JCTC Journal of Chemical Theory and Computation

Article

pubs.acs.org/JCTC

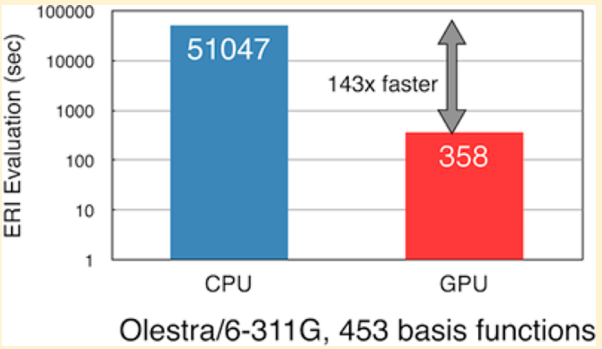
Acceleration of Electron Repulsion Integral Evaluation on Graphics Processing Units via Use of Recurrence Relations

Yipu Miao and Kenneth M. Merz, Jr.*

Department of Chemistry and Quantum Theory Project, 2328 New Physics Building, P. O. Box 118435, University of Florida, Gainesville, Florida 32611-8435, United States

Supporting Information

ABSTRACT: Electron repulsion integral (ERI) calculation on graphical processing units (GPUs) can significantly accelerate quantum chemical calculations. Herein, the *ab initio* self-consistent-field (SCF) calculation is implemented on GPUs using recurrence relations, which is one of the fastest ERI evaluation algorithms currently available. A direct-SCF scheme to assemble the Fock matrix 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. Realized speedups on GPUs reach 10−100 times relative to traditional CPU nodes, with accuracies of better than 1×10^{-7} for systems with more than 4000 basis functions.



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.

A GPU is an example of the single-instruction, multiple data (SIMD) paradigm, which, unlike CPUs that are designed for sequential code executing a single thread very quickly, has a parallel architecture that executes many concurrent threads comparatively 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. Recently, computational chemists have used GPUs extensively to treat a wide range of problems,[2] including molecular dynamics simulations (AMBER PMEMD),[3,4] quantum Monte Carlo,[5] DFT (density functional theory), SCF (self-consistent-field),[6−9] and post HF (Hartree−Fock) theory.[10−12] In this way, computationally intensive scientific applications, which previously required expensive supercomputing facilities, are now within reach using relatively low-cost GPU cards.

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 a 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.

Typically, for *ab initio* or DFT, Electron Repulsion Integral (ERI) calculations, together with basic linear algebraic operations, dominate the computational time. 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 prescreening, cutoffs, and efficient computational schemes. So,

Received: August 30, 2012

   dx.doi.org/10.1021/ct300754n | *J. Chem. Theory Comput.* XXXX, XXX, XXX−XXX

for most calculations with N less than several thousand basis functions, ERI evaluation is the most time-consuming part, while linear algebra, which scales cubically, will eventually dominate with very large numbers of basis functions. However, ERI evaluations are not as general as linear algebraic manipulations, and 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 will jeopardize computing efficiency. In addition, ERI evaluations require large amounts of both register and memory space. In GPU computation, slow memory access and register shortfalls are two factors that affect computational speed and should be considered during software development. Hence, careful software design and special treatments are necessary to accelerate ERI evaluation using a GPU.

In this paper, 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} au or better for moderate systems with thousands of basis functions, suitable for most applications. We have applied these ideas to 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 discussed in this section, where we show that systems with up to ∼10 000 basis function are feasible with a typical GPU. Finally, we conclude the paper with a brief discussion and conclusions.

2. ELECTRON REPULSION INTEGRALS

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

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

where the ϕ_μ are one-electron basis functions and $\mathbf{r}$ 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

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

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

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

where

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

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

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

ERIs possess 8-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, three $[ps|ss]$ 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 1000 basis functions). Once contracted ERIs are generated as given in eq 1, it is straightforward to form the Fock matrix with

$$\begin{aligned} F_{\mu\nu} &= H_{\mu\nu}^{\mathrm{core}} + \sum_{\lambda\sigma} P_{\lambda\sigma}\left[(\mu\nu|\sigma\lambda) - \frac{1}{2}(\mu\lambda|\sigma\nu)\right] \\ &= H_{\mu\nu}^{\mathrm{core}} + J_{\mu\nu} - \frac{1}{2}K_{\mu\nu} \end{aligned} \tag{6}$$

where $\mathbf{P}$ is the density matrix and $\mathbf{J}$ and $\mathbf{K}$ are the Coulomb and exchange matrices respectively.

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

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

dx.doi.org/10.1021/ct300754n | J. Chem. Theory Comput. XXXX, XXX, XXX−XXX

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

and α, β, γ, and δ 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 $[ss|ss]^{(m)}$ analytically:

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

where

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

$$F_m(T) = \int_1^0 t^{2m}\, \mathrm{e}^{-Tt^2}\, \mathrm{d}t \tag{14}$$

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

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

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

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 part 2, it may be applied to contracted ERIs.

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

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 the $(ab|cd)$ 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-ζ 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. IMPLEMENATION

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. GPUs are good examples of a massive parallel stream-processing architecture that uses the single-instruction multiple data (SIMD) model. Currently, the most widely used language environment for GPGPU technology is the 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 (SM) that consist of eight scalar processors (SP). 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.

Within the CUDA framework, a batch of threads is hierarchically arranged into a one-, two-, or three-dimensional grid of blocks up to 65 535 blocks, and each block of threads further consists of one-, two-, or three-dimensional grids. 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. Every 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 what happens sequentially, a phenomenon termed thread divergence. This 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 a SM (two for the Fermi architecture), multiple thread blocks can be active. For the memory hierarchies, each thread has access to its local register on the processor, and shared data in a block are 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.

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 the 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.

dx.doi.org/10.1021/ct300754n | J. Chem. Theory Comput. XXXX, XXX, XXX−XXX

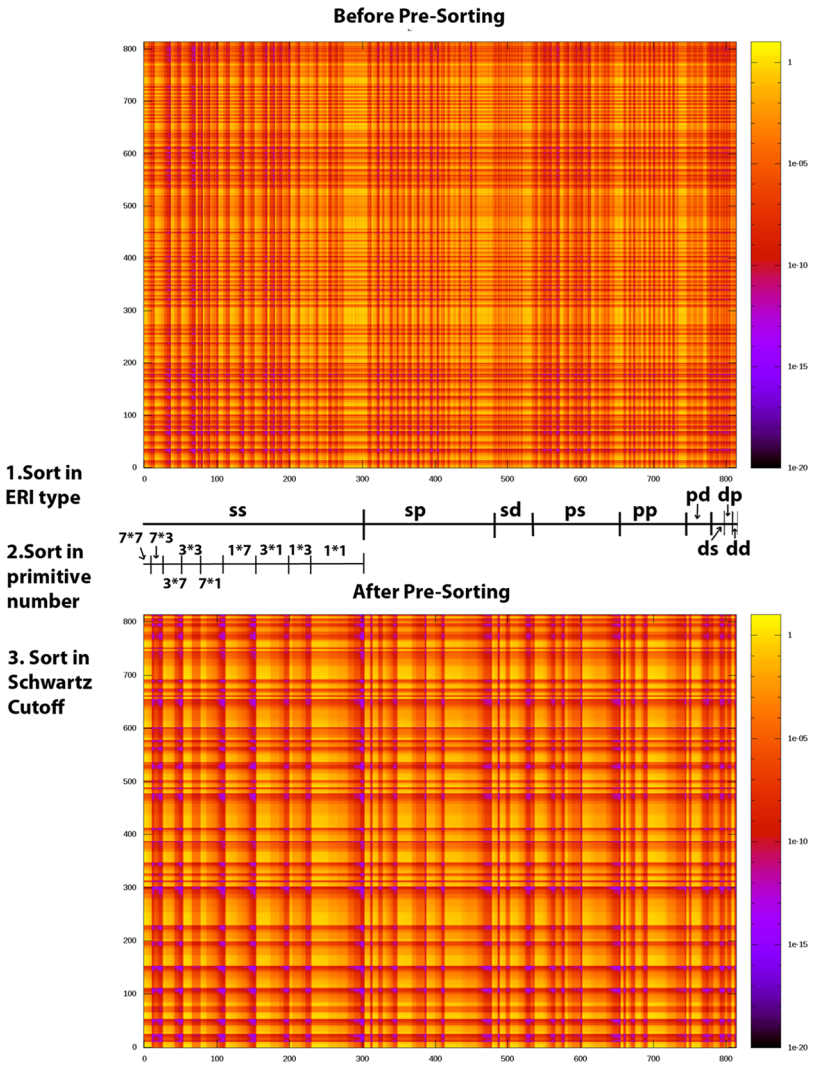


Figure 1. Presorting scheme for half ERIs. Row and column indices correspond to bra and ket pairs of contracted integrals, and the colors reflect the estimated magnitude of the ERI value. Sorting order is described in the text.

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 10-fold speedup for large General Matrix Multiply (GEMMs) over the INTEL MKL library.

3.2. Related Work. 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 on 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 the 6-31G basis set (168 atom, 882 basis sets) have been realized. Asadchev et al.[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 which has relatively few intermediate steps per integral and memory requirements, similar to the Rys quadrature.[18] HF SCF calculations with the 3-21G and 6-31G basis sets using their implementation and the NVIDIA GTX280 card[7−9] were, in some cases, more than 100 times faster than those using GAMESS[21] on a CPU. However, most of their comparisons were between single-precision GPU results 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]

dx.doi.org/10.1021/ct300754n | J. Chem. Theory Comput. XXXX, XXX, XXX−XXX

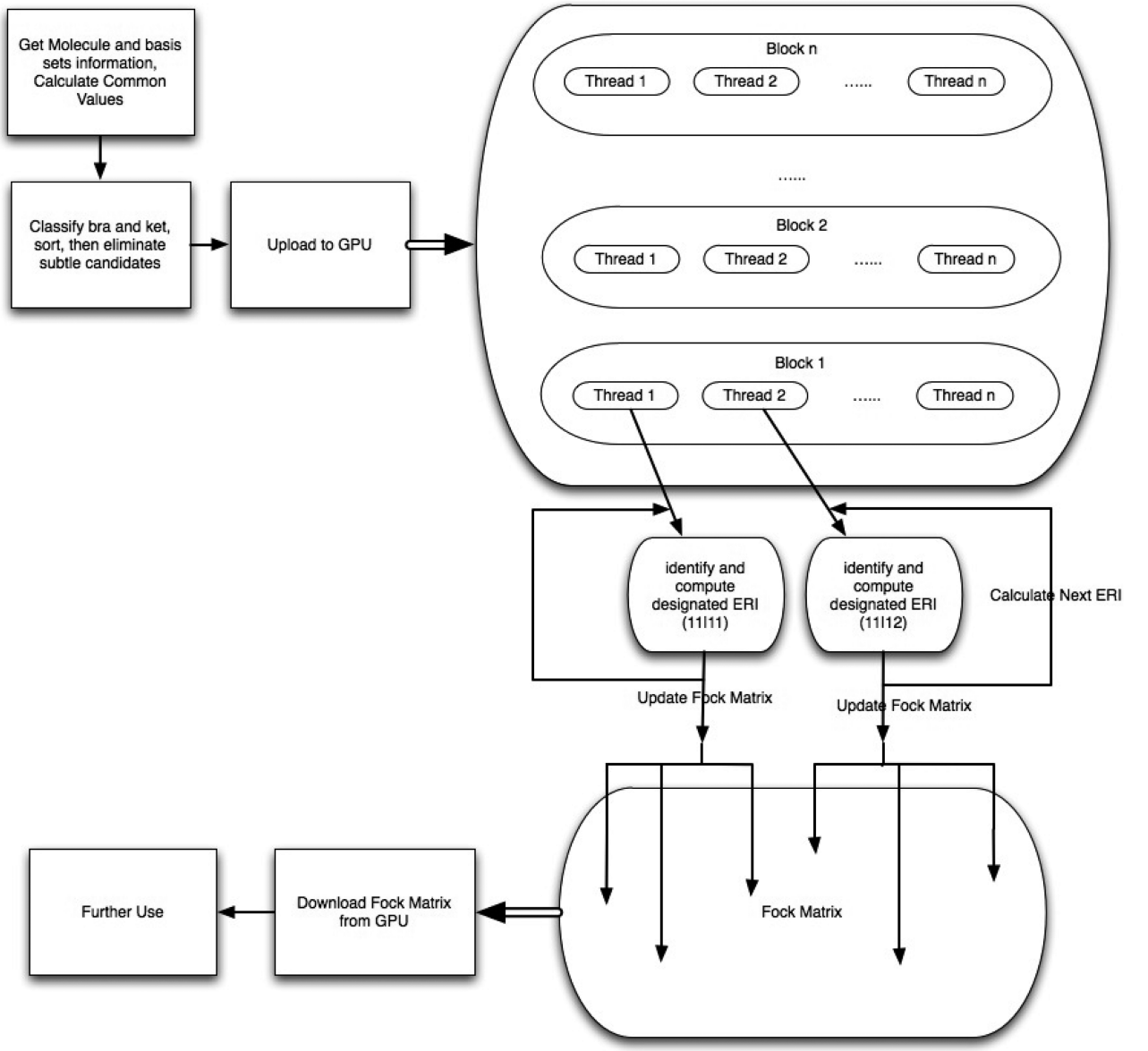


Figure 2. Flowchart for GPU implementation of ERI evaluation within an SCF cycle.

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, the GPU architecture is very different from that of a 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. 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 types and subsequently reorder them so that threads in one warp will likely undertake the same ERI or loop evaluation (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 1 illustrates our approach using a water cluster with four 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 the $|ab]$ half ERIs type, for instance, the highest angular momentum in this system with the cc-pVDZ basis set is a d orbital, so we have nine combinations as shown in the figure. 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

dx.doi.org/10.1021/ct300754n | J. Chem. Theory Comput. XXXX, XXX, XXX−XXX

Table 1. Timing for Two SCF Schemes for Hydrogen Atom System ERI Evaluation[a]

hydrogen number		direct SCF		conventional SCF		
		kernel time/s	GPU−CPU transfer/s	kernel time/s	GPU−CPU transfer/s	CPU time/s
8*8	STO-3G	0.183	0.062×10^{-3}	0.051	0.008	6.30
	6-31G	0.443	0.229×10^{-3}	0.127	0.086	14.97
10*10	STO-3G	0.454	0.049×10^{-3}	0.121	0.032	19.00
	6-31G	1.178	0.289×10^{-3}	0.340	0.447	47.40
12*12	STO-3G	0.918	0.101×10^{-3}	0.274	0.139	46.18
	6-31G	2.630	0.789×10^{-3}	0.770	1.942	121.83

[a]GPU timing refers to kernel time and GPU−CPU data transfer time. CPU time refers to the Fock matrix formation time. The unit is seconds.

while 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 the Cauchy−Schwarz inequality[24,25]

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

With this cutoff bound sparsely 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 be set as available for future computation. These three sorting orders create a well-balanced grid as shown in Figure 1 and guarantee that threads are kept busy at or near maximum usage.

Most streaming-type architectures, such as CUDA, do not provide efficient tools for interthread 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 (Figure 2). 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 ($\zeta = \alpha + \beta$, $P_i = ((\alpha A_i + \beta B_i)/(\alpha + \beta))$ and $K_{AB} = \sqrt{2}((\pi^{5/4})/(\alpha + \beta))\exp[-((\alpha\beta)/(\alpha + \beta))(\vec{A} - \vec{B})^2])$ are all precalculated 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 for speed. With these precalculated values, both DRAM data access count and floating-pointing count are 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 recalculated 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 the GPU kernel time,[7] and similar validation will be shown here. 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 beginning with 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 eight times if i, j, k, and l are not equal to each other.

One concern for the atomic function is that 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 arises 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, and 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.

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 eight times at most, then 8192 integrals are calculated in one loop and a maximum value of 65 536 memory addresses are requested to be modified. The Fock matrix has $N*N$ elements where N is the 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 are inactive due to a 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 a neighbor distance of 1 Å. The systems used here are "toy" models because only s orbitals are required, which excludes the overhead brought by thread convergence. Table 1 shows the timing results produced by the two methods, conventional SCF and direct SCF. All these results are based on 64-bit double-precision computation on an NVIDIA M2090 GPU 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 1 is the impressive GPU speedup, 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 a 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% of 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 the memory requirement scale as 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 to abandon sophisticated code optimized for CPUs over years in order to avoid expensive copy operations.

4. RESULTS AND DISCUSSION

Before we began tests on our GPU code, we implemented the same recurrence relations for a CPU in QUICK.[27] Our test indicates it has competitive efficiency with GAMESS, as shown in Table 2. The GAMESS version used was the 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 2427 CPU, and we list their first iteration times in Table 2. QUICK performs competitively in most categories; therefore, we conclude that QUICK's implementation is as efficient as a standard quantum chemistry package.

Table 2. Time Comparisons between QUICK and GAMESS[a]

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

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 2- to 4-fold speedup for split Gaussian basis sets. In the following benchmark test, both CPU and GPU codes use the direct SCF procedure with the two-electron integral cutoff set as starting from 10^{-9} and will be changed to tighter criteria after reaching a convergence threshold. A Schwarz upper bound cutoff is used to prescreen small ERI blocks.

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 $O(n*\log n)$, and in our case, $n = N^2$, where N is the number of unique shells therefore, the scale of sorting is $O(N^2* \log N)$, which is smaller than the ERI scale and linear algebra; therefore, sorting will not be a bottleneck in terms of scaling. In the benchmark, we compare performance on one typical CPU processor used in high-performance computation (AMD Opteron 2427) and one of the state-of-the-art GPUs (TESLA M2090, with ECC off). An M2090 card has a specification of 6 GB RAM, with 16 streaming multiprocessor and 512 CUDA cores. We set the block number equal to the number of stream multiprocessors in device, 16 in this case. Timing is set for the Fock matrix formation time including CPU−GPU data transfer except when mentioned otherwise.

The first set of test cases was linear $(alanine)_n$ chains. We chose different Gaussian split valence basis sets varying from 3-21G to 6-311G**, and the chain length was 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, while we compare the energy deviation of the GPU relative to the CPU in the first iteration in Figure 4. From Figure 3, an obvious trend is that the speedup increase is directly related to the system size; for example, for the 6-31G* basis set, GPU

dx.doi.org/10.1021/ct300754n | J. Chem. Theory Comput. XXXX, XXX, XXX−XXX

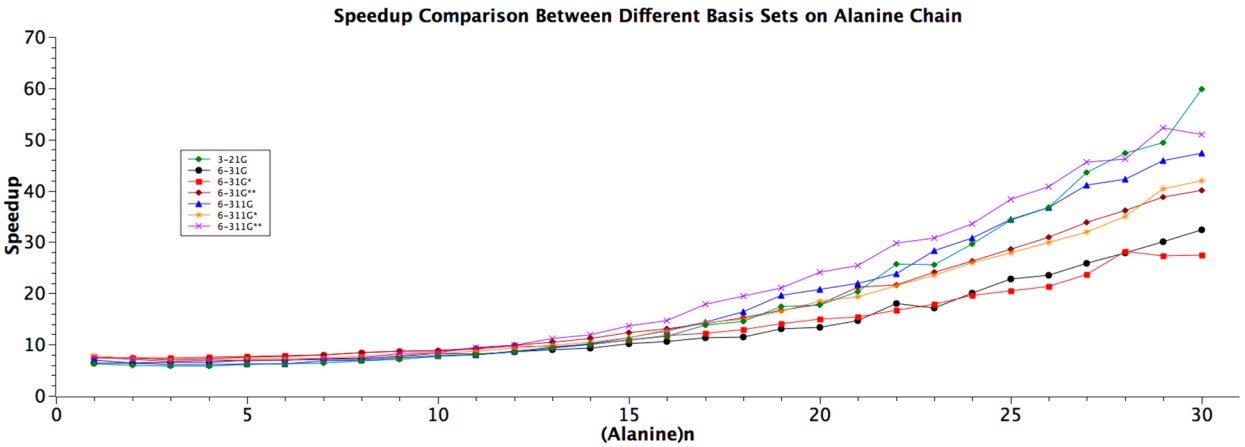


Figure 3. Speedup comparisons between different basis sets on alanine chain series. Timings for the CPU and GPU are based on the Fock Matrix iteration including data transfer time. Platform and software details are described in text.

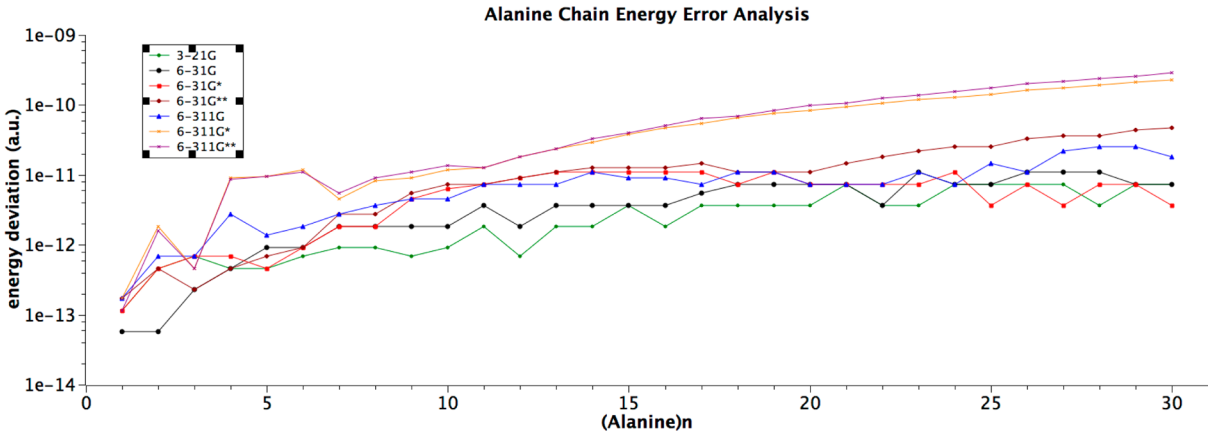


Figure 4. Energy deviation comparisons between different basis sets on alanine chain series. A logarithmic scale for the *Y* axis is used.

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 a GPU, so if a larger number of basis functions is used, there is less chance that two threads modify the same address when they finish ERI evaluation. In contrast, if the density matrix size is small, that leads to some threads adopt idle states which slows calculation speed. The third reason is because of the prescreen strategy (including the Schwarz cutoff) used in the GPU 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 that doubly-polarized basis sets are generally most accelerated and their speedup increases fastest, while nonpolarized 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 singly-polarized basis sets. This is because polarized basis sets will have fewer integrals cut off but possess a large number of basis functions, which takes 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 the single-polarized basis set, this is switched.

In Figure 4, energy differences are shown. Since all of the calculations are double precision except for atomic operations, the only error beyond numerical error is brought on by atomic operations. We found that all of 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 the accuracy of double-precision . 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 $n^{2.5-3.0}$; 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 10 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 approximately 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} au accuracy because of error

dx.doi.org/10.1021/ct300754n | J. Chem. Theory Comput. XXXX, XXX, XXX−XXX

Table 3. Accuracy and Performance Comparison between CPU and GPU Calculation[a,b]

molecule (atom number)	basis sets (function number)	first iteration/s			last iteration/s			energy	
		CPU	GPU	speedup	CPU	GPU	speedup	CPU/au	GPU/ *10^{-9} au
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
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 (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 (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
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 the text. [b]PDB ID and number of positive and negative charge center included in parentheses.

accumulation. The error order we achieve, 10^{-10} Hartrees, is accurate enough for most chemistry calculations, especially considering the accompanying speedup.

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 midsize systems like taxol, valinomycin, olestra, and proteins with PDB[28] IDs of 1M2C, 1OMG, and 1VTP. 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 of 10^{-7}, and an

dx.doi.org/10.1021/ct300754n | J. Chem. Theory Comput. XXXX, XXX, XXX−XXX

Table 4. Timings for GPU 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 gives the Fock matrix formation time in the GPU in seconds. [b]Global memory size (also know as DRAM) included in parentheses. [c]CPU calculation is on an AMD Opteron 2427 CPU.

Table 5. Peak Memory Usage Comparison[a]

		6-31G		6-31G*		6-31G**		6-311G		6-311G*		6-311G**	
molecule	number of atoms	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	674 MB	3418	880 MB	4000	1049 MB	4420	963 MB	5002	1224 MB	4000	1427 MB
crambin	642	3597	1796 MB	5559	2338 MB	6509	2785 MB	5244	2589 MB	7206	3279 MB	8151	3818 MB
1BQ9	765	4318	2570 MB	6694	3357 MB	7801	3985 MB	6287	3687 MB	8663	4686 MB	9770	5441 MB

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

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 the later 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. We list first iteration time and last iteration time and compare the converged total energy deviation between the CPU and GPU results. From the table, we observe that a single GPU can speed up 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 the 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 cut off. We also find that the olestra molecule and water cluster $(H_2O)_{100}$ yields impressive speedups, about 100 times faster for 6-31G, due to the large amount of hydrogen atoms, where a large portion of the integrals are of the (*ssIss*) type, which requires the least computational time and memory access. According to Table 1, for a hydrogen mesh with the 6-31G basis set, in the conventional SCF scheme, the kernel time in GPU versus CPU can reach a 150 times speedup, which is the maximum speedup a GPU of the type employed can achieve. Therefore, the speedups achieved for olestra and the 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 the 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), scales cubically with the number of basis functions (these calculations cost more than ERI computation) for large systems, this 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 371.51 s (37.7% of Fock matrix formation time), while diagonalization is 323.68 s (32.9% of Fock matrix formation time) and the DIIS time (excluding the diagonalization time) is 289.45 s (29.38% of Fock matrix formation time), while the ERI evaluation and DIIS times for the CPU were 50 009.94 and 16 079.17 s. 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. The timings for the GPU version calculation using different devices are given in Table 4.

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 5. Memory usage is in a tradeoff with speed as described above by storing precalculated values and they, together with the necessary molecular basis set and electron structure information such as the 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 system sizes achieved about 10 000 basis sets because of well-designed prescreening strategies. We could avoid the precalculation 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 speedup is not significantly different (∼10−20%) when lower level cards were used (M2070 and a GTX580 for example), as shown in Table 5. For some small basis sets, 6-

dx.doi.org/10.1021/ct300754n | J. Chem. Theory Comput. XXXX, XXX, XXX−XXX

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 Tesla or better cards for their stability and fault tolerance.

5. CONCLUSIONS

In this paper, 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} au relative to the CPU result, which meets most 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 required in chemistry 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. Moreover 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 intio* 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 a 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 a GPU is still an open question. In addition, higher angular momentum ERI evaluation is under development.

■ ASSOCIATED CONTENT

Supporting Information

Further calculation details such as test molecule coordinates in the text. This information is available free of charge via the Internet at http://pubs.acs.org/.

■ AUTHOR INFORMATION

Corresponding Author

*Phone: 352-392-6973. Fax: 352-392-8722. E-mail: merz@qtp.ufl.edu.

Notes

The authors declare no competing financial interest.

■ ACKNOWLEDGMENTS

This work was supported by the NSF (MCB-0211639). We also thank the University of Florida High-Performance Computing Center for providing facility and technical support.

■ 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*; Wheeler, R. A., Ed.; Elsevier: Amsterdam, 2010; Vol. 6, pp 21−35.

(3) Case, D. A.; Darden, T. A.; Cheatham, T. E.; 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. *AMBER 11*; University of California: San Francisco, CA, 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. *Proc. R. Soc. London, Ser. A* **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. *Lect. Notes Quant. Chem. II* **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) Miao, Y.; He, X.; Ayers, K.; Brothers, E.; Merz, K. M. *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.