JCTC Journal of Chemical Theory and Computation

Quantum Chemistry on Graphical Processing Units. 1. Strategies for Two-Electron Integral Evaluation

Ivan S. Ufimtsev and Todd J. Martínez*

Department of Chemistry and The Beckman Institute, 600 S. Mathews, University of Illinois, Urbana, Illinois 61801

Received October 13, 2007

Abstract: Modern videogames place increasing demands on the computational and graphical hardware, leading to novel architectures that have great potential in the context of high performance computing and molecular simulation. We demonstrate that Graphical Processing Units (GPUs) can be used very efficiently to calculate two-electron repulsion integrals over Gaussian basis functions—the first step in most quantum chemistry calculations. A benchmark test performed for the evaluation of approximately 10^6 $(ss|ss)$ integrals over contracted *s*-orbitals showed that a naïve algorithm implemented on the GPU achieves up to 130-fold speedup over a traditional CPU implementation on an AMD Opteron. Subsequent calculations of the Coulomb operator for a 256-atom DNA strand show that the GPU advantage is maintained for basis sets including higher angular momentum functions.

1. Introduction

The past decade has seen a tremendous increase in the computing requirements of consumer videogames, and this demand is being met through novel hardware architectures in the form of proprietary consoles and graphics cards. Offerings such as the Sony PlayStation 3 (designed around IBM's Cell processor[1]) and the nVidia GeForce 8800 GTX graphics card are excellent examples, both of which may be characterized as stream processors.[2] Stream processing is a generalization of the single instruction multiple data (SIMD) vector processing model which formed the core of the Cray-1 supercomputer.[3] Applications are organized into streams and kernels, representing blocks of data and code transformations, respectively. The kernel is typically comprised of a tight loop of relatively few instructions. Streams of data are then processed in pipelined and parallel fashion by many processors executing a small number (possibly only one) of kernels. In the case of the nVidia 8800 GTX, there are 128 total processors organized as 16 multiprocessor units comprised of 8 processing units each. These run at a clock speed of 1.35 GHz, which is comparable to the conventional CPUs commonly used as the basis for scientific computing clusters.

Since a graphics card typically costs less than a single CPU used in conventional scientific clusters, it is tempting to consider the use of graphics cards for computational chemistry. The earliest attempts to use graphics processing units (GPUs) for nongraphical computing in fields outside of chemistry[4−6] were largely stymied by limited precision and difficulty of programming. The former problem has been partially remedied, and the latest GPUs support 32-bit floating point arithmetic. The next generation of GPUs and stream processors from nVidia and AMD have already been announced and will extend this support to 64-bit. The latter problem of programming difficulty has been largely removed by nVidia's recent introduction of the Compute Unified Device Architecture (CUDA), which provides a relatively simple programming interface that can be called from the standard C language. A few groups have recognized the potential of GPUs in the context of computational chemistry,[7−9] with some recent implementations within the CUDA framework.[8−10] Much of the focus has been on questions of accuracy associated with 32-bit single precision arithmetic, but we discuss this only very briefly here since the precision problem will be much less important with the advent of 64-bit GPUs and stream processors. Instead, our paper focuses

* Corresponding author e-mail: tjm@spawn.scs.uiuc.edu.

10.1021/ct700268q CCC: $40.75 © 2008 American Chemical Society
Published on Web 01/25/2008

on the implementation of quantum chemistry algorithms in the context of stream processors.

In this paper, we propose and test three different algorithms to solve one of the bottleneck problems of most ab initio calculations, the two-electron integral evaluation part, entirely on the GPU. Yasuda has recently demonstrated that that GPU evaluation of two-electron integrals is feasible with up to 10× speedups compared to conventional CPUs.[9] He introduced a novel scheme that calculates the largest integrals on the CPU in double precision and others on the GPU in single precision. In contrast, we explore several computational organizations for the problem, determining which are most appropriate for the GPU architecture. We formulate three different approaches, each with its own strong and weak points. By taking the architectural details of the GPU into account, we are able to achieve speedups of more than 100× compared to mature algorithms on conventional CPUs. One of the algorithms is particularly suitable for direct SCF methods,[11] where the integrals are recomputed every SCF iteration, while the others are better for conventional SCF methods, where primitive integrals have to be contracted and stored (usually on disk) before the SCF procedure starts.

To assess the relative performance of these three algorithms, we have chosen a relatively simple test system consisting of 64 H atoms organized on a 4 × 4 × 4 cubic lattice with nearest-neighbor spacing of 0.74 Å using the STO-6G and 6-311G basis sets. In this test system, only $(ss|ss)$ type integrals need to be evaluated. Having identified the algorithm which is most suitable for direct SCF calculations, we then use it to construct (entirely on the GPU) the Coulomb contribution to the Fock matrix (the J-matrix) for a much larger system including both s- and p-type basis functions—a 256-atom DNA strand using the 3-21G basis set (1699 basis functions). Comparison of the corresponding CPU and GPU timings confirms that the GPU architecture is well-suited for use in quantum chemistry calculations. The algorithms presented here, and to a large extent also the code, will be directly applicable to double-precision GPUs and stream processors with little or no modification.

This paper is organized in the following way. Section 1 is the Introduction; section 2 outlines the problem background; section 3 provides a brief overview of the GPU architecture and some programming basics required for understanding the further material; the integral computation algorithms on GPU are described in section 4; and section 5 includes the benchmark timing results as well as a brief discussion concerning the impact of 32-bit precision in the GPU calculations.

2. Two-Electron Repulsion Integrals

The first step in any ab initio Molecular Orbital (MO) or Density Functional Theory (DFT) treatment of electronic structure is the evaluation of a large number of two-electron repulsion integrals over N atom-centered one-electron basis functions φ

$$(pq|rs) = \int\int \varphi_p(\vec{r}_1)\varphi_q(\vec{r}_1)\frac{1}{|\vec{r}_1 - \vec{r}_2|}\varphi_r(\vec{r}_2)\varphi_s(\vec{r}_2)d\vec{r}_1 d\vec{r}_2 \quad (1)$$

where $\vec{r}$ refers to the electronic coordinates. In practice, these basis functions are typically linear combinations of primitive atom-centered Gaussian basis functions:

$$\varphi_p(\vec{r}) = \sum_{k=1}^{K} c_{pk}\chi_k(\vec{r}) \quad (2)$$

The primitive basis functions χ are centered at the coordinates $\vec{R}_A = (X_A, Y_A, Z_A)$ of the Ath nucleus:

$$\chi(\vec{r}) = N(x - X_A)^{n_x}(y - Y_A)^{n_y}(z - Z_A)^{n_z}\exp(-\alpha|\vec{r} - \vec{R}_A|^2) \quad (3)$$

The two-electron integrals in the contracted basis are thus evaluated as

$$(pq|rs) = \sum_{k_1=1}^{K_p}\sum_{k_2=1}^{K_q}\sum_{k_3=1}^{K_r}\sum_{k_4=1}^{K_s} c_{pk_1}c_{qk_2}c_{rk_3}c_{sk_4}[\chi_{k_1}\chi_{k_2}|\chi_{k_3}\chi_{k_4}] \quad (4)$$

where we use brackets to denote two-electron integrals over primitive basis functions and parentheses to denote such integrals over contracted basis functions. The angular momentum of the basis functions is given by the sum of the three integer parameters, $\lambda = n_x + n_y + n_z$, in the usual way. The primitive integrals can be evaluated analytically as originally shown by Boys.[12] Since Boys' seminal work, numerous computational approaches have been developed to minimize the effort in this N^4 bottleneck.[13−17] However, even if the most efficient algorithm is being used, the two-electron integral evaluation phase still takes much of the computation time.

3. Overview of GPU Hardware and CUDA API

All calculations throughout this project were performed on one nVidia GeForce 8800 GTX graphical processor running under the Windows XP operating system. The Compute Unified Device Architecture (CUDA) Application Programming Interface (API) provided by nVidia[18] was used to develop the GPU-side code. Perhaps the most detailed descriptions of the nVidia GeForce GPU architecture and the CUDA API are provided in the CUDA Programming Guide available for download free of charge.[19] We briefly outline some features of the hardware and programming models that are needed to understand our implementation of two-electron integral evaluation.

In the beta version of the CUDA implementation which we used for this work, all GPU functions (those functions which are executed on the GPU, not on the CPU) are called synchronously. In other words, when the CPU reaches the point where a GPU function ("kernel") is called, the CPU waits until this function returns and only then can proceed further.[20] From this point of view, the GPU can be considered as a coprocessor to the CPU—a fast processor that is responsible for executing the most computationally intensive parts of a program which can be efficiently processed in parallel. The processors of the GPU are not able to access CPU memory directly. Therefore, before a GPU kernel is executed, the CPU (using functions provided by the CUDA host runtime component) must copy required data from CPU memory to GPU memory. Likewise, if desired, the results

in GPU memory may be copied to CPU memory after GPU processing is completed.

The GeForce 8800 is able to process a large number of parallel threads. Within the CUDA framework, the whole batch of threads is arranged as a one- or two-dimensional grid of blocks with up to 65 535 blocks in each dimension. Each block of threads can be one-, two-, or three-dimensional depending on the problem being solved. The number of threads in a block must be specified explicitly in the code and should not be more than 512 in the current CUDA implementation. The best performance is obtained if the number of threads in a block is a multiple of 32, for scheduling reasons discussed below. The CUDA framework assigns a unique serial number (*threadIdx*) to each thread in a block. Likewise, each block of threads is assigned a unique identification number (*blockIdx*). For a two-dimensional grid of blocks, *blockIdx* consists of two numbers, $blockIdx_x$ and $blockIdx_y$. Using both *threadIdx* and *blockIdx*, one can completely identify a given thread. This makes it possible for each thread to identify its share of the work in a Single Program Multiple Data (SPMD) application.

The GeForce 8800 GTX consists of 16 independent stream multiprocessors (SM). Each SM has a Single Instruction Multiple Data (SIMD) implementation with eight scalar processors and one instruction unit. At each clock cycle, the instruction unit of an SM broadcasts the same instruction to all eight of its scalar processor units, which then operate on different data. Each SM can process several blocks concurrently, but all the threads in a given block are guaranteed to be executed on a single SM. Threads within the same block are thereby able to communicate with each other very efficiently using fast on-chip shared memory and are furthermore able to synchronize their execution. In contrast, threads belonging to different blocks are not able to communicate efficiently nor to synchronize their execution. Thus, interblock communication must be avoided in an efficient GPU algorithm.

Since the number of threads in a typical GPU application is much larger than the total number of scalar processing units, all the threads are executed using time slicing. All blocks are split into 32-thread groups (which is why the number of threads in a block should be a multiple of 32) called warps. Each warp is then processed in SIMD fashion (all 32 threads are executed by 8 processor units in 2 clock cycles). The thread scheduler periodically switches the active warps to maintain load balancing, maximizing the overall performance.

4. GPU Algorithms for Two-Electron Integrals

Most parallel programs use one of two general organizational schemes: the master−slave model or the peer model. In the master−slave model, there is one master node which executes a common serial program but distributes the most computationally intensive parts among the slave nodes. After the slave nodes are done, the master node gathers the results and uses them in further computations. In the case of the two-electron integral evaluation problem, a common implementation[21−23] of the master−slave model is as follows: the master node loops over all atomic orbitals, generating lists of $(pq|rs)$ integrals (i.e., index ranges) and the required input data (exponents, contraction coefficients, and atomic coordinates). These lists are sent to the slave nodes, which then evaluate the corresponding integrals. In the peer model, there is no master node, and all computational nodes execute the same program.

There are two levels of parallelism in a typical GPU code. The first is between the CPU and the GPUs, which is handled in the master−slave model. The CPU is the master node which calls one or more GPUs as slave nodes. The second level of parallelism is within the GPU itself, which is implemented in the peer model, where each thread executes the same program and must use its *threadIdx* and *blockIdx* to determine precisely what work it is to perform.

Because of the relatively slow 2Gb/s transfer speeds between the CPU and GPU, it is important to avoid CPU-GPU data transfer as much as possible. Below, we present several algorithms for two-electron integral evaluation. We test them on a simple system consisting of 64 H atoms. Finally, we show that these algorithms preserve their efficiency for much more complex systems, specifically a 256-atom DNA strand, containing 685 contracted *s*- and 1014 *p*-type basis functions.

4a. Mapping Integrals to the GPU Threads. The mapping procedure starts by enumerating all the N atomic orbitals φ_p in the system (p=1...N) and then constructing corresponding *bra*- and *ket*-arrays $\varphi_p\varphi_q$ of length $M = N(N + 1)/2$. The two-electron integrals can then be generated as $(pq|rs)$ *bra*- and *ket*-vector element combinations. This is schematically represented in Figure 1. Each light green square represents one $(bra|ket)$ integral out of M^2 total integrals. The $(bra|ket)$=$(ket|bra)$ symmetry reduces these M^2 integrals to the final set of $M(M + 1)/2$ unique two-electron integrals represented by the upper-right triangle submatrix in Figure 1. Several different integral↔GPU thread mappings can be envisioned. The two-dimensional square grid of contracted integrals in Figure 1 can be naturally mapped to a two-dimensional grid of GPU thread blocks. In this case, there are two possibilities—either each block of GPU threads calculates a contracted integral as depicted in Figure 1, or each GPU thread calculates a contracted integral as depicted in Figure 2. In the first case, the primitive integrals contributing to the same contracted integral are cyclically mapped to the threads of a block (the predefined number of threads in a block is the same for all blocks). Following the calculation of the primitive integrals, a block sum reduction performs the final summation leading to the contracted integral. In the second case, each individual GPU thread calculates its own contracted integral by looping over all contributing primitive integrals and accumulating the result in a local variable. A third possible mapping is shown in Figure 3, where each thread calculates just one primitive integral, no matter to which contracted integral it contributes. In this case, additional overhead is needed to perform a series of local sum reductions converting the grid of primitive integrals to the grid of contracted integrals.

These three approaches cover a wide range of possible mapping schemes. The third mapping (one thread ↔ one primitive integral) is very fine-grained with perfect computa-

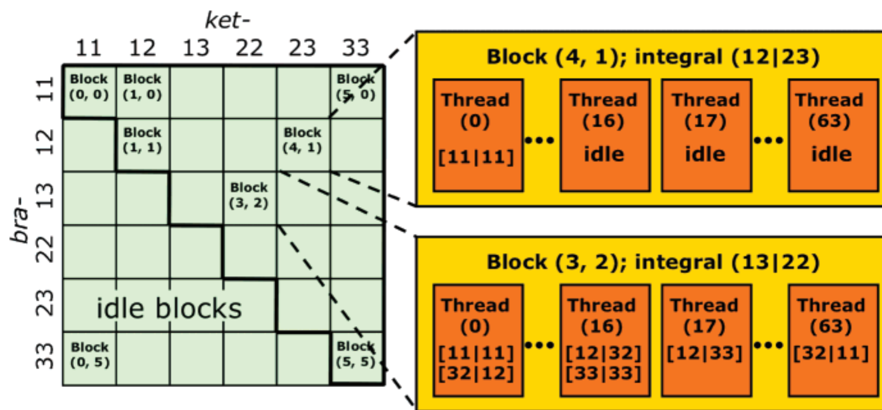

Figure 1. "One Block ↔ One Contracted Integral" (1B1CI) mapping. The green squares represent the contracted integrals as well as one-dimensional 64-thread blocks mapped to them. To the right, the GPU thread to primitive integral mapping is illustrated for two contracted integrals containing 1 and $3^4 = 81$ primitive integrals. After all the primitive integrals are calculated, a block sum reduction leads to the final result.

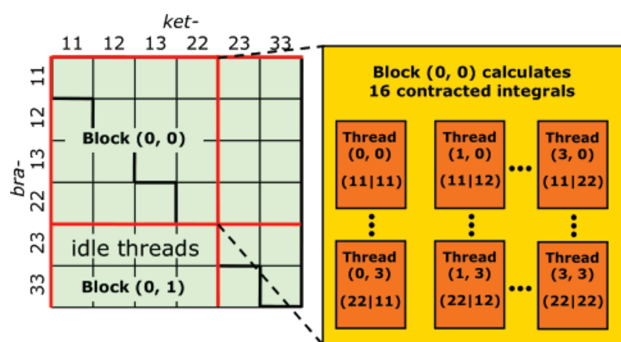

Figure 2. "One Thread ↔ One Contracted Integral" (1T1CI) mapping. Green squares represent the contracted integrals as well as individual GPU threads mapped to them. A two-dimensional 4 × 4 thread block is outlined in red. Each thread calculates its integral by looping over all primitive integrals and accumulating the result in a local variable.

tion load balancing but relatively large data reduction overhead. In contrast, the second mapping (one thread ↔ one contracted integral) is very coarse-grained with imperfect load balancing. This is because different threads can have different loop lengths, depending on the degree of contraction of each of the four basis functions in the integral. On the other hand, this mapping has no data reduction overhead (summation of data held in different threads). The first mapping, (one thread block ↔ one contracted integral) is intermediate in terms of the grain of parallelism. Load imbalancing can decrease the performance significantly, for example in basis sets with low average degree of contraction. However, the load is balanced more effectively than in the second mapping. Additionally, the data reduction overhead is small because the threads that need to communicate are all located in the same thread block (and hence reside on the same SM).

We have tested all these three approaches on a system of 64 hydrogen atoms using the STO-6G[24,25] and 6-311G[26] basis sets, representing highly contracted and relatively uncontracted basis set cases, respectively.

4b. One Thread Block ↔ One Contracted Integral Mapping. The "One Block ↔ One Contracted Integral" mapping (1B1CI) is schematically represented in Figure 1. The green squares represent the contracted integrals as well as the blocks of computational GPU threads mapped to them. Because of (*bra*|*ket*)=(*ket*|*bra*) symmetry, those integrals lying below the main diagonal should be disregarded. This is easily done with a logic statement at the beginning of thread execution. If the thread is assigned to an integral in the lower triangle, it simply exits without computing anything—this is indicated in Figure 1 by the designation "idle blocks". This "outscheduling" has little effect on performance since the scheduler switches between GPU warps very quickly (once all threads in a warp have completed processing, they are removed from the scheduling list and do not impose any load balancing penalty). After each contracted integral is mapped to the corresponding block of GPU threads, the primitive integrals contributing to the particular contracted integral are assigned to the threads constituting this block. Different schemes can be used here—in our program we use a cyclic mapping to a one-dimensional block of 64 threads (orange rectangles in Figure 1). Each successive term in the sum of eq 4 is then mapped to a successive GPU thread, i.e., [11|11] to thread 0, [11|12] to thread 1, and so on. If the number of primitive integrals is larger than the number of GPU threads in the block, the procedure repeats: the 65th primitive integral is mapped to thread 0, the 66th to thread 1, and so on until all primitive integrals have been assigned to a GPU thread. Depending on the number of terms in eq 4 for the contracted integral under consideration, two situations are possible as shown on the right in Figure 1. Block (4,1) represents the case when some threads have no integrals mapped to them. This can happen, for example, when there is only one term in the sum of eq 4, i.e. the contraction length for all basis functions involved in the integral is unity. Since the number of threads in a block is fixed and is the same for all the blocks, the number of threads in Block (4,1) will be 64, of which only one will do useful work—the others will just waste the computational resources executing unnecessary instructions. Thus, the 1B1CI mapping is more efficient for highly contracted basis sets. Direct computational tests confirm this conclusion and show that for low-contracted basis sets the performance drops by a factor of 2−3. Note that the

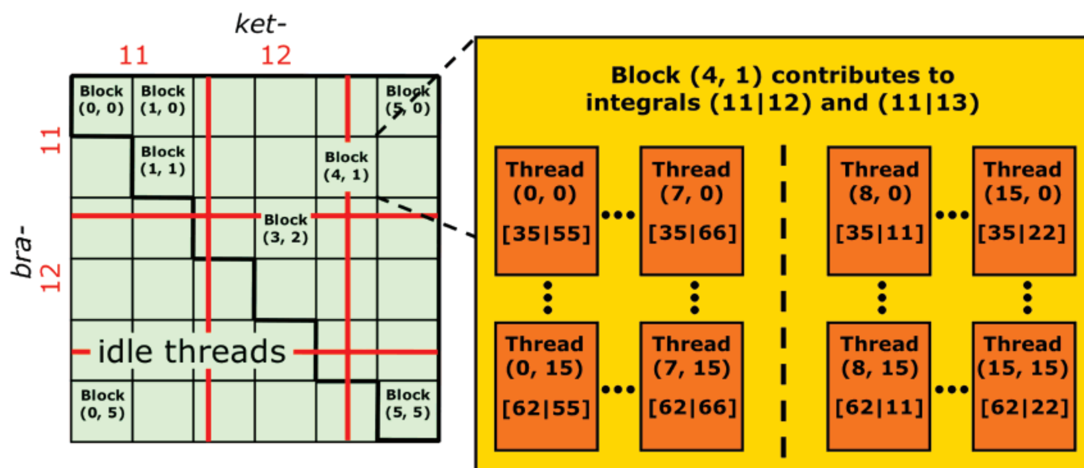

Figure 3. "One Thread ↔ One Primitive Integral" (1T1PI) mapping. The two-dimensional 16 × 16 thread blocks are represented by green squares, while the red squares represent contracted integrals. To the right, we show an example where the primitives calculated by one thread block contribute to more than one contracted integral.

performance penalty is less than might have been expected. This is partly because of the efficient scheduling and organization of threads into warps as discussed above. When all of the threads in a warp are idle, the entire warp wastes only one clock cycle, after which the GPU "outschedules" the warp, i.e., it is removed from consideration for further scheduling. Block (3, 2) represents the case when the number of primitive integrals is not a multiple of the number of the number of threads in a block. In this case, there is also performance degradation since threads 17−31 are only calculating one primitive integral, while threads 0−16 calculate two primitive integrals. However, in general the effect is much smaller than in the previous case. Again, the efficient scheduling of the GPU and organization of threads into warps is the reason for relatively minor effects of load imbalance. Given the current lack of performance monitoring tools for the GPU, the only way to assess the impact of load imbalancing is by direct experimentation for realistic test cases.

4c. One Thread ↔ One Contracted Integral Mapping. The "One Thread ↔ One Contracted Integral" (1T1CI) mapping is shown schematically in Figure 2. Again, the green squares represent the contracted integrals, while the blocks of threads are sketched in red. In contrast to the previous model, the blocks of threads are now two-dimensional. Figure 2 shows a block of 16 threads arranged as a 4 × 4 grid for illustrative purposes, while in our test calculations we found the 16 × 16 configuration to be optimal. Each thread now has its own contracted integral to calculate. It does this by looping over all primitive integrals and accumulating the result. Figure 4 presents a detailed flowchart of the procedure. As mentioned above, the 1T1CI mapping scheme can suffer from load balancing problems since there may be a different number of terms in each of the sums of primitive integrals. This results in a different number of loop cycles needed to calculate each contracted integral and a corresponding imbalance in the computational work per thread. The effect of this load imbalance can be minimized by organizing the contracted integrals into subgrids, so that each subgrid contains contracted integrals involving the same number of primitive integrals. The computation threads are then assigned over all the subgrids serially through a series of synchronous function calls from the CPU. This provides all GPU threads in each subgrid exactly the same amount of work, which is nonetheless different for different subgrids. The sorting step to divide the work into subgrids is done on the CPU prior to integral evaluation by the GPU.

4d. One Thread ↔ One Primitive Integral Mapping. The "One Thread ↔ One Primitive Integral" (1T1PI) mapping exhibits the finest grained parallelism of all the mappings we consider here. The mapping scheme is shown schematically in Figure 3, where it can be seen that each individual GPU thread calculates only one primitive integral, no matter which contracted integral it contributes to. In Figure 3, two-dimensional blocks of threads are represented by green squares, while the red squares represent contracted integrals. Individual primitive integrals are not displayed. The situation shown in Figure 3 corresponds to the STO-6G basis set (with 1296 primitive integrals in each contracted integral) and a block size of 16 × 16 threads. Since 16 is not a multiple of 36, some blocks like block (4, 1) calculate primitive integrals which belong to different contracted integrals like (11|12) and (11|13). From the computational point of view, this is the fastest version because of ideal load balancing. However, the following sum reduction stage, which converts the calculated primitive integral grid to the final contracted integral grid, is the most expensive part in this model and can decrease the ultimate performance. As depicted in Figure 3, the reduction to contracted integrals will sometimes require summation over values from threads in different blocks, and these therefore may reside on different SMs, necessitating expensive communication.

4e. The $(ss|ss)$ Integral Computation Algorithm. The general formula for primitive $[ss|ss]$ integral evaluation is[12]

$$[s_1s_2|s_3s_4] = \frac{\pi^3}{AB\sqrt{A+B}}K_{12}(\vec{\mathbf{R}}_{\mathbf{12}})K_{34}(\vec{\mathbf{R}}_{\mathbf{34}})F_0\left(\frac{AB}{A+B}[\vec{\mathbf{R}}_{\mathbf{P}} - \vec{\mathbf{R}}_{\mathbf{Q}}]^2\right) \quad (5)$$

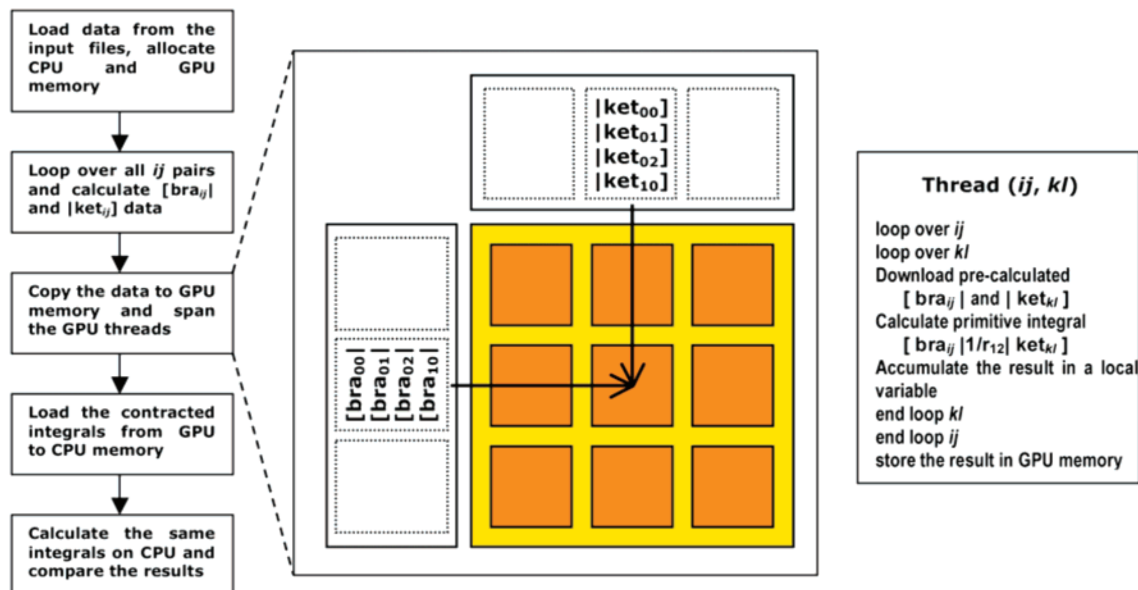

Figure 4. Flowchart for the 1T1CI mapping algorithm. The small orange boxes represent individual threads, each calculating a contracted integral. The whole block of threads (large yellow box) thus calculates a number of contracted integrals (nine in the example shown). The “bra-” and “ket-” rectangles on the top left of the thread block represent the pairwise quantities precalculated on the CPU.

where

$$A = \alpha_1 + \alpha_2; \quad B = \alpha_3 + \alpha_4 \tag{6}$$

$$K_{ij}(\vec{\mathbf{R}}_{ij}) = \exp\left(-\frac{\alpha_i\alpha_j}{\alpha_i + \alpha_j}[\vec{\mathbf{R}}_i - \vec{\mathbf{R}}_j]^2\right) \tag{7}$$

$$\vec{\mathbf{R}}_P = \frac{\alpha_1\vec{\mathbf{R}}_1 + \alpha_2\vec{\mathbf{R}}_2}{\alpha_1 + \alpha_2}; \quad \vec{\mathbf{R}}_Q = \frac{\alpha_3\vec{\mathbf{R}}_3 + \alpha_4\vec{\mathbf{R}}_4}{\alpha_3 + \alpha_4} \tag{8}$$

$$\vec{\mathbf{R}}_{kl} = \vec{\mathbf{R}}_k - \vec{\mathbf{R}}_l \tag{9}$$

$$F_0(t) = \frac{erf(\sqrt{t})}{\sqrt{t}} \tag{10}$$

In eqs 5−10, α_k and $\vec{\mathbf{R}}_k$ denote the exponent and atomic center of the kth primitive basis function in the integral. Instead of having each GPU thread calculate a primitive integral directly through eqs 5−10, we precalculate all pair quantities on the host CPU. Alternatively, this could be done on the GPU, if desired. The following terms are precalculated and stored in the GPU memory: $\alpha_1 + \alpha_2$, $\pi^{3/2} c_1c_2/(\alpha_1 + \alpha_2)\exp(-\alpha_1\alpha_2/(\alpha_1 + \alpha_2)\vec{\mathbf{R}}_{12}^2)$, and $(\alpha_1\vec{\mathbf{R}}_1 + \alpha_2\vec{\mathbf{R}}_2)/(\alpha_1 + \alpha_2)$. Having loaded these pair quantities for both *bra* and *ket* parts, each GPU thread is then able to calculate the required primitive integrals.

4f. Extension to Basis Sets with Higher Angular Momentum Functions. The algorithm presented for $(ss|ss)$ integral calculations is easily generalized to allow for higher angular momentum functions. We discuss some of the relevant considerations here. Consider an example of a system consisting of five uncontracted basis shells: four s-shells and one p-shell. The total number of basis functions is thus seven. First, the shells can be sorted from the lowest angular momentum to the highest angular momentum. In our example, this would lead to $\{1,2,3,4,5\} \leftrightarrow \{s,s,s,s,p\}$. The integral grid can now be generated in exactly the same way as previously discussed. This is shown for the given example in Figure 5, where now each individual square represents an integral batch rather than a single integral. For example, every $(ss|pp)$ batch (pink squares in Figure 5) contains nine integrals. The resulting grid for this example contains 120 unique two-electron integral batches as shown in Figure 5. Different colors represent different types of batches: orange − $(ss|ss)$, blue − $(ss|sp)$, yellow − $(sp|sp)$, pink − $(ss|pp)$, green − $(sp|pp)$, and dark red − $(pp|pp)$. Typical integral evaluation programs have separate routines for each class of integrals represented as different colors in Figure 5. A straightforward method to calculate all the integrals would be the following: a) each of 6 functions spawns a 15 × 15 grid of blocks to cover the whole integral batch grid (Figure 5a); b) depending on its type [$(ss|ss)$, $(ss|sp)$, etc.], each routine (“kernel”) has its own set of rules to extract only those batches (small squares of the same color in Figure 5a) which it is responsible to calculate and schedules out the others; c) the result is then stored in the GPU memory and another function, responsible for another type of batch is called on the same 15 × 15 grid. However, such an approach has one serious drawback—the batches of the same type are scattered over the whole grid (Figure 5a). As a result, the rules each integral evaluation kernel needs to apply to outschedule the unsuitable batches will be rather complicated. In addition, the number of such integral batches rapidly increases with the system size, which will ultimately decrease computational performance.

The integral batch grid shown in Figure 5a was generated by a conventional loop structure as shown in Figure 6a. If one instead adopts a less conventional loop structure as shown in Figure 6b, one obtains the integral batch grid shown

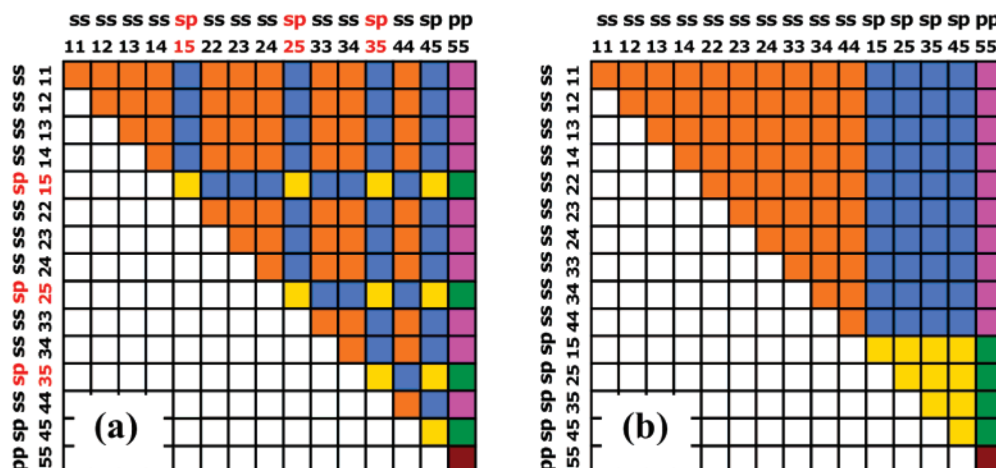

Figure 5. a) The integral grid generated by a conventional loop over shells ordered according to their angular momentum (*s*, then *p*, etc.). b) Rearranged loop sequence that leads to a well-ordered integral grid, suitable for calculations on the GPU. Different colors represent different integral types such as (*ss*|*ss*), (*ss*|*sp*), etc. as discussed in section 4f.

```
Version (a).
loop i over s-shells
        loop j over s-shells
        loop j over p-shells
        loop j over d-shells
loop i over p-shells
        loop j over p-shells
        loop j over d-shells
loop i over d-shells
        loop j over d-shells
```

```
Version (b).
loop i over s-shells
        loop j over s-shells
loop i over s-shells
        loop j over p-shells
loop i over s-shells
        loop j over d-shells
loop i over p-shells
        loop j over p-shells
loop i over p-shells
        loop j over d-shells
loop i over d-shells
        loop j over d-shells
```

Figure 6. Pseudocode for the loop arrangement corresponding to Figure 5.

in Figure 5b. This new integral grid, or *supergrid*, has very well defined *subgrids* of integral batches of the same type. Calculation on the GPU is now straightforward—all the different integral type calculation functions are called on their own subgrids. In this case, the off-diagonal integral types, i.e., (*ss*|*sp*), (*ss*|*pp*), and so on, do not require any block outscheduling, since the corresponding subgrids have rectangular shape. The diagonal integral types still need the very inexpensive "main-diagonal" outscheduling, as discussed previously in the context of the (*ket*|*bra*)=(*bra*|*ket*) symmetry (section 4b). Any of the three different thread-integral mapping schemes discussed above can now be directly applied.

4g. Considerations for Direct Self-Consistent Field Calculations. The next step after generation of the two-electron integrals is the assembly of the Fock operator for direct SCF calculations. In this context, there is no need to generate the contracted integrals explicitly—instead, one can use the primitive integrals directly to generate the desired Coulomb and exchange operators. Thus, the best of the three integral mappings considered above for direct SCF will be the 1T1PI scheme, since it exhibits ideal load balancing and the problematic inter-SM communication requirements are completely alleviated if the contracted integrals are never explicitly formed.

Given the considerations listed above when angular momenta higher than *s* functions are involved, we have extended the 1T1PI scheme such that each individual GPU thread evaluates an entire batch of integrals (the small squares in Figure 5) instead of a single primitive integral (which was the case when only *s* functions were being considered above). Thus, when angular momenta higher than *s* are involved, this scheme might be better denoted as "One Thread ↔ One Batch" (1T1B). Once every integral batch is assigned to a corresponding computation thread, the thread first evaluates the integral Schwartz upper bound.[27] If this upper bound is larger than some predefined threshold (we used the value of 10^{-9} au), the thread calculates all the integrals in the batch and accumulates them in the corresponding elements of the Coulomb matrix.

We have used the Rys-quadrature approach[28] for evaluating integrals involving basis functions of higher angular momenta than *s* functions because it requires little memory for intermediate quantities. This is an important consideration because the amount of memory available to each thread during the computation is limited on the GPU—for optimal performance, one should stay within the register space of each SM as much as possible. On the 8800 GTX, there are 8192 32-bit registers per SM, and this register space must be evenly distributed among all threads executing on the SM. Thus, decreasing memory usage per thread is important to ensure that a large number of threads can execute in parallel on each SM (streaming processors exploit this parallelism to hide latency associated with memory-access). Six different GPU kernels were hand-coded, each capable of handling one of the six unique batch types—(*ss*|*ss*), (*ss*|*sp*), (*ss*|*pp*), (*sp*|*sp*), (*sp*|*pp*), and (*pp*|*pp*). We are developing a program that will generate the source code for basis sets including angular momenta higher than *p* functions.

5. Results and Discussion

First, we have benchmarked these three different mapping schemes on a relatively simple system consisting of 64 hydrogen atoms organized on a 4 × 4 × 4 atom cubic lattice with 0.74 Å nearest-neighbor interatomic distance. The STO-6G and 6-311G basis sets were used, representing highly contracted and relatively uncontracted basis sets, respectively. All GPU computations were performed on one nVidia Geforce 8800 GTX card. For comparison, all reference

Table 1. Timings for the 64 H Atom System Two-Electron Integral Evaluation on the GPU Using the Algorithms (1B1CI, 1T1CI, 1T1PI) Described in the Text[a]

	GPU 1B1CI (s)	GPU 1T1CI (s)	GPU 1T1PI (s)	CPU pre-calculation (s)	GPU-CPU transfer (s)	GAMESS
6-311G	7.086	0.675	0.428	0.009	0.883	170.8
STO-6G	1.608	1.099	2.863	0.012	0.012	90.6

[a] The "CPU precalculation" column lists the amount of time required to generate pair quantities on the CPU, and the "GPU-CPU transfer" column lists the amount of time required to copy the contracted integrals from the GPU to CPU memory. Timings for the same test case using the GAMESS program package on a single Opteron 175 CPU are provided for comparison.

timing data was generated by GAMESS[29,30] running on a single AMD Opteron 175 processor. The GAMESS source code was modified to prevent it from storing the computed integrals on the hard drive, avoiding all I/O and ensuring fair timing comparisons. The results are presented in Table 1. Note that the time required to transfer the integrals from the GPU to CPU memory can be significant, especially for weakly contracted basis sets. In fact, this transfer time can be comparable to the integral evaluation time. The GPU-CPU transfer time is determined by the speed of the host bus, and straightforward calculation from the data in Table 1 (only unique integrals are transferred) gives a speed of 0.7 Gbytes per second, consistent with the expected speeds[31] on the PCI Express x8 bus used (unidirectional peak speed of 2 Gb/s). For the 6-311G basis set, the GPU-CPU transfer time exceeds the integral evaluation itself. Thus, it is clearly desirable to implement a direct SCF approach[11] entirely on the GPU.

As mentioned in section 4, we have chosen the 1T1PI (or 1T1B) mapping for future work in generating a direct SCF algorithm. For a realistic benchmark, we have chosen a 256-atom DNA strand shown in Figure 7. The 3-21G basis set is used, with a total of 1699 basis functions, including both *s* and *p* angular momenta. Although our direct SCF implementation is still under development, we are able to provide significant timing comparisons based on the formation of the Coulomb contribution to the Fock matrix. We compare the GPU time for Coulomb matrix construction to the time GAMESS requires to evaluate all two-electron integrals. For both our GPU implementation and GAMESS, the integral upper bound used for Schwartz inequality prescreening was 10^{-9} Hartree. The GPU implementation does not utilize prescreening based on the density matrix elements, i.e., all two-electron integrals which are not prescreened are calculated and contracted with density matrix elements. The current version of the GPU code for Coulomb matrix construction evaluates $O(N^4/4)$ integrals, while the total number of unique two-electron integrals is $O(N^4/8)$. In other words, each integral is calculated twice. In spite of this fact, the GPU algorithm demonstrates impressive performance—the time required to calculate the Coulomb matrix for the DNA molecule described above (Figure 7) is 19.8 s. Further elimination of the redundant integrals calculated in the GPU algorithm is expected to improve its performance. For comparison, GAMESS requires 1600 s (on an AMD Opteron 175) just to evaluate all the two-electron integrals (which

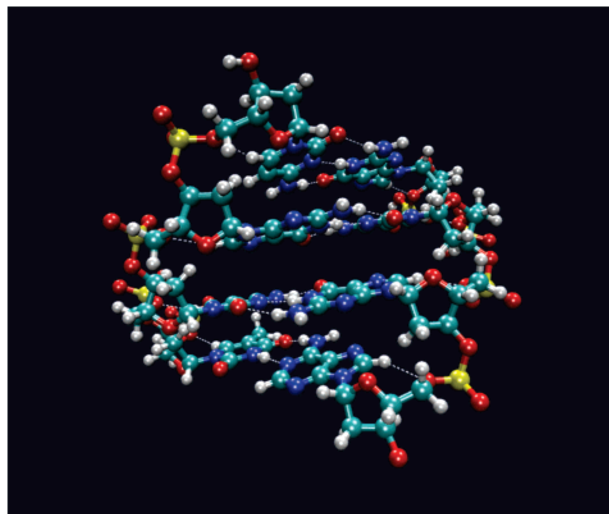

Figure 7. The 256-atom DNA strand used for the Coulomb matrix formation benchmark. The chemical formula of the molecule is $C_{77}P_8N_{31}H_{91}O_{49}$. Our GPU algorithm calculates the Coulomb matrix for this molecule in 19.8 s compared to 1600 s required by GAMESS (on a single AMD Opteron 175 CPU) for evaluation of the two-electron integrals (which need to be further contracted to form the Coulomb matrix).

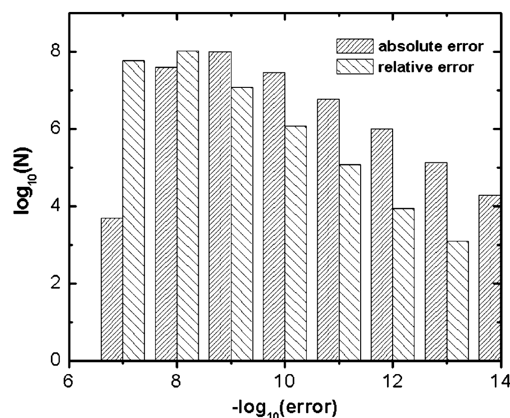

Figure 8. Absolute and relative error distribution of two-electron integrals generated on the GPU for the test system of 64 H atoms on a 4 × 4 × 4 cubic lattice using the 6-311G basis set.

need to be further contracted with the density matrix elements to generate the Coulomb matrix).

An additional issue that merits some discussion is the fact that the 8800 GTX hardware supports only single precision floating point operations. As a result, all the integrals calculated on the GPU have single precision (7-digit) accuracy. Figure 8 presents the number of two-electron integrals calculated for the 64 H atom test system using the 6-311G basis set with given absolute and relative errors as determined by comparison with double precision CPU results. The relative error distribution demonstrates typical behavior for single precision calculations (relative error of $10^{-7}-10^{-8}$). However, electronic structure calculations are often held to an absolute accuracy standard, since it is energy differences that are important. The absolute error distribution has a maximum at $10^{-8}-10^{-10}$ Hartree. In order to save CPU

time, electronic structure codes often neglect two-electron integrals less than 10^{-9} or 10^{-10} Hartree. Thus, the accuracy achieved by the GPU is somewhat worse than what is usually required in quantum chemistry calculations. Yasuda has discussed this in detail and outlined a solution which calculates large integrals (where higher precision is needed) on the CPU and others on the GPU.[9] This is definitely a fruitful approach when confronted with hardware limited to single precision. However, the introduction of double precision support in upcoming GPUs from nVidia and AMD's FireStream processor makes it unclear whether such mixed CPU-GPU schemes will be worth the extra effort in the future. Certainly, it will not be necessary to adopt such a strategy. Only minor revisions of our current single-precision accuracy code will be needed for these 64-bit stream processors, and the relative effectiveness of the algorithms presented here will not be affected at all.

6. Conclusions

We have demonstrated that graphical processors can significantly outpace the usual CPUs in one of the most important quantum chemistry problems bottlenecks—the evaluation of two-electron repulsion integrals. We have achieved speedups of more than 130× for two-electron integral evaluation in comparison to the GAMESS program package running on a single Opteron 175 CPU. One can easily anticipate the possibility of using GPUs in parallel, and hardware support for 4 GPUs per CPU has already been announced by nVidia. Parallelization of electronic structure calculations over GPUs is an obvious next step that is expected in the near future. We have demonstrated the calculation of the Coulomb matrix for a chemically significant test molecule corresponding to a 256-atom DNA strand, showing that the speedups we have obtained are representative of what can be expected on realistic systems.

The integrals generated on the GPU in single precision have a relatively large absolute error of $\approx 10^{-8}$ Hartree. Possible accuracy problems can be addressed by using effective core potentials for medium sized molecules to reduce the dynamic range of the one- and two-electron integrals used to construct the Fock matrix. Alternatively, a hybrid strategy evaluating some of the integrals on the GPU and others on the CPU could be used, as previously demonstrated.[9] However, both nVidia and AMD have already announced GPUs with hardware support for double precision, so this will likely be a moot point within the next few months. Consequently, we are focusing on the development of a complete electronic structure code running almost entirely on the GPU, in anticipation of the coming hardware improvements.

Acknowledgment. This work has been supported by the National Science Foundation (CHE-06-26354 and DMR-03-25939) and the Department of Energy (DEFG02-00ER-15028 and DEFG02-05ER-46260).

References

(1) Kahle, J. A.; Day, M. N.; Hofstee, H. P.; Johns, C. R.; Maeurer, T. R.; Shippy, D. Introduction to the Cell Multiprocessor. *IBM J. Res. Dev.* **2005**, *49*, 589.

(2) Kapasi, U. J.; Rixner, S.; Dally, W. J.; Khailany, B.; Ahn, J. H.; Mattson, P.; Owens, J. D. Programmable Stream Processors. *Computer* **2003**, *36*, 54.

(3) Russel, R. M. The Cray-1 Computer System. *Comm. ACM* **1978**, *21*, 63.

(4) Fatahalian, K.; Sugerman, J.; Hanrahan, P. Understanding the Efficiency of GPU Algorithms for Matrix-Matrix Multiplication. In *Graphics Hardware*; Akenine-Moller, T., McCool, M., Eds.; A. K. Peters: Wellesley, 2004.

(5) Hall, J.; Carr, N.; Hart, J. *Cache and Bandwidth Aware Matrix Multiplication on the GPU*; University of Illinois Computer Science Department Web Site, 2003. http://graphics.cs.uiuc.edu/~jch/papers/UIUCDCS-R-2003-2328.pdf (accessed December 1, 2007).

(6) Bolz, J.; Farmer, I.; Grinspun, E.; Schroder, P. Sparse matrix solvers on the GPU: Conjugate gradients and multigrid. *ACM Trans. Graph.* **2003**, *22*, 917.

(7) Anderson, A. G.; Goddard, W. A., III.; Schroder, P. Quantum Monte Carlo on graphical processing units. *Comput. Phys. Commun.* **2007**, *177*, 265.

(8) Stone, J. E.; Phillips, J. C.; Freddolino, P. L.; Hardy, D. J.; Trabuco, L. G.; Schulten, K. Accelerating molecular modeling applications with graphics processors. *J. Comput. Chem.* **2007**, *28*, 2618.

(9) Yasuda, K. Two-electron integral evaluation on the graphics processor unit. *J. Comput. Chem.* **2007**, *00*, 0000.

(10) Kermes, S.; Olivares-Amaya, R.; Vogt, L.; Shao, Y.; Amador-Bedolla, C.; Aspuru-Guzik, A. Accelerating Resolution of the Identity Second Order Moller-Plesset Calculations with Graphical Processing Units. *J. Phys. Chem. A* **2007**, in press.

(11) Almlof, J.; Faegri, K.; Korsell, K. Principles for a direct SCF approach to LCAO-MO ab initio calculations. *J. Comput. Chem.* **1982**, *3*, 385.

(12) Boys, S. F. Electronic Wave Functions. I. A General Method of Calculation for the Stationary States of Any Molecular System. *Proc. R. Soc. London, Ser. A* **1950**, *200*, 542.

(13) McMurchie, L. E.; Davidson, E. R. One- and two-electron integrals over cartesian gaussian functions *J. Comput. Phys.* **1978**, *26*, 218.

(14) Gill, P. M. W.; Head-Gordon, M.; Pople, J. A. An Efficient Algorithm for the Generation of Two-Electron Repulsion Integrals over Gaussian Basis Functions. *Int. J. Quantum Chem.* **1989**, *23S*, 269.

(15) Dupuis, M.; Marquez, A. The Rys quadrature revisited: A novel formulation for the efficient computation of electron repulsion integrals over Gaussian functions. *J. Chem. Phys.* **2001**, *114*, 2067.

(16) Pople, J. A.; Hehre, W. J. Computation of electron repulsion integrals involving contracted Gaussian basis functions. *J. Comput. Phys.* **1978**, *27*, 161.

(17) Obara, S.; Saika, A. Efficient recursive computation of molecular integrals over Cartesian Gaussian functions. *J. Chem. Phys.* **1986**, *84*, 3963.

(18) http://developer.nvidia.com/cuda (accessed June 1, 2007).

(19) NVIDIA CUDA. *Compute Unified Device Architecture Programming Guide Version 1.0*; nVidia Developer Web Site. 2007. http://developer.download.nvidia.com/compute/cuda/1_0/NVIDIA_CUDA_Programming_Guide_1.0.pdf (accessed December 1, 2007).

(20) In the latest implementation of CUDA, GPU functions are called asynchronously.

(21) Harrison, R. J.; Kendall, R. A. *Theo. Chim. Acta* **1991**, *79*, 337.

(22) Colvin, M. E.; Janssen, C. L.; Whiteside, R. A.; Tong, C. H. *Theo. Chim. Acta* **1993**, *84*, 301.

(23) Brode, S.; Horn, H.; Ehrig, M.; Moldrup, D.; Rice, J. E.; Ahlrichs, R. Parallel Direct SCF and Gradient Program for Workstation Clusters. *J. Comput. Chem.* **1993**, *14*, 1142.

(24) Hehre, W. J.; Stewart, R. F.; Pople, J. A. *J. Chem. Phys.* **1969**, *51*, 2657.

(25) Hehre, W. J.; Ditchfield, R.; Stewart, R. F.; Pople, J. A. *J. Chem. Phys.* **1970**, *52*, 2769.

(26) Krishnan, R.; Binkley, J. S.; Seeger, R.; Pople, J. A. *J. Chem. Phys.* **1980**, *72*, 650.

(27) Whitten, J. L. Coulombic potential energy integrals and approximations. *J. Chem. Phys.* **1973**, *58*, 4496.

(28) Dupuis, M.; Rys, J.; King, H. F. Evaluation of molecular integrals over Gaussian basis functions. *J. Chem. Phys.* **1976**, *65*, 111.

(29) 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. General Atomic and Molecular Electronic Structure System. *J. Comput. Chem.* **1993**, *14*, 1347.

(30) Gordon, M. S.; Schmidt, M. W. Advances in electronic structure theory: GAMESS a decade later. In *Theory and Applications of Computational Chemistry: the first forty years*; Dykstra, C. E., Frenking, G., Kim, K. S., Scuseria, G. E., Eds.; Elsevier: Amsterdam, 2005; p 1167.

(31) The GPU-CPU transfer was not done in page-locked mode for technical reasons related to the version of CUDA we used. Tests using page-locked mode show transfer speeds much closer to the 2Gb/s peak speed.

CT700268Q