lett* **2011**, *2*, 1789-1793.

(18) Kulik, H. J.; Isborn, C. M.; Luehr, N.; Ufimtsev, I.; Martinez, T. J. *Abstr Pap Am Chem S* **2011**, *242*.

(19) Bhaskaran-Nair, K.; Ma, W. J.; Krishnamoorthy, S.; Villa, O.; van Dam, H. J. J.; Apra, E.; Kowalski, K. *J. Chem. Theory Comput.* **2013**, *9*, 1949-1957.

(20) Ma, W. J.; Krishnamoorthy, S.; Villa, O.; Kowalski, K. *J. Chem. Theory Comput.* **2011**, *7*, 1316-1327.

(21) Isborn, C. M.; Luehr, N.; Ufimtsev, I. S.; Martinez, T. J. *J. Chem. Theory Comput.* **2011**, *7*, 1814-1823.

(22) Vogt, L.; Olivares-Amaya, R.; Kermes, S.; Shao, Y.; Amador-Bedolla, C.; Aspuru-Guzik, A. *J. Phys. Chem. A* **2008**, *112*, 2049-2057.

(23) Isborn, C. M.; Luehr, N.; Gour, J. R.; Ufimtsev, I. S.; Martinez, T. J. *Abstr Pap Am Chem S* **2011**, *242*.

(24) Gotz, A. W.; Williamson, M. J.; Xu, D.; Poole, D.; Le Grand, S.; Walker, R. C. *J. Chem. Theory Comput.* **2012**, *8*, 1542-1555.

(25) Salomon-Ferrer, R.; Götz, A. W.; Poole, D.; Le Grand, S.; Walker, R. C. *J. Chem. Theory Comput.* **2013**, *9*, 3878-3888.

(26) D.A. Case; T.A. Darden; T.E. Cheatham, I.; C.L. Simmerling; J. Wang; R.E. Duke; R. Luo; R.C. Walker; W. Zhang; K.M. Merz; B.P. Roberts; B. Wang; S. Hayik; A. Roitberg; G. Seabra; I. Kolossváry; K.F. Wong; F. Paesani; J. Vanicek; J. Liu; X. Wu; S.R. Brozell; T. Steinbrecher; H. Gohlke; Q. Cai; X. Ye; J. Wang; M.-J. Hsieh; G. Cui; D.R. Roe; D.H. Mathews; M.G. Seetin; C. Sagui; V. Babin; T. Luchko; S. Gusarov; A. Kovalenko; Kollman, P. A. *AMBER11,University of California, San Francisco*. **2010**.

(27) DePrince, A. E.; Hammond, J. R. *J. Chem. Theory Comput.* **2011**, *7*, 1287-1295.

(28) DePrince, A. E.; Kennedy, M. R.; Sumpter, B. G.; Sherrill, C. D. *Mol Phys* **2014**, *112*, 844-852.

(29) Asadchev, A.; Gordon, M. S. *J. Chem. Theory Comput.* **2013**, *9*, 3385-3392.

(30) Titov, A. V.; Ufimtsev, I.; Martinez, T.; Dunning, T. H. *Abstr Pap Am Chem S* **2010**, *240*.

(31) Gill, P. M. W. *Adv Quantum Chem* **1994**, *25*, 141-205.

(32) Fletcher, G. D. *Int J Quantum Chem* **2006**, *106*, 355-360.

(33) Rys, J.; Dupuis, M.; King, H. F. *J. Comput. Chem.* **1983**, *4*, 154-157.

(34) McMurchie, L. E.; Davidson, E. R. *J. Comput. Phys.* **1978**, *26*, 218-231.

(35) Obara, S.; Saika, A. *J. Chem. Phys* **1988**, *89*, 1540-1559.

(36) Head-Gordon, M.; Pople, J. A. *J. Chem. Phys* **1988**, *89*, 5777-5786.

(37) Boys, S. F. *Proceedings of the Royal Society of London. Series A. Mathematical and Physical Sciences* **1950**, *200*, 542-554.

(38) Komornicki, A.; Ishida, K.; Morokuma, K.; Ditchfield, R.; Conrad, M. *Chem Phys Lett* **1977**, *45*, 595-602.

(39) Strout, D. L.; Scuseria, G. E. *J. Chem. Phys* **1995**, *102*, 8448-8452.

(40) Miao, Y; He, X; Ayers, K.; Brothers, E.; Merz, K.M. QUICK, version 2.0.140304; University of Florida: Gainesville, FL, 2013, Download free at: http://www.merzgroup.org/quick.html

(41) He, X.; Merz, K. M. *J. Chem. Theory Comput.* **2010**, *6*, 405-411.

(42) Van Lenthe, J. H.; Zwaans, R.; Van Dam, H. J. J.; Guest, M. F. *J. Comput. Chem.* **2006**, *27*, 926-932.

(43) Berman, H. M.; Westbrook, J.; Feng, Z.; Gilliland, G.; Bhat, T. N.; Weissig, H.; Shindyalov, I. N.; Bourne, P. E. *Nucleic Acids Res* **2000**, *28*, 235-242.

(44) "Megakernels Considered Harmful: Wavefront Path Tracing on GPUs" Samuli Laine (NVIDIA), Tero Karras (NVIDIA), Timo Aila (NVIDIA), in High-Performance Graphics 2013, July 2013

# CHAPTER 5. GPU ACCELERATION ON HISTOGRAM ANALYSIS

## 5.1. INTRODUCTION

The concept of the potential of mean force (PMF) [1] $\mathcal{W}(\vec{\xi})$ with coordinate $\vec{\xi}$, was originally introduced by Kirkwood and is frequently used to understand the mechanism of "rare" transitions in solid, fluids or complex biochemical systems. To obtain accurate estimates of a free energy barrier along a reaction coordinate that is substantially higher than the minima, a standard canonical (NVT) simulation provides little or no sampling in the barrier region. In order to address this issue, a set of N separate "umbrella" sampling [2,3] windows are created using a biasing potential, $w(\vec{\xi})$, in order to confine the system in the neighborhood of position $\vec{\xi}$ in order to enhance sampling. A typical umbrella biasing potential uses the harmonic form:

$$w_i(\vec{\xi}) = \frac{K_i}{2}(\vec{\xi} - \vec{\xi_i})^2 \qquad (1)$$

which restrains the system at position $\vec{\xi_i}$ with a force constant $K_i$. The windows created along a reaction path are then "unbiased" and combined to obtain the PMF over the whole region of interest.

The weighted histogram analysis method (WHAM) described by Kumar et al[4-6] is an algorithm that unbiases a set of simulations that have sampled different regions of $\vec{\xi_i}$ and then assembles the free energy profile. This method is based on the histogram method proposed by Ferrenberg and Swendsen [7], whose central idea is an optimal estimate of an unbiased distribution function as a weighted sum over data extracted from simulations using the maximum overlap method and statistical error minimization. Not only single coordinate [8,9] but also multidimensional free energy surfaces [5] can be determined with this method, and many successful applications have been reported [10-12]. However, one of the difficulties of multidimensional WHAM is its cost which is O(Nm), where N is the number of simulations and m is the number of grid points. Here grid points are defined by the WHAM calculation itself and represent a further parsing of the region between umbrella sampling points. Hence, depending on the data set size the computational cost can range into days on a single processor for high-resolution WHAM calculations.

The use of graphics processing units (GPUs) in computational chemistry and biology has rapidly expanded over the past decade due to the availability of GPU software development tools and because GPUs offer massively parallel computing at a reasonable cost. Compared with traditional parallel approaches, such as OpenMP or MPI, which are designed for sequential execution, GPU programming is especially suitable for calculations that require massive data parallelism where thread computation is independent from each other. The most widely used GPU programming interface, which was introduced by NVIDIA, is the Compute Unified Device Architecture (CUDA) [13]. Within the CUDA framework, the most basic units called threads are arranged into one-, two- or three-dimensions to form a block, and blocks then form one-, two- or three-dimensional grids. Configuration of blocks and threads are logical concepts, so they can be tuned to maximize performance. CUDA provides threads "private" and fast, albeit limited registers, and threads in a block can communicate with each other via shared memory, which has medium access speed. Global memory (also know as DRAM) and constant memory are two types of memory that CUDA provides that are visible to every thread. Global memory has a large capacity (*e.g.*, 6.0 GB for the NVIDIA FERMI M2090) but large access latency, while constant memory is fast but read-only.

In this chapter, we implement Grossfield's WHAM program[14] to run on a GPU. We will briefly describe the theory of WHAM and then introduce the algorithm and strategy used in the GPU implementation. We then demonstrate the performance improvement by calculating the energy surface for a typical enzyme reaction. Finally, we draw conclusions in the final section of this chapter. Our GPU tests were performed on the widely used NVIDIA K40 with 12.0 GB of DRAM.

## 5.2. THEORY AND METHODS

In this section, we will first briefly review the theoretical basis of WHAM. WHAM deals with a set of histograms obtained using the same grid size from the N independent window simulations. The independent simulations are generally called windows, that have been collected via molecular dynamics simulations biased with an umbrella potential like that given in eq (1). Generally, like most statistical methods, the accuracy of the calculation and cost are dependent on the numbers of windows and the amount of sampling done at each window.

From the $i^{\text{th}}$ biased ensemble, suppose that the distribution function along the $\vec{\xi}$ is a Boltzmann distribution, then the unbiased distribution function obtained is

$$\rho_i^{(unbiased)}\left(\vec{\xi}\right) = e^{\beta[w_i\left(\vec{\xi}\right)-f_i]}\rho_i^{(biased)}\left(\vec{\xi}\right) \qquad (2)$$

Where $f_i$ is the umbrella potential perturbation constant, and the b is the inverse temperature $\beta = 1/k_b T$. It has been shown[4,7] that the optimal estimate of $\rho(\vec{\xi})$, the unbiased probability distribution at position $\vec{\xi}$ is:

$$\rho(\vec{\xi}) = \frac{\sum_{i=1}^{N} n_i e^{-\beta[w_i(\vec{\xi})-f_i]} \rho_i^{(unbiased)}(\vec{\xi})}{\sum_{j=1}^{N} n_j e^{-\beta[w_j(\vec{\xi})-f_j]}} \tag{3}$$

where $n_i$ is the number of data points obtained from ith the umbrella simulation used to construct the biased distribution function. We assume the probability distribution is normalized, and if it is not, we can simply normalize it in the end. Combining eqs(2) and (3),

$$\rho(\vec{\xi}) = \frac{\sum_{i=1}^{N} n_i \rho_i^{(biased)}(\vec{\xi})}{\sum_{j=1}^{N} n_j e^{-\beta\left[w_j(\vec{\xi})-f_j\right]}} \tag{4}$$

For the constant $f_i$, we have

$$e^{-\beta f_i} = \int d\vec{\xi}\, \rho(\vec{\xi}) e^{-\beta w_i(\vec{\xi})} \tag{5}$$

Eqs(4) and (5) are the WHAM equations, and they depend on each other, so in practice the unbiased potential is achieved through an iterative procedure by starting with an initial guess for the $f_i$ (usually set as 0) and then generating new estimates of $\rho(\vec{\xi})$ and $f_i$ until convergence is achieved. When implementing this algorithm, we divide the coordinates into very small grids that have the same size (not to be confused with the points used in the umbrella simulations). We can rewrite eqs (4) and (5) as:

$$\rho(\vec{\xi}) = \frac{\sum_{i=1}^{N} n_i(\vec{\xi})}{\sum_{j=1}^{N} n_j e^{-\beta\left[w_j(\vec{\xi})-f_j\right]}} \tag{6}$$

$$e^{-\beta f_i} = \sum_{\vec{\xi}} \rho(\vec{\xi}) e^{-\beta w_i(\vec{\xi})} \tag{7}$$

where $n_i(\vec{\xi})$ is the number of sample points located on the same grid point with $\vec{\xi}$, and $f_i$ is the average additional free energy.

Up to this point, eq (6) and (7) are applicable to a general N-dimensional space, but we have only implemented the two-dimensional case because 1-D is relatively fast and 2-D analyses are the most widely used. Higher dimensions can be analyzed in an analogous fashion, but have not been explored herein. However, extensions to 1-D or higher dimensions involve similar algorithms and can be readily implemented if needed. For two-dimensions, eq(6) yields $N_{grid\_x}$*$N_{grid\_y}$ equations, where, $N_{grid\_x}$ stands for the number of x axis grid points in each WHAM histogram. Each equation requires two summations, and each of them has N terms that need to be added up. Therefore, for eq(6), the computation cost is O(N*$N_{grid\_x}$*$N_{grid\_y}$). Eq(7) represents N independent equations and each one has $N_{grid\_x}$*$N_{grid\_y}$ elements to be gathered up yielding a computational cost of O(N*$N_{grid\_x}$*$N_{grid\_y}$) as well. Its worth noting that the computational cost is only relative to the number of umbrella windows and the number of WHAM grid points, and not to the number of sample points obtained in the constrained MD trajectory, but the accuracy depends on all the three factors.

```
__global__ void calc_rou(){
    // array numerator and denominator are stored in shared
    // memory so that thread in this block can have access
    __shared__ numerator[THREAD_PER_BLOCK];
    __shared__ denominator[THREAD_PER_BLOCK];
    for( int i = blockIdx.x; i < $N_{grid\_x}$; i+= gridDim.x){
        for( int j = blockIdx.y; j < $N_{grid\_y}$; j+= gridDim.y){
            k = threadIdx.x;  // k is the thread ID
                              //numerator for one thread is the
                             // histogram value fetched from global
                             //memory
            numerator[k] = get_histogram_val(i, j, k);

            // denominator requires potential calculation
            denominator[k] = get_number_points(k)
                               * exp(-beta*(calc_potential(i, j, k) –
                f[k]));
            __syncthreads();

            // thread 0 will summarize
            if (k ==0){
                summarize numerator array to numerator[0];
                summarize denominator array to denominator [0];

                // rou is two-dimensional array that store in global
                // memory as eq6 calculated
                rou[i][j] = numerator[0]/denominator[0];
            }
        }
```

**Figure 5.1.** Psudo-code for implementation of part 1.

Below, we describe the details of the present implementation on the CUDA platform. The description is presented using C++ pseudocode in C++ (see Figure 5.1). The lines after “//” in pseudocode are comments.

```
__global__ void calc_F(){
  // array F are 2-dimensional and stored in shared memory so that
  // every thread in this block can access it
    __shared__  F_cache[THREAD_PER_BLOCK][THREAD_PER_BLOCK];
    for( int i = blockIdx.x; i < $N_{windows}$; i+= gridDim.x){
        for( int j = threadIdx.x; j < $N_{grid\_x}$; j+= blockDim.x){
            for( int k = threaIdx.y; k < $N_{grid\_y}$; k+= blockDim.y){
                F_cache[j][k]=get_rou(j,k)*exp(-beta*calc_potential(j,k,i));
            }
        }
        __syncthreads();
        if(threadIdx.x == 0 && threadIdx.y == 0) {
            summarize F_cache to F_cache[0][0];
           // F is one-dimensional array that store F in global memory that
          // eq(7) wants to calculate.
            F[i] = F_cache[0][0];
        }
    }
```

**Figure 5.2.** Psudo-code for implementation of part 2.

To begin with, we call the implementation of eq(6) and eq(7) as Part 1 and Part 2 in the following discussion (see Figure 5.1 and 5.2). For eq(6), as the pseudocode shown in Figure 5.1 indicates, our strategy was to assign a block to an equation, then all threads in a block calculate $n_k e^{-\beta\left[w_k\left(\vec{\xi}\right)-f_k\right]}$ and fetch $n_k\left(\vec{\xi}\right)$ from global memory, where k is the thread id. Therefore, within the two-dimensional WHAM framework, we configure a two-dimensional block to map each equation with a one-dimensional thread. For example, Block (0,0) will work on $\rho((0,0))$, and, for example, the very first thread with threadID equals 0, will fetch $n_0((0,0))$ and calculate $n_0 e^{-\beta[w_0((0,0))-f_0]}$. After that, a reduction in a block is executed to sum the denominator and numerator. As a result,

$\rho(\vec{\xi})$ can be computed in one block and stored in global memory so that it is visible to all threads on the GPU. Similarly for eq(7), as the pseudocode listed in Figure 5.2 describes, each block works on one equation, and threads in this block generate $\rho(\xi(x,y))e^{-\beta w_i(\xi(x,y))}$, where $\xi(x,y)$ is the assigned grid point. The most suitable configuration in this case will be a one-dimensional block with two-dimensional threads. The configuration utilized is illustrated in Figure 5.3. It is worth noticing that the data transfer from CPU to GPU is carried out once before the first iteration starts rather than at every iteration, which includes the histogram and force constant(s), and the variables reside in the GPU until the probability distribution reaches convergence, and data is transferred from the GPU to CPU to download the distribution and $\rho(\vec{\xi})$ and $f_i$. Therefore, data transfer time is negligible compared to the full calculation.

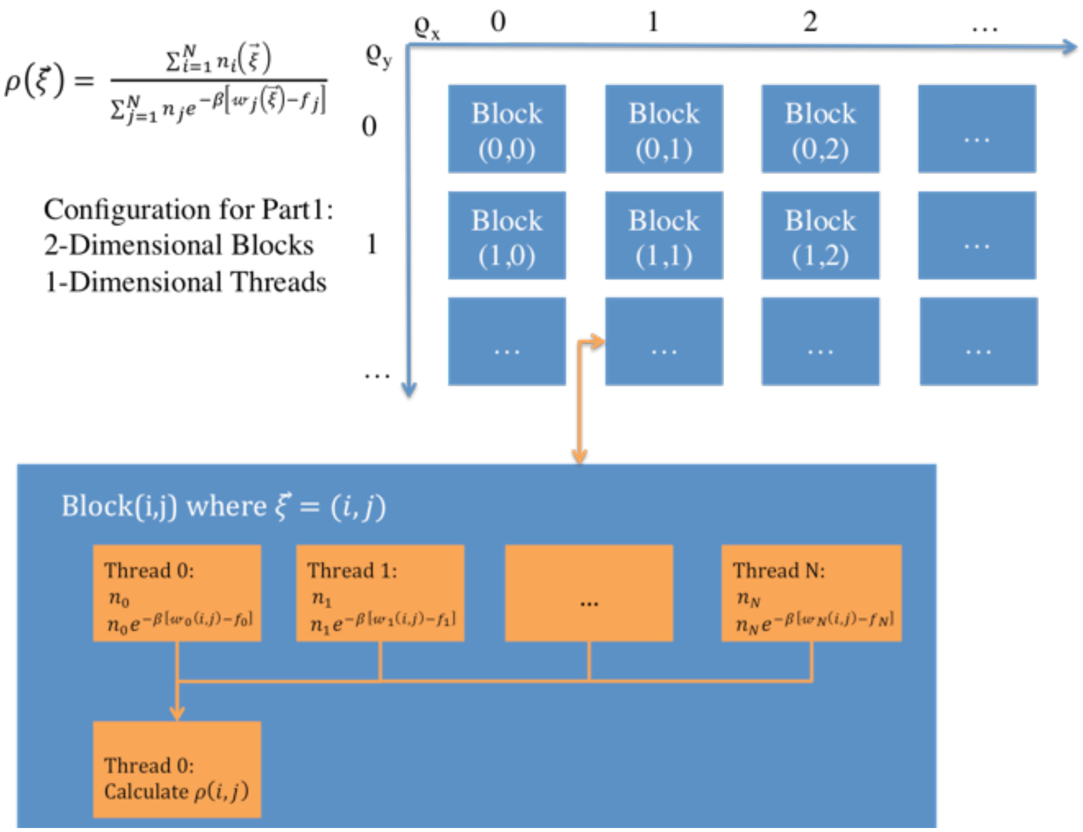

$\rho(\vec{\xi}) = \frac{\sum_{i=1}^{N} n_i(\vec{\xi})}{\sum_{j=1}^{N} n_j e^{-\beta[w_j(\vec{\xi})-f_j]}}$
Configuration for Part1:
2-Dimensional Blocks
1-Dimensional Threads
$\varrho_x$
0
1
2
…
$\varrho_y$
0
1
…
Block (0,0)
Block (0,1)
Block (0,2)
…
Block (1,0)
Block (1,1)
Block (1,2)
…
…
…
…
…
Block(i,j) where $\xi = (i,j)$
Thread 0: $n_0$ $n_0 e^{-\beta[w_0(i,j)-f_0]}$
Thread 1: $n_1$ $n_1 e^{-\beta[w_1(i,j)-f_1]}$
…
Thread N: $n_N$ $n_N e^{-\beta[w_N(i,j)-f_N]}$
Thread 0: Calculate $\rho(i,j)$

**Figure 5.3.** Configuration used for the first step in a WHAM calculation. Blue rectangles indicate a block unit, while orange rectangles represent threads.

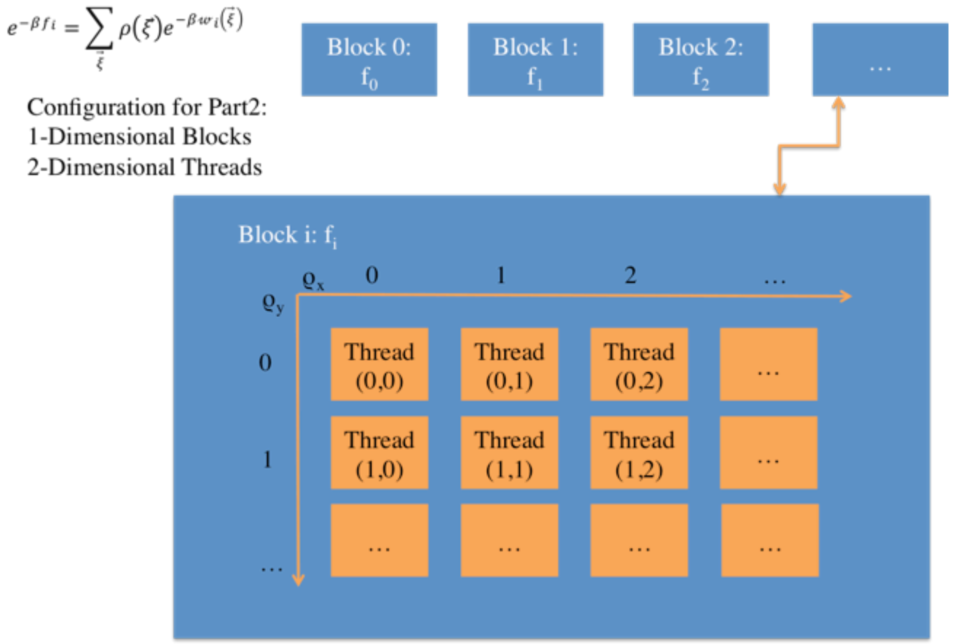


**Figure 5.4.** Configuration used for the second step in a WHAM calculation. Blue rectangles indicate a block unit, while orange rectangles represent threads.

## 5.3. RESULTS AND DISCUSSION

In this section, we benchmarked our GPU implementation by evaluating the energy surface of CusF metallochaperone. CusF is a periplasmic metallochaperone exist in Escherichia coli, encoded by cus-CFBA operon[30]. It was hypothesized to capture Cu+/Ag+ ions and transfer them to CusCBA pump in order to expel them out of the cell[31]. It is related to metal ion and antibiotic resistance, which makes it a potential pharmaceutical target.

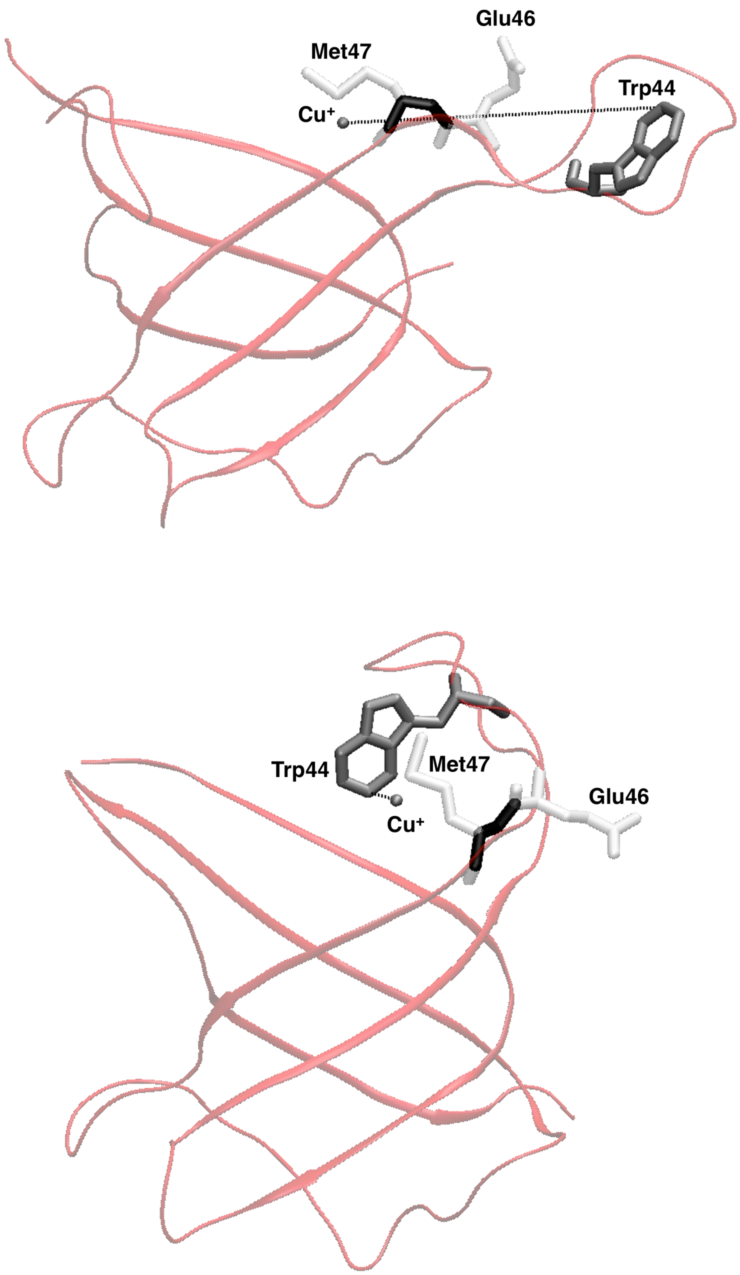


**Figure 5.5.** Benchmark reaction studied: CusF metallochaperone. Open(top) and close(bottom) state exist for CusF protein. Umbrella sampling carried on for two dimensions with x axis as one distance (Cu+--Trp44@CZ3 distance) and y axis as one dihedral angle (Glu36-Met47 Phi angle, highlighted in black). See text for details.

In recent simulations of our group, we found there is an open state structure exist for the CusF protein, while the transition between the closed and open states may play a significant role in the metal ion transfer process between CusF and CusCBA complexes (unpublished results). Just like the former work[32], we modeled the system based on the PDB entry 2VB2. AMBER ff99SB force field4 was employed for the protein system while the SPC/E water model5 was used to solvate the system. A hybrid metal model was utilized to model the metal binding site. The interaction of Cu+ ion and Trp44 residue was treated with nonbonded model while the interactions between Cu+ ion and the other three ligating residues (His36, Met47 and Met49) were represented by the bonded model. The bonded model are parameterized by Metal Center Parameter Builder (MCPB)6 software in Modeling ToolKit++ (MTK++) in AmberTools[33]. During the research we found there are one distance (Cu+--Trp44@CZ3 distance) and one dihedral angle (Glu36-Met47 Phi angle) in the metal binding site, as shown in Figure 5.5, could distinguish the closed and open state remarkably. For example, in a closed state structure we found the distance is 2.33 Å while the dihedral angle is -147.53°. In an open state structure the corresponding distance is 17.42 Å while the dihedral angle is 47.61°. To further clarify the free energy differences between the closed and open state, we processed a 2D potential of mean force (PMF) simulations by treating the Cu-Trp44@CZ3 distance as the reaction coordinate (RC) X axis while the Glu46-Met47 Phi dihedral angle as RC Y axis. We began the umbrella sampling from a closed state structure of metal loaded CusF. The harmonic potential of all the simulations is set as 10 kcal/mol$^{-1}$Å$^{-2}$ for the distance and 500 kcal/mol-1Rad$^{-2}$ for the dihedral angle. The initial closed state structure has a RC as (X=2.33Å, Y=-147.52°) while the PMF termination has

RC as (X=17.83 Å, Y=98.72°). The X-axis has 0.50 Å increased and Y- axis has 6.48° increased and in sum there are 32 * 39 =1408 windows for the whole umbrella sampling map. There are 6 ns sampling for each window with stepsize as 1 fs. The data points were stored for each 1 ps during the simulation so that sample data is uncorrelated.

As mentioned before, our implementation is based on Grossfield's WHAM program[14], so the original  code is used for the CPU calculations and we directly ported this code to run on GPUs. Our code can be downloaded under a GNU general public license. The GPU benchmarks were executed on an NVidia K40 card. This card has 6 Gigabytes of memory, 12 Streaming Multiprocessors and 2880 CUDA cores. For comparison, the CPU version was run on a single core of an Intel Xeon X5675 CPU with 3.07GHz frequency. Both codes were compiled using the Intel C++ Compiler 10.1.15 using optimization level 3(-O3) and the CUDA compiler 4.0 v0.2.1221 is used for the GPU code. We used the fast math library option (-use_fast_math) for better performance on the GPUs. Using a CPU, a single iteration takes 90 s on average, and each calculation takes more than two thousand iterations to reach convergence, which requires more than 50 hours in total to reach convergence on this data set. In most of our calculations, the grid was fine enough to obtain accurate results and in fact, but the effect of using coarser grids is examined below as well.

For GPU computations, single-precision (SP) should be carefully used because it may produce significantly less accurate results when compared with double-precision (DP), but offers a two-fold speed-up. In contrast, for a CPU, single and double-precision have similar performance on 64-bit hardware. So in our benchmarks, we do identical calculations for both single and double-precision in order to fairly compare their speed and accuracy.

The calculation efficiency is greatly dependent on the block and thread configuration. To find the most suitable one to maximize the performance, for first step, eq(6), we used different block sizes from (32*32) to (256*256) blocks per grid and thread sizes from 64 to 256 threads per block. Figure 5.6 shows the time spent in one iteration using different thread per block parameters for step 1. As it indicates, for the Tesla platform, the most suitable configuration is (256*256) blocks per grid and 128 threads per block for both precision options. For the second step, eq(7), we tested the same example making a similar comparison, and we find that the configuration with 192 blocks per grid and (32*32) threads per block performs best (see Figure 5.7). It is worth noting that the total time cost for single iteration decreased to 86.4 ms, which roughly represents more than 1000 times speedup relative to the CPU calculation.

In order to further compare the computational efficiency of GPU-WHAM, we examined another data set using various levels of WHAM grids, from 32*44 to 2048 * 2816. The profiling results are summarized in Figure 5.8. According to the benchmark results, the speedup achieved by a GPU compared to a single CPU reaches about 1200 times for double-precision and 2020 times for single-precision. The speedup is dependent on system size when the number of grid points is large enough. The grid used for this case is 256*352 and this resolution was used for plotting the PMF surface below. However, an artificially too high-resolution grid choice will produce unreliable results if the sampling was insufficient. However, we still profile the speedup of very fine grids since the computation time is independent to the thorough sampling of the data points. This exercise, while not physically justified given the amount of data we have does serve to show that even finer grids can be used with our GPU code. The largest grid choices in this test require about 1.39 second per iteration (single-precision) and about a half hour to complete the full calculation (3760 second GPU time and 3784 second total wall time), which covers 2061 iterations to reach convergence. The CPU implementation of WHAM is projected to finish the same task in about 814 hours. Besides WHAM grid point influence, when we compare the speedup horizontally between calculations with different numbers of umbrella windows but with the same number of grid points, the speedup increased slightly with more umbrella windows.

In practice, after running the MD simulations, the number of windows and the number of sample points is a constant, but the grid resolution can be specified in WHAM and grid resolution impact the accuracy of the calculation. Because of the speed improvements realized using GPU-WHAM, inaccuracies arising from low-resolution grids can be reduced if sufficient sampling data is collected. However, in most case, data collection is the most expensive part to generate an accurate result. However, this type of analysis is beyond the scope of the present chapter, which focuses on the creation of GPU-WHAM. For the benchmark cases studied the 160*640 grid scheme is fine enough that grid point density is not the factor that limits calculation precision, but higher-resolution does continue to offer some modest improvements.

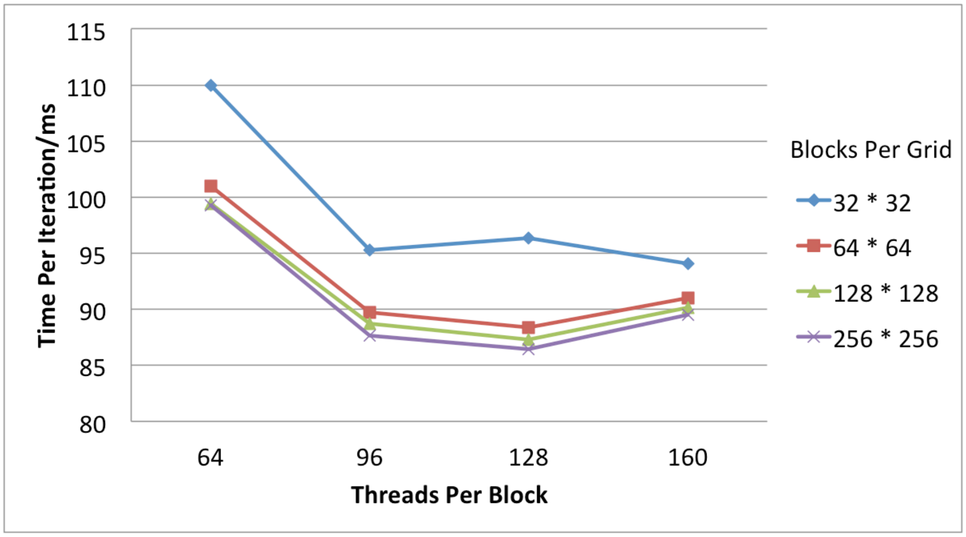


**Figure 5.6.** Profiling of the GPU-WHAM code for step 1 with different thread and block choices.

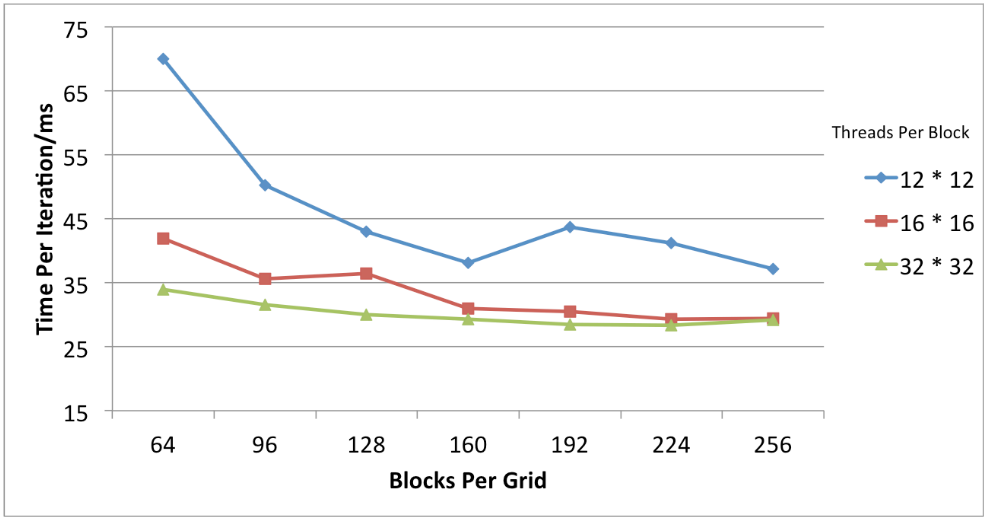


**Figure 5.7.** Profiling of the GPU-WHAM code for step 2 with different thread and block choices.

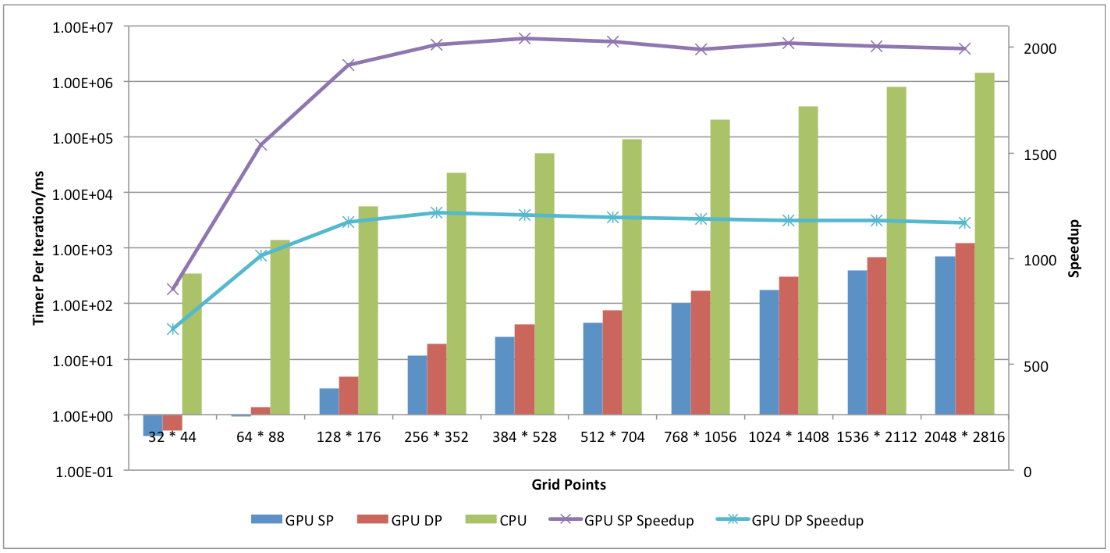


**Figure 5.8.** Benchmark results for two sets of data with different numbers of windows (1024 for top, 1533 for bottom) and different grid points (from 32*128 to 768*1024). Profiling with CPU, GPU single-precision (SP) and GPU double-precision (DP) calculation is presented. A logarithmic scale on the y-axis is used. Speedup by GPU SP and GPU DP is also plotted. The benchmark reaction is described in the text.

In order to evaluate single *versus* double precision we present the PMF surface with both single precision and double precision with the difference as well in Figure 5.9. As the figure suggests, the two PMF surfaces have only minor differences arising from the use of single precision. Our test results indicate that the average absolute energy difference and RMS (Root Mean Square), which is defined as,

$$RMS = \sqrt{\frac{\sum_{i=1,N_x,j=1,N_y}(E_{single}(i,j)-E_{double}(i,j)^2}{N_x \times N_y}} \qquad (8)$$

between double and single-precision is 9.5 E-7 kcal/mol and 1.5 E-4 kcal/mol, respectively. The maximum absolute difference between grid points is less than 0.4 kcal/mol with most of them being well under 0.01 kcal/mol. Most of the maximum of deviations appear regions near the edge of the PMF map and in areas of little relevance to the reactive process. These differences are acceptable for most applications of WHAM especially considering single-precision provides a 2-fold speed up in return. For example, the energy difference between the two minima of Figure 5.9 is 2.74 kcal/mol in SP and is 2.73 kcal/mol for DP, which is accurate enough for most chemical applications.

## 5.4. CONCLUSIONS

In this chapter, we accelerate WHAM using GPU technology by selecting a suitable computational configuration to maximum efficiency. The GPU-WHAM performance is as much as 1200 times in double precision faster for the Kepler card when compared with a single CPU. Single-precision is two times faster than double-precision with up to 2020 time speedup compared with a single CPU, but reduces accuracy slightly, but for most cases this will not be an issue because the introduced inaccuracy is generally less than the inaccuracy of the method utilized. Nonetheless, even very high-resolution grids are now affordable in a computationally reasonable amount of time.

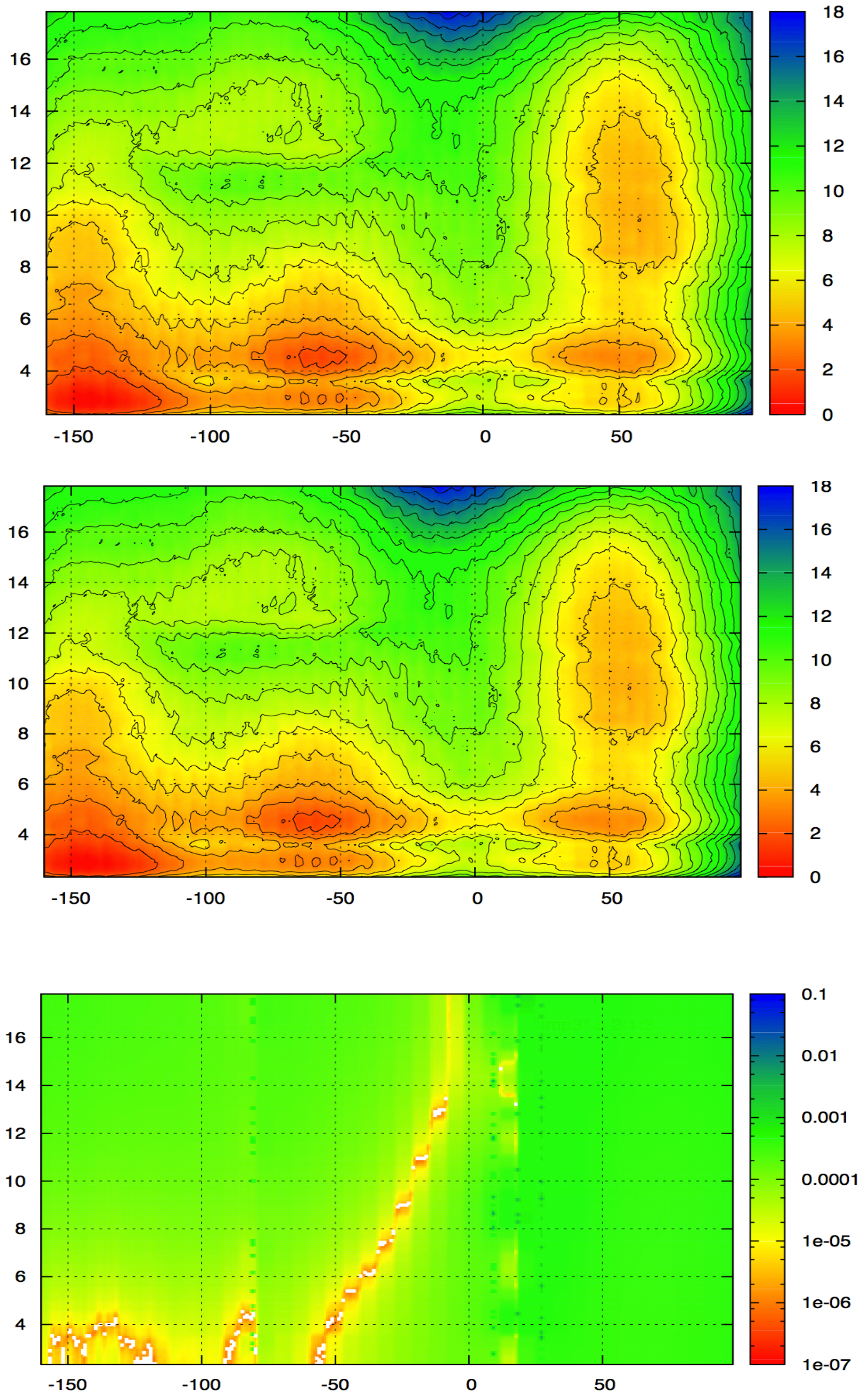


**Figure 5.9**. The PMF surface and contour map produced using the GPU-WHAM code

**Figure 5.9** (cont'd) with double-precision (top), single-precision (middle) and the absolute energy difference (bottom). The energy unit is kcal/mol. X- and y-axes are Cu+--Trp44@CZ3 distance and Glu36-Met47 Phi angle the unit is Å and rad. Full computational details are given in the text.

# APPENDIX

# MOLECULE DYNAMIC DETAILS OF CUSF

CusF is a periplasmic metallochaperone xist in *Escherichia coli*, encoded by *cus*-CFBA operon[36]. It was hypothesized to capture $Cu^+$/$Ag^+$ ions and transfer them to CusCBA pump in order to expel them out of the cell.[37] It is related to metal ion and antibiotic resistance, which makes it a potential pharmaceutical target.

In recent simulations of our group, we found there is an open state structure exist for the CusF protein, while the transition between the closed and open states may play a significant role in the metal ion transfer process between CusF and CusCBA complexes. Just like the former work[38], we modeled the system based on the PDB entry 2VB2. AMBER ff99SB force field[39] was employed for the protein system while the SPC/E water model[40] was used to solvate the system. A hybrid metal model was utilized to model the metal binding site. The interaction of $Cu^+$ ion and Trp44 residue was treated with nonbonded model while the interactions between $Cu^+$ ion and the other three ligating residues (His36, Met47 and Met49) were represented by the bonded model. The bonded model are parameterized by Metal Center Parameter Builder (MCPB)[41] software in Modeling ToolKit++ (MTK++) in AmberTools.

We found there are one distance ($Cu^+$--Trp44@CZ3 distance) and one dihedral angle (Glu36-Met47 Phi angle) in the metal binding site, could distinguish the closed and open state remarkably. For example, in a closed state structure we found the distance is 2.33 Å while the dihedral angle is -147.53°. In an open state structure the corresponding distance is 17.42 Å while the dihedral angle is 47.61°. To further clarify the free energy differences between the closed and open state, we processed a 2D potential of mean force (PMF) simulations by treating the Cu-Trp44@CZ3 distance as the reaction coordinate (RC) X axis while the Glu46-Met47 Phi dihedral angle as RC Y axis. We began the umbrella sampling from a closed state structure of metal loaded CusF. Generally there are three steps in the 2D PMF umbrella sampling process while the scheme of first two steps are shown in Figure 5.10. The harmonic potential of all the simulations is set as 10 kcal/mol$^{-1}$Å$^{-2}$ for the distance and 500 kcal/mol$^{-1}$Rad$^{-2}$ for the dihedral angle. There is 1 ns sampling for each window with stepsize as 1 fs. The data points were stored for each 10 fs during the simulation.

Step1: Firstly we performed the PMF simulations based on the conjugated changing the distance and dihedral variables to "push" the structure from closed state to open state. The initial closed state structure has a RC as (X=2.33Å, Y=-147.52°) while the PMF termination has RC as (X=21.33 Å, Y=98.72°). The total RC pathway was divided into 39 windows with the each later window has 0.50 Å increased in the X axis and 6.48° increased in the Y axis than the former window. The final snapshot of the former window was treated as the initial structure of next window.

Step 2: We began from the RC windows finished in the former step, and scanned across the Y axis for each certain X value individually. The scanning headed for two directions at the same time with the ending points for the negative direction and positive directions are -173.44° and 105.20° respectively. To be consistent with Step 1, the windows are spaced by 6.48° apart along the Y axis as well. Totally there are 44 windows across the Y axis for each certain X value. By multiplying the window numbers in X axis, in sum there are 44*39 =1716 windows for the whole umbrella sampling map. Up to then, we have finished the first 1 ns sampling of all the windows on the whole PMF profile map.

Step 3: The final umbrella samplings are performed independently for each window. By treating the final structure of the first 1 ns sampling as the initial structure, another 5 ns sampling was performed for each window. In total, there are 1716 windows with each window has 6 ns umbrella sampling. There are more than 1 billion correlated data points and more than 1 million uncorrelated data points (with treating 1 ps as the uncorrelated timescale) were collected for the umbrella sampling simulations. Based on which the free energy profile were generated by using the weighted histogram analysis method (WHAM) software.

It is notable to mention that in MD simulation, for x axis, there is 44 sample, but we only pick the first 32 to optimized and benchmark GPU-WHAM program because of the architecture of GPUs.

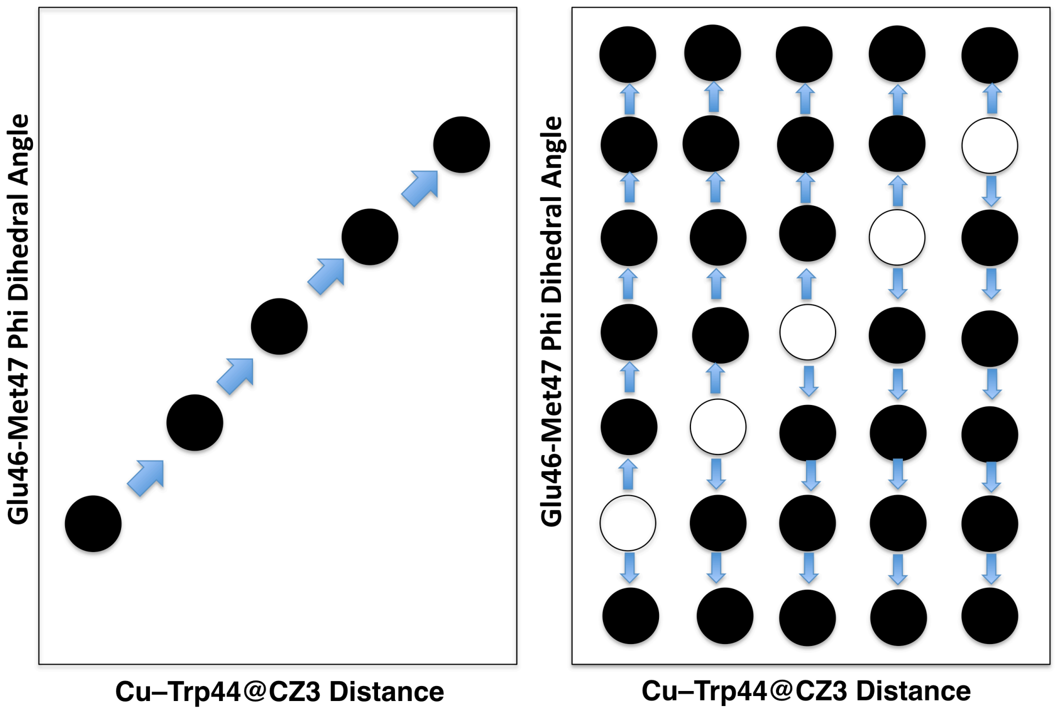

Glu46-Met47 Phi Dihedral Angle
Cu–Trp44@CZ3 Distance
Glu46-Met47 Phi Dihedral Angle
Cu–Trp44@CZ3 Distance

**Figure 5.10.** The scheme of step 1 (left) and step 2(right) in CusF 2D PMF simulations. The arrow indicated the direction of the reaction coordinate. The hollow circle on the right figure was the windows have been finished in step 1. (Just to be symbolize, the figure shows 5 windows on X axis and 7 windows on Y axis but actually there are 39 windows in the X axis and 44 windows in the Y axis)

# REFERENCES

# REFERENCES

(1) J. G. Kirkwood, *J. Chem. Phys.* **1935**, *3*, 300-313.

(2) G. M. Torrie, J. P. Valleau, *Chem. Phys. Lett.* **1974**, *28*, 578-581.

(3) G. M. Torrie, J. P. Valleau, *J. Comput. Phys.* **1977**, *23*, 187-199.

(4) S. Kumar, J. M. Rosenberg, D. Bouzida, R. H. Swendsen, P. A. Kollman, *J. Comput. Chem.* **1992**, *13*, 1021.

(5) S. Kumar, J. M. Rosenberg, D. Bouzida, R. H. Swendsen, P. A. Kollman, *J. Comput. Chem.* **1995**, *16*, 1350.

(6) S. Kumar, P. W. Payne, M. Vásquez, *J. Comput. Chem.* **1996**, *17*, 1269-1275.

(7) A. M. Ferrenberg, R. H. Swendsen, *Phys. Rev. Lett.* **1989**, *63*, 1195-1198.

(8) C. H. Bennett, *J. Comput. Phys.* **1976**, *22*, 245-268.

(9) E. M. Boczko, C. L. Brooks, *J. Phys. Chem.* **1993**, *97*, 4509-4513.

(10) G. Cui, B. Wang, K. M. Merz, *Biochemistry* **2005**, *44*, 16513-16523.

(11) G. Cui, X. Li, K. M. Merz, *Biochemistry* **2007**, *46*, 1303-1311.

(12) E. Rosta, M. Nowotny, W. Yang, G. Hummer, *J. Am. Chem. Soc.* **2011**, *133*, 8934-8941.

(13) NVIDIA. Compute Univied Device Architecture(CUDA). http://www.nvidia.com/object/cuda_home_new.html

(14) Alan Grossfield,, "WHAM: the weighted histogram analysis method", version 2.06, http://membrane.urmc.rochester.edu/content/wham

(15) D. Porezag, T. Frauenheim, T. Köhler, G. Seifert, R. Kaschner, *Phys. Rev. B* **1995**, *51*, 12947-12957.

(16) G. Seifert, D. Porezag, T. Frauenheim, *Int. J. Quant. Chem.* **1996**, *58*, 185-192.

(17) D.A. Case. T.A. Darden, T.E. Cheatham, I., C.L. Simmerling, J. Wang, R.E. Duke, R. Luo, R.C. Walker, W. Zhang, K.M. Merz, B.P. Roberts, B. Wang, S. Hayik, A. Roitberg, G. Seabra, I. Kolossváry, K.F. Wong, F. Paesani, J. Vanicek, J. Liu, X. Wu, S.R. Brozell, T. Steinbrecher, H. Gohlke, Q. Cai, X. Ye, J. Wang, M.-J. Hsieh, G. Cui, D.R. Roe, D.H. Mathews, M.G. Seetin, C. Sagui, V. Babin, T. Luchko, S. Gusarov, A. Kovalenko, Kollman, P. A. *AMBER11,University of California, San Francisco*. **2010**.

(18) P. Kollman, *Chem. Rev.* **1993**, *93*, 2395-2417.

(19) T. Bereau, R. H. Swendsen, *J. Comput. Phys.* **2009**, *228*, 6119-6129.

(20) B. Roux, *Comput. Phys. Commun.* **1995**, *91*, 275-282.

(21) D. Bouzida, P. A. Rejto, G. M. Verkhivker, , *Int. J. Quant. Chem.* **1999**, *73*, 113-121.

(22) M. Souaille, B. Roux, *Comput. Phys. Commun.* **2001**, *135*, 40-57.

(23) E. Gallicchio, M. Andrec, A. K. Felts, R. M. Levy, *J.Phys. Chem. B* **2005**, *109*, 6722-6731.

(24) J. D. Chodera, W. C. Swope, J. W. Pitera, C. Seok, K. A. Dill, *J. Chem. Theory Comput.* **2006**, *3*, 26-41.

(25) V. Hornak, R Abel, A. Okur, B. Strockbine, A. Roitberg, C. Simmerling, *Proteins: Struct., Funct., Bioinf.* **2006**, *65*, 712-725.

(26) R.C. Walker, M.F. Crowley, D.A. Case, *J. Comput. Chem.* **2008**, *29*, 1019-1031.

(27) G.M. Seabra, R.C. Walker, M. Elstner, D.A. Case, A. Roitberg, *J.Phys. Chem. A.* **2007**,*111*, 5655-5664.

(28) J. D. Chodera, W. C. Swope, J. W. Pitera, C. Seok, K. A. Dill, *J. Chem. Theory Comput.* **2007**, *3*, 26-41

(29) H. M. Berman, J. Westbrook, Z. Feng, G. Gilliland, T. N. Bhat, H. Weissig, I. N. Shindyalov, P. E. Bourne, *Nucleic Acids Res.* **2000**, *28*, 235-242.

(30) S. Franke, G. Grass, C. Rensing, D.H. Nies, *J. Bacteriol.* **2003**, 185, 3804-3812.

(31) I. R. Loftin, S. Franke, S. A. Roberts, A. Weichsel, A. Héroux, W. R. Montfort, C. Rensing, M. M. McEvoy, *Biochemistry* **2005**, *44*, 10533-10540.

(32) D. K. Chakravorty, B. Wang, M. N. Ucisik, K. M. Merz, *J. Am. Chem. Soc.* **2011**, *133*, 19330-19333.

(33) D. A. Case, T. E. Cheatham, T. Darden, H. Gohlke, R. Luo, K. M. Merz, A. Onufriev, C. Simmerling, B. Wang, R. J. Woods, *J. Comput. Chem.* **2005**, *26*, 1668-1688.

(34) H. J. C. Berendsen, J. R. Grigera, T. P. Straatsma, *J. Phys. Chem.* **1987**, *91*, 6269-6271.

(35) M. B. Peters, Y. Yang, B. Wang, L. s. Füsti-Molnár, M. N. Weaver, K. M. Merz, J. *Chem. Theory Comput.* **2010**, *6*, 2935-2947.

(36) Franke, S.; Grass, G.; Rensing, C.; Nies, D. H. *J. Bacteriol.* **2003**, *185*, 3804-

3812.

(37) Loftin, I. R.; Franke, S.; Roberts, S. A.; Weichsel, A.; Héroux, A.; Montfort, W. R.; Rensing, C.; McEvoy, M. M. *Biochemistry* **2005**, *44*, 10533-10540.

(38) Chakravorty, D. K.; Wang, B.; Ucisik, M. N.; Merz, K. M. *J. Am. Chem. Soc.* **2011**, *133*, 19330-19333.

(39) Case, D. A.; Cheatham, T. E.; Darden, T.; Gohlke, H.; Luo, R.; Merz, K. M.; Onufriev, A.; Simmerling, C.; Wang, B.; Woods, R. J. *J. Comput. Chem.* **2005**, *26*, 1668-1688.

(40) Berendsen, H. J. C.; Grigera, J. R.; Straatsma, T. P. *J. Phys. Chem.* **1987**, *91*, 6269-6271.

(41) Peters, M. B.; Yang, Y.; Wang, B.; Füsti-Molnár, L. s.; Weaver, M. N.; Merz, K. M. *J. Chem. Theory Comput.* **2010**, *6*, 2935-2947.

# CHAPTER 6. SUMMARY AND CONCLUSION REMARKS

In chapter 1, we briefly introduced GPU and CUDA programming by illustrating GPU architecture, thread hierarchy and memory hierarchy. We also provided some details about philosophy of GPU programming and tuning tricks, which enlighten the following research projects to realize GPU programming on quantum chemistry.

In chapter 3, we evaluated ERIs on a GPU using recurrence relations and form the Fock matrix entirely in GPU memory. And in full SCF benchmark calculations demonstrate the impressive speedups GPU ERI implementations achieved and the energy error GPU produced associated with double-precision calculations meets most of computational chemistry calculation accuracy requirements. To realize the GPU implementations and achieve the speedup, a well-sorted integral grid that reduces thread divergence and provides an optimized memory access pattern boosts the performance in terms of efficiency. This speedup is also achieved by optimizing the Fock matrix formation scheme by introducing the atomic-operation to significantly reduce data transfer from $N^4$ to $N^2$, which was one of the most time-consuming steps in conventional GPU SCF programming that limited by th CPU-GPU bandwidth. Moreover, this approach also reduces redundant and unnecessary re-calculated ERI calculation by reusing ERI data. Our benchmarks show the speedup increases with increasing system size, and our code now is applicable to s, p and d orbital functions which are in most most organic or biochemistry calculations.

In the chapter 4, based on the work we completed in last chapter, we implemented the evaluation of ERIs up to f orbitals and ERI derivatives up to d orbitals using GPU technology and ERI recurrence relations. Our SCF and gradient calculations demonstrate the similar speedup of GPU implementation compared to over modern CPUs. A partition strategy is introduced to solve the difficulties encountered in computing ERIs including f orbitals and was further applied to d orbital gradient calculation. Importantly, we observed a very limited efficiency decrease after employed this strategy. The code is written efficiently by our machine-generated code which is almost impossible for human to complete because its complicity. Similar to ERI evaluation, a well-sorted pre-sorting strategy and several other improvements boost overall GPU performance and atomic operations are used as well to reduce data transfer. Moreover, GPU-based DFT calculation is also available in QUICK although we did not touch on this topic.

In chapter 5, to boost molecular dynamic simulation, we accelerate WHAM using GPU technology by selecting a suitable computational configuration to maximum efficiency. The GPU-WHAM performance is as much as 1200 times in double precision faster for compared with a single CPU while single-precision is two times faster than double-precision with up to 2020 time speedup reducing accuracy slightly. But according to our test and observation, for most cases this will not be an issue because the introduced inaccuracy is generally less than the inaccuracy of the method utilized. Nonetheless, even very high-resolution grids are now affordable in a computationally reasonable amount of time.

These projects utilized GPU as an economical accelerator to multiply areas of computational chemistry calculations. Meanwhile, there is still plenty of room to apply

GPU technology. Therefore, our future work will focus on following aspects. First of all, we are integrating our source code with the AMBER MD package to further enable *ab initio* QM/MM simulations. Second, we are developing GPU-based ERI and ERI derivative generator that computational chemist can use our tool to develop their own algorithm by using GPU to accelerate ERI calculation, a very common step for most advanced *ab initio* computational chemistry methods. Moreover, we are going to develop GPU enabled correlated *ab initio* methods such as MP2 and coupled-cluster methods. In addition, to calculate frequencies and other physical and chemical properties, GPU enable second order derivatives would also be helpful. In terms of technology, a multi-GPU implementation and implementation of the code on INTEL PHI platform is on our time-line order to have another option to accelerate *ab initio* calculation.