JCTC Journal of Chemical Theory and Computation

Article

pubs.acs.org/JCTC

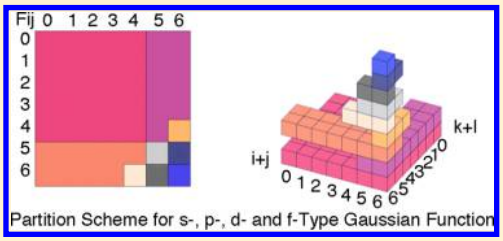
Acceleration of High Angular Momentum Electron Repulsion Integrals and Integral Derivatives on Graphics Processing Units

Yipu Miao and Kenneth M. Merz, Jr.*

Department of Chemistry Quantum Theory Project, University of Florida, 2328 New Physics Building, Gainesville, Florida 32611-8435, United States

Department of Chemistry and Department of Biochemistry and Molecular Biology, Michigan State University, 578 S. Shaw Lane, East Lansing, Michigan 48824-1322, United States

ABSTRACT: We present an efficient implementation of ab initio self-consistent field (SCF) energy and gradient calculations that run on Compute Unified Device Architecture (CUDA) enabled graphical processing units (GPUs) using recurrence relations. We first discuss the machine-generated code that calculates the electron-repulsion integrals (ERIs) for different ERI types. Next we describe the porting of the SCF gradient calculation to GPUs, which results in an acceleration of the computation of the first-order derivative of the ERIs. However, only s, p, and d ERIs and s and p derivatives could be executed simultaneously on GPUs using the current version of CUDA and generation of NVidia GPUs using a previously described algorithm [Miao and Merz *J. Chem. Theory Comput.* **2013**, *9*, 965−976.]. 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 10−18 times relative to traditional CPU execution. An accuracy analysis using double-precision calculations demonstrates that the overall accuracy is satisfactory for most applications.



1. INTRODUCTION

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 (GPGPUs), 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 (HF-SCF) and density functional theory (DFT) methods are widely used to rationalize the behavior of molecular systems. However, the broad application of these methods, to large systems containing thousands 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 GPUs 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] and 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 the computation of the energy gradients is also desired 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 the computational chemistry community in its ongoing effort to study larger chemical systems.

Recently, 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

Received: November 3, 2014
Published: February 19, 2015

DOI: 10.1021/ct500984t
J. Chem. Theory Comput. 2015, 11, 1449−1462

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 the MP2[22] and coupled-cluster methods[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 paper, 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 paper, 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 paper with a brief discussion and conclusions.

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

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

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 eq 1 contributing to both the Coulomb and exchange ERI terms. Four-centered ERIs are given by

$$(\mu\nu|\kappa\lambda) = \iint \phi_\mu(\vec{r}) \phi_\nu(\vec{r}) \frac{1}{r - r'} \phi_\kappa(\vec{r'}) \phi_\lambda(\vec{r'})\, \mathrm{d}\vec{r}\, \mathrm{d}\vec{r'} \tag{2}$$

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

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

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 eq 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.

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.

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

A primitive Cartesian Gaussian function centered at $\vec{A} = (A_x + A_y + A_z)$ with exponent α 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 eq 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, three [pslss] 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, [ddldd] will have 1296 kinds of different ERIs because each index can have 6 different integer combinations to satisfy the precondition 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.

DOI: 10.1021/ct500984t
J. Chem. Theory Comput. 2015, 11, 1449−1462

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

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

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

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

and α, β, γ, and δ 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

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

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

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

In, eq 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

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

and can be applied to contracted ERIs

$$\begin{aligned}(a(b+1_i)|cd)^{(m)} &= ((a+1_i)b|cd)^{(m)} + (A_i - B_i)(ab|cd)^{(m)} \\ &= \sum_{a\le k\le a+b+1, c\le l\le c+d} c_{kl}(k0|l0)^{(m)} \end{aligned} \tag{16}$$

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 eq 7, and we can obtain the (half) coefficient analytically as well from eq 16.

$$\begin{aligned}[(a+1_i)0|c0]^{(m)} &= (P_i - A_i)[a0|c0]^{(m)} \\ &+ (W_i - P_i)[a0|c0]^{(m+1)} + \frac{a_i}{2\zeta}\left([(a-1_i)0|c0]^{(m)}\right. \\ &\left. - \frac{\eta}{\eta+\zeta}[(a-1_i)0|c0]^{(m+1)}\right) \\ &+ \frac{c_i}{2(\eta+\zeta)}[a0|(c-1_i)0]^{(m+1)} \\ (a_n b_{n'}| &= \sum_{i=0}^{n_x'}\sum_{j=0}^{n_y'}\sum_{k=0}^{n_z'}\binom{n_x'}{i}\binom{n_y'}{j}\binom{n_z'}{k}(A_x - B_x)^{n_x'-i} \\ &\times (A_y - B_y)^{n_y'-j}(A_z - B_z)^{n_z'-k}(a_{(n_x+i)(n_y+j)(n_z+k)}0| \end{aligned} \tag{17}$$

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:

$$\frac{\partial\chi_a}{\partial A_i} = 2\alpha(\chi_{a+1_i}) - a_i(\chi_{a-1_i}) \tag{18}$$

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.

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

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:

$$\left(\frac{\partial}{\partial A_i} + \frac{\partial}{\partial B_i} + \frac{\partial}{\partial C_i} + \frac{\partial}{\partial D_i}\right)(ab|cd) = 0 \tag{20}$$

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 eq 19 using

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

where the subscript α 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 eq 19, if three centers are

DOI: 10.1021/ct500984t
J. Chem. Theory Comput. 2015, 11, 1449−1462

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:

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

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

ζ, η, $\vec{P}$, $\vec{Q}$ and $\vec{W}$

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 $(0,0)^0$, $(0,0)^1$

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

...

$C_{0,1}$ $(0,1)^{i+j-1}$: $C_{0,0}$ $(0,0)^{i+j-1}$, $C_{0,0}$ $(0,0)^{i+j}$

...

$C_{i,j}$ $(i,j)^0$: $C_{i,(j-1)}$ $(i, j-1)^1$, $C_{i,(j-1)}$ $(i, j-1)^0$, $C_{i,(j-2)}$ $(i, j-2)^1$, $C_{i,(j-2)}$ $(i, j-2)$, $C_{(i-1),(j-1)}$ $(i-1, j-1)^1$

Return $(i,j)^0$

Figure 1. Class and function description for the ERI code. $(i, j)^m$ represent an ERI class for $[i0|j0]^{(m)}$ and class C_{ij} is built to express its relation with other classes with lower i and j and the same m and higher m values. The goal of function F_{ij} 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.

evaluated simultaneously, only one set of temporary ERIs is needed, which is of $[ab|cd]$ type, while eq 21 requires four sets ($(ab|cd)_a$, $(ab|cd)_b$, $(ab|cd)_c$, 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.

3. IMPLEMENTATION

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 multiprocessor 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 and 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) and also 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

DOI: 10.1021/ct500984t
J. Chem. Theory Comput. 2015, 11, 1449−1462

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.

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 eq 17, while the HRR step, which is given in eq 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 five other classes via the determination of the necessary coefficients (using eq 17). The class description is presented in Figure 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 eq 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 treelike vertical recurrence relation as expressed in eq 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 remain in the pool. As noted, in class C_{ij} of Figure 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 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 subexpression. 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 six, respectively. This is sufficient for [ffff] 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.

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. NVIDIA's CUDA Compiler (NVCC) can successfully compile and execute this subroutine. In our earlier paper,[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.

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 eq 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,

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

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 $Z_1 = \{a < 3 \text{ and } c < 3\}$, $Z_2 = \{a \geq 3 \text{ or } c \geq 3\}$. 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.

$$\begin{aligned}(a(b+1_i)|cd)^{(m)} &= \sum_{Z}\sum_{k,l\in Z} c_{kl}(k0|l0)^{(m)} \\ &= \sum_{Z}(((a+1_i)b|cd)_Z^{(m)} \\ &\quad + (A_i - B_i)(ab|cd)_Z^{(m)})\end{aligned} \tag{23}$$

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.

$$\frac{\partial}{\partial A_i}[ab|cd] = \sum_{Z}(2\alpha[(a+1_i)b|cd]_Z - a_i[(a-1_i)b|cd]_Z) \tag{24}$$

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

$$\begin{aligned}\frac{\partial}{\partial \mathrm{A}_i}(ab|cd) &= \sum_{Z}((((a+1_i)b|cd)_Z)_\alpha \\ &\quad - a_i((a-1_i)b|cd)_Z)\end{aligned} \tag{25}$$

With these equations, we can further consider a partition scheme. For now, we only consider F_{00} to F_{66}, which computes ERIs up to [ffff]. All of these 48 subroutines (again, F_{00} 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

DOI: 10.1021/ct500984t
J. Chem. Theory Comput. 2015, 11, 1449−1462

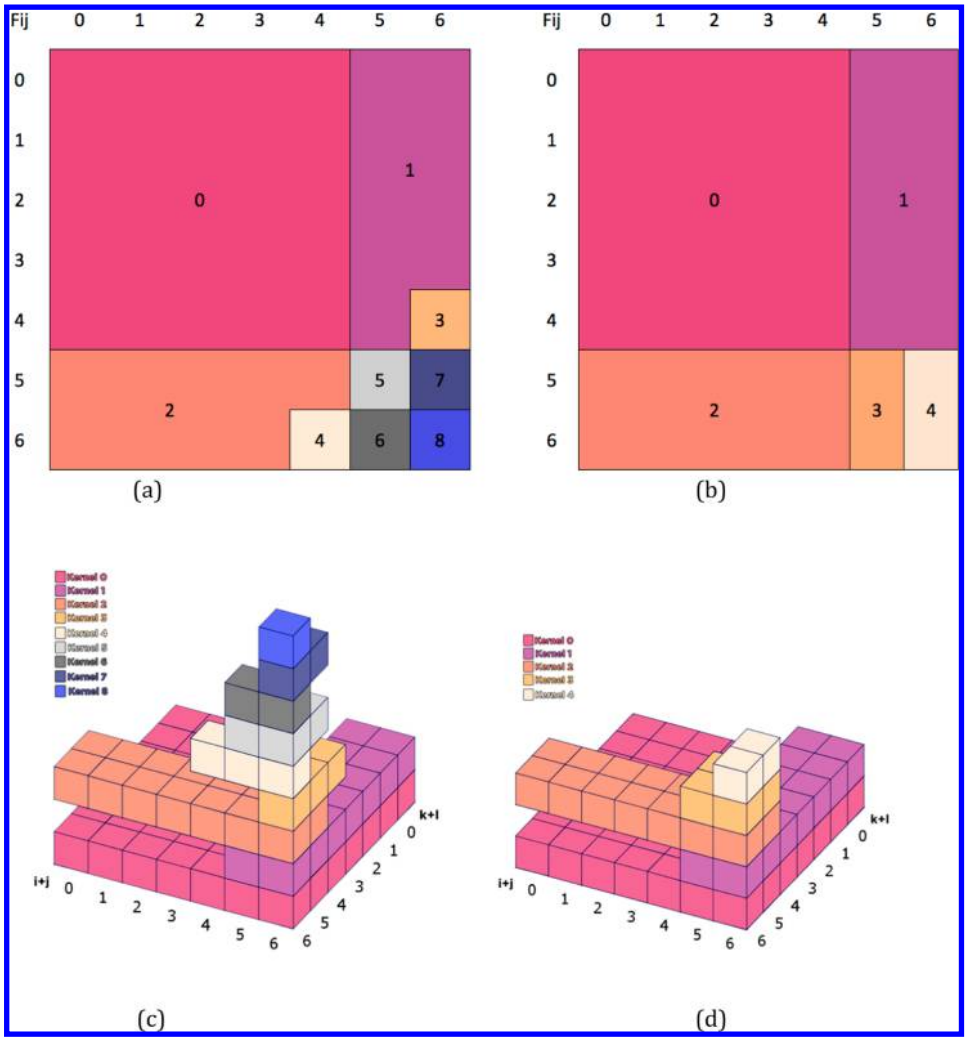


Figure 2. Optimized subroutine partition pattern for (a) CUDA compute capability 3.x and (b) CUDA compute capability 2.x. All F_{ij} functions that are within one zone will be warped in one subroutine and called in a kernel. (c and d) Kernel call pattern for ERI type $[ij|kl]$ with the partition strategy (c) for CUDA compute capability 3.x (part a) and (d) for CUDA compute capability 2.x (part b). See text for details.

scheme is to try to bind as many subroutines into one subroutine as possible. Therefore, we developed the partition design presented in Figure 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, nine and five 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 zones 3 and 1 in Figure 2a 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 used in our previous paper[8] and, indeed, most GPU-based SCF implantations, a presorting 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 presorting 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 $N^2 \times N^2$ 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 the following: (1) Different ERI types call different ERI subroutines, an operation 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. This 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. This 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]

DOI: 10.1021/ct500984t
J. Chem. Theory Comput. 2015, 11, 1449−1462

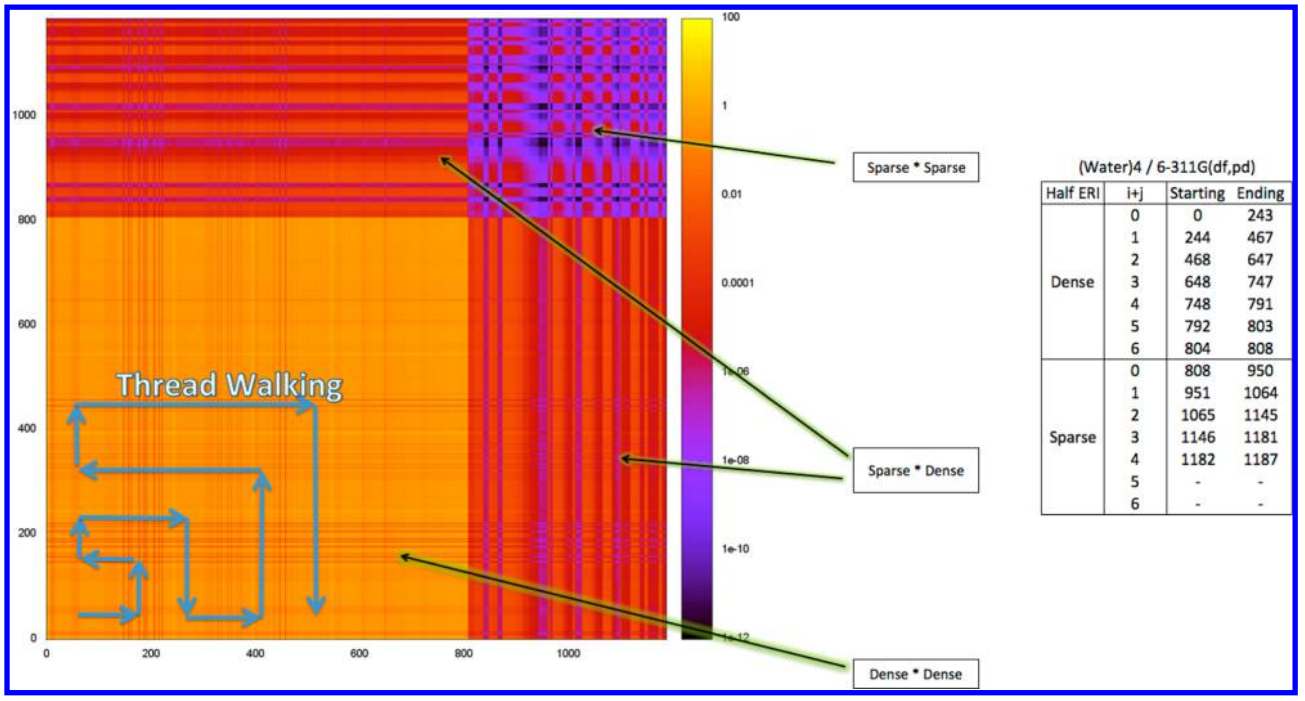
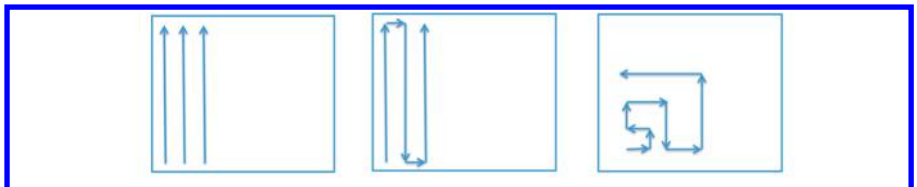


Figure 3. Presorting and thread walking illustration. The test case is a water cluster with four water molecules. The color scheme indicates the upper bound of ERIs, dense*dense, sparse*dense, and sparse*sparse areas can be visually identify in the ERI matrix. The table on the right is the boundary for different ERI types. See details in the text.



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. Thread walk strategy comparison. Three different thread walk strategies are presented for GPU calculations. Time used to calculate the second iteration for four examples of alanine chains of different lengths using 6-31g** are reported. The units are seconds. The improvement percent row indicates the performance improvement of the circular strategy over the linear strategy.

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

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 $i \leq j$ and $i \leq k$ and $k \leq l$ 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 3). Via this, the order of the half matrix will be randomized resulting in instances where $i > j$ which will be

DOI: 10.1021/ct500984t
J. Chem. Theory Comput. 2015, 11, 1449−1462

ignored by symmetry ($i \leq j$ are retained). If this conditional check is not made with the Schwartz cutoff divergence may occur. Without the Schwarz cutoff $i \leq j$ 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 3, using an example of a water cluster with four molecules 6 Å from each other such that both the inter- and intramolecular 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 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 3 indicates the boundaries for the half ERIs.

Besides this change, a second change is the thread walk was altered from a linear search to a circular search as shown in Figure 3. This adaptation is especially suitable to the first change because the large- 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.

We visualize the kernel call pattern in Figure 2c and d for an ERI type $[ijlkl]$ with two axes with $i + j$ and $j + k$ values, respectively. We show the partition strategies for CUDA capability of 3.x and 2.x in Figure 2a and b, respectively. Each cube corresponds to a kernel run for this type. For example, if $i + j$ equals 5 and $j + k$ equals 6, kernels 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 2a (designed for the device of CUDA compute capability 3.x), the kernel containing subroutines $F_{(5-6)(0-6)}$ is needed by ERI types $[ijlkl]$ 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.

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 eq 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 eq 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 [ddldd] requires a series of subroutines $F_{(0-5)(0-5)}$ 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 5, which represents

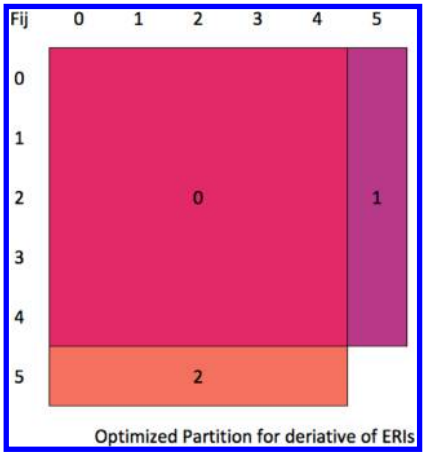


Figure 5. Optimized subroutine partition for ERI derivatives. This partition strategy works for both CUDA compute capability 3.x and 2.x. The kernel call mapping is similar to Figure 2 but simplified so it is not shown here.

a subset of the normal ERI evaluation process. As Figure 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 12 derivatives classes (four centers and three directions), but nine classes

DOI: 10.1021/ct500984t
J. Chem. Theory Comput. 2015, 11, 1449−1462

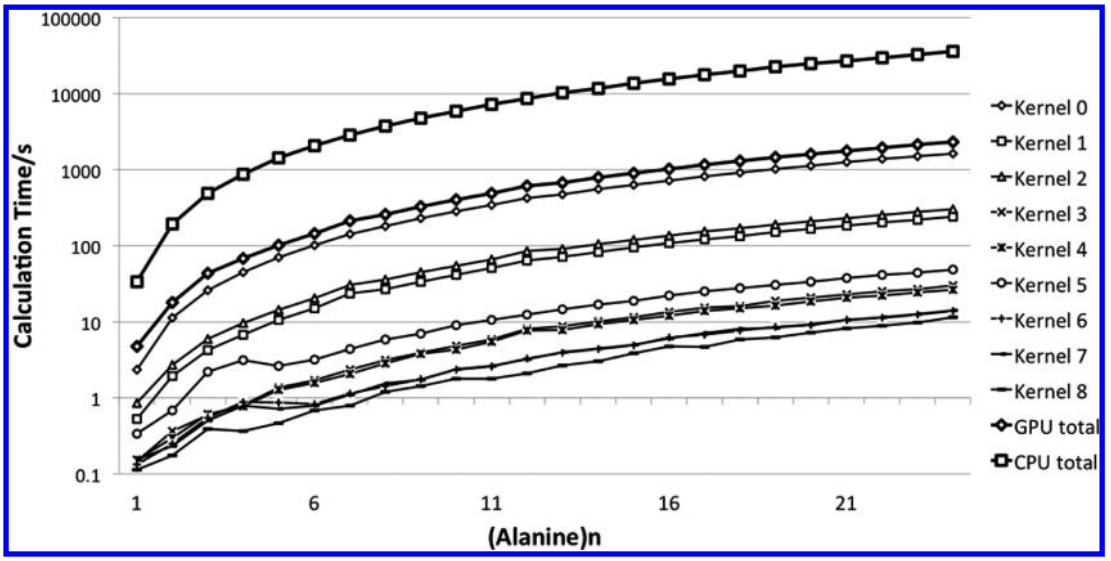


Figure 6. Calculation time profiles for the CPU and GPU calculations on alanine chains. Timings for the CPU and GPU kernel time (top two curves) are based on the Fock matrix formation excluding the one-electron contribution. Timings for kernels 0−8 are given as well and the kernel call mapping corresponds to Figure 2. Platform and software details are described in the text. A logarithmic scale is used for the *Y* axis.

need to be computed (see eq 20) simultaneously once the value of the auxiliary ERIs from [00|00] to $[(i + j + 1)0|(j + k)0]$ and $[(i + j)0|(j + k + 1)0]$ 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 six and three HRR computes are required at most for zones 1 and 2.

However, the contraction step, sacrificing space usage for reduced FLOP counts, given by eq 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 eq 21, an efficient way to implement this is to store four sets of auxiliary classes whose class size increases exponentially with the growth in quantum number: For example, because the degeneracy numbers of s, p, d, f, g, h, and i orbitals are 1, 3, 6, 10, 15, 21, and 28, for [dd|, $L = 4$, we have 1 + 3 + 6 + 10 + 15 = 35; therefore, [dd|dd] requires 35 × 35 for each class and 56 × 56 for the derivative; [ff|ff] 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 zones 1 and 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 2-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 ∼3× 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 zones 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 presorting strategy for the gradient computation as well, but this added overhead would eliminate the realized computational gains.

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 3.07

DOI: 10.1021/ct500984t
J. Chem. Theory Comput. 2015, 11, 1449−1462

Table 1. Performance Comparison between CPU and GPU SCF Calculation[a,b]

molecule/atom number	basis sets/no. of basis functions	kernel time/s									GPU total/s	CPU total/s	speedup
		0	1	2	3	4	5	6	7	8			
taxol (110)	6-31G**/1160	132.1									132.1	1735.1	**13.1**
	6-31G(df,dp)/2064	607.6	86.2	114.4	10.8	10.6	17.2	4.8	5.3	4.7	861.6	13178.3	**15.3**
	6-31G(2df,2dp)/2577	1488.6	221.3	283.7	20.1	20.2	38.5	9.0	9.3	9.1	2099.7	28856.4	**13.8**
valinomycin (168)	6-31G**/1620	276.3									276.3	3398.0	**12.3**
	6-31G(df,dp)/2940	1206.4	158.8	205.1	21.3	19.2	30.0	10.6	11.0	16.8	1679.1	24905.0	**14.9**
	6-31G(2df,2dp)/3678	3014.2	411.1	513.3	41.6	37.9	68.1	21.5	21.7	36.0	4165.5	56135.2	**13.4**
1AKG (209)	6-31G**/2171	616.0									616.0	7290.8	**11.8**
	6-31G(df,dp)/3839	2874.8	370.2	469.5	53.4	47.3	68.3	28.2	28.5	47.7	3987.7	49457.7	**12.4**
	6-31G(2df,2dp)/4829	6220.4	841.3	1055.0	90.6	81.0	141.3	48.9	48.3	86.6	8613.3	107771.6	**12.5**
1CNL (169)	6-31G**/1771	424.4									424.4	5054.8	**11.9**
	6-31G(df,dp)/3149	2033.5	263.8	335.7	35.9	32.5	48.9	17.7	18.6	26.3	2812.8	34725.4	**12.4**
	6-31G(2df,2dp)/3929	4405.8	597.6	744.0	62.0	56.4	98.0	29.6	30.7	49.0	6073.1	75738.1	**12.5**
1M2C (220)	6-31G**/2276	704.0									704.0	7969.3	**11.3**
	6-31G(df,dp)/4060	3245.2	410.6	511.8	61.7	52.6	76.4	33.3	34.3	60.8	4486.6	53311.6	**11.9**
	6-31G(2df,2dp)/5068	7055.4	937.3	1158.7	107.7	95.4	156.8	57.6	57.6	110.9	9737.4	119672.6	**12.3**
1PEN (203)	6-31G**/2131	607.5									607.5	7537.3	**12.4**
	6-31G(df,dp)/3789	2812.7	365.3	465.0	54.6	48.8	69.3	28.0	28.1	46.1	3917.8	48023.1	**12.4**
	6-31G(2df,2dp)/4728	6101.4	829.1	1034.7	89.0	80.8	138.7	46.9	46.9	82.3	8449.8	106282.3	**12.6**

[a]Time listed in the table indicates the time used to form the Fock Matrix in the second iteration of the SCF calculation excluding the one-electron contribution and the diagonalization time. [b]Platform and software details are described in the text.

GHz 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 875 MHz to maximize the performance. The K40 card has compute capability 3.5, so the kernel partition and kernel call pattern will follow Figure 2a 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 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 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 our previous paper[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 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.

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] IDs 1AKG, 1CNL, 1M2C, and 1PEN. The results are presented in Table 1. The number of basis functions varies

DOI: 10.1021/ct500984t
J. Chem. Theory Comput. 2015, 11, 1449−1462

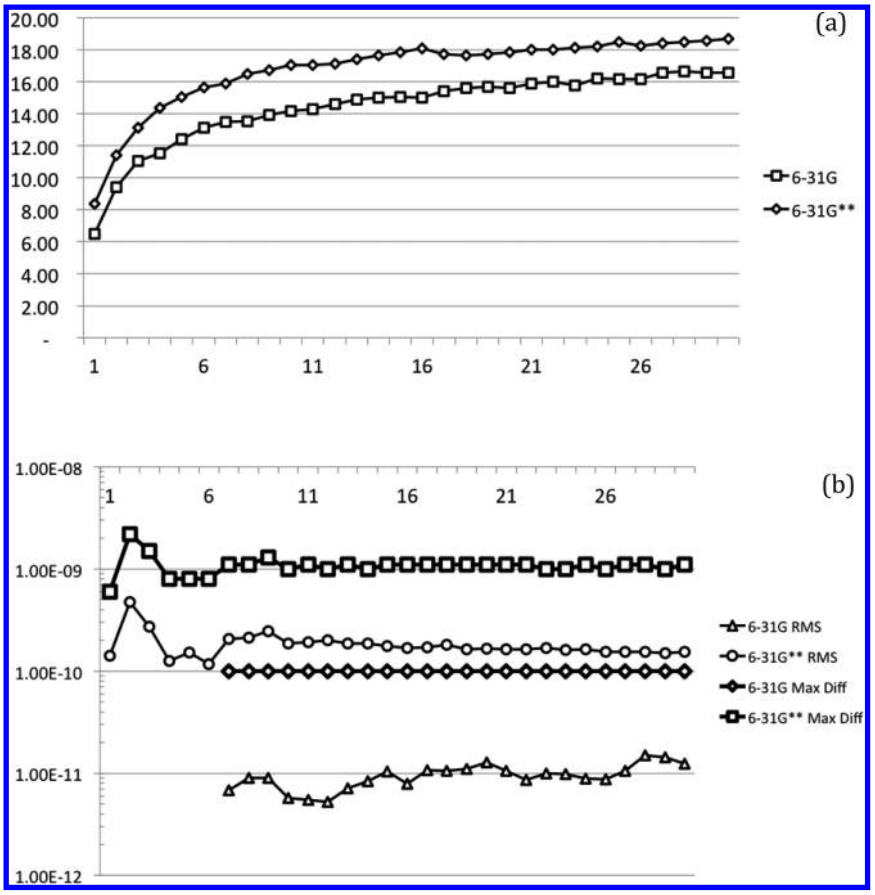


Figure 7. (a) Speedup comparison and (b) gradient deviation comparison between CPU and GPU calculations with different basis sets for alanine chain lengths from 1 to 30. Timing for CPU and GPU are based on the gradient generation time including data transfer but excludes the one-electron integral contribution. Platform and software details are described in text. For (b) RMSD and maximum difference (bolded) between the GPU and CPU result are presented. The first six points are not presented because the difference is smaller than 10^{-12}. A logarithmic scale is used for the *Y* axis.

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 kernels 1−8 account 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 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 four-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 7a by comparing the time used to evaluate the derivative ERI part of the gradient calculation. As Figure 7b 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 six 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 7a, we find that our GPU implementation speeds up the gradient calculation by as much as 19 times ($(\text{alanine})_{30}$ 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

DOI: 10.1021/ct500984t
J. Chem. Theory Comput. 2015, 11, 1449−1462

Table 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 (110)	6-31G/647	98.0			98.0	1345.8	**13.7**
	6-31G**/1160	379.7	18.7	33.8	432.1	6939.5	**16.1**
valinomycin (168)	6-31G/882	183.5			183.5	2584.3	**14.1**
	6-31G**/1620	691.0	35.8	57.6	784.4	12538.6	**16.0**
1AKG (209)	6-31G/1211	444.7			444.7	5729.6	**12.9**
	6-31G**/2171	1522.3	75.2	133.0	1730.5	25422.0	**14.7**
1CNL (169)	6-31G/991	304.0			304.0	3914.5	**12.9**
	6-31G**/1771	1078.6	49.8	90.3	1218.7	17861.0	**14.7**
1M2C (220)	6-31G/1268	477.7			477.7	6238.7	**13.1**
	6-31G**/2276	1677.2	86.2	146.9	1910.3	27997.6	**14.7**
1PEN (203)	6-31G/1192	423.7			423.7	5616.1	**13.3**
	6-31G**/2131	1500.2	74.3	133.0	1707.5	25548.4	**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.

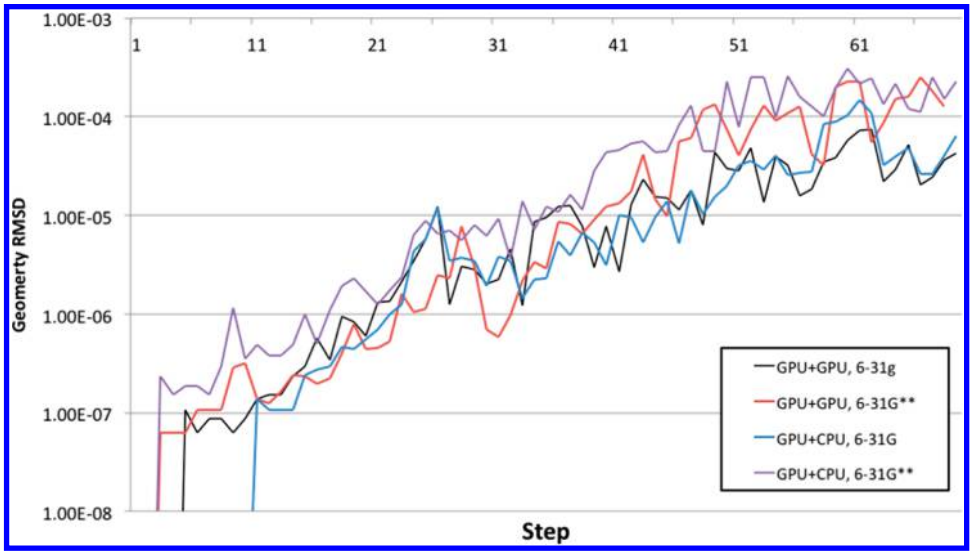


Figure 8. Accuracy of geometry optimization using a GPU versus a CPU. The test molecule was $(glycine)_{12}$ with the 6-31G and 6-31G** basis sets. Three sets of computations were executed including CPU + CPU, GPU + CPU, GPU + GPU. The curves indicate geometry RMSD relative to the CPU + CPU results at the same step. The starting geometry is optimized by GPU based calculation using the STO-3G basis set.

calculation. The results are summarized in Table 2. Generally, the extra calculation (kernels 1 and 2) 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 and 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 $(glycine)_{12}$ 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 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 1.80×10^{-6} and 3.21×10^{-6} for 6-31G and 7.46×10^{-6} and 4.28×10^{-7} for 6-31G**.

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 GB, for example, calculation of the $(alanine)_{28}$ chain using the 6-

DOI: 10.1021/ct500984t
J. Chem. Theory Comput. 2015, 11, 1449−1462

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.

5. CONCLUSIONS

In this paper, 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 presorting 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 $F_{(0-7)(0-7)}$ 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]

Our software can be freely download at http://www.merzgroup.org/quick.html under GPL (The GNU General Public License).

AUTHOR INFORMATION

Corresponding Author

*Phone: 517-355-9715. E-mail: kmerz1@gmail.com.

Notes

The authors declare no competing financial interest.

ACKNOWLEDGMENTS

This work is supported by The National Institues of Health (GM044974 and GM066859). We want to thank University of Florida High-Performance Computing Center and Institute for Cyber-Enabled Research (iCER) of Michigan State University. The authors also want to thank NVidia Cooperation for facility support.

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.; Elsevier, 2010; Vol. *6*, pp 21−35.
(13) Kulik, H. J.; Martinez, T. J. *Abstr. Pap.—Am. Chem. Soc.* **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. Soc.* **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. Soc.* **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. Soc.* **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.

DOI: 10.1021/ct500984t
J. Chem. Theory Comput. 2015, 11, 1449−1462

(25) Salomon-Ferrer, R.; Götz, A. W.; Poole, D.; Le Grand, S.; Walker, R. C. *J. Chem. Theory Comput.* **2013**, *9*, 3878−3888.
(26) Case, DA.; Darden, T. A.; Cheatham, T. E. I.; Simmerling, C.L.; Wang, J.; Duke, R. E.; Luo, R.; Walker, R. C.; Zhang, W.; Merz, K. M.; Roberts, B. P.; Wang, B.; Hayik, S.; Roitberg, A.; Seabra, G.; Kolossváry, I.; Wong, K. F.; Paesani, F.; Vanicek, J.; Liu, J.; Wu, X.; Brozell, S. R.; Steinbrecher, T.; Gohlke, H.; Cai, Q.; Ye, X.; Wang, J.; Hsieh, M.-J.; Cui, G.; Roe, D. R.; Mathews, D. H.; Seetin, M. G.; Sagui, C.; Babin, V.; Luchko, T.; Gusarov, S.; Kovalenko, A.; 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. Soc.* **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. *Proc. R. Soc. London. Ser. A.* **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 for 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) Laine, S.; Karras, T.; Aila, T. Megakernels Considered Harmful: Wavefront Path Tracing on GPUs. *High-Performance Graphics 2013*, Anaheim, CA, July 19−21, 2013.

DOI: 10.1021/ct500984t
J. Chem. Theory Comput. 2015, 11, 1449−1462