JCTC Journal of Chemical Theory and Computation

Quantum Chemistry on Graphical Processing Units. 2. Direct Self-Consistent-Field Implementation

Ivan S. Ufimtsev[†] and Todd J. Martinez*,[†]

Department of Chemistry and The Beckman Institute, University of Illinois, Urbana, Illinois 61801

Received December 2, 2008

Abstract: We demonstrate the use of graphical processing units (GPUs) to carry out complete self-consistent-field calculations for molecules with as many as 453 atoms (2131 basis functions). Speedups ranging from 28× to 650× are achieved as compared to a mature third-party quantum chemistry program (GAMESS) running on a traditional CPU. The computational organization used to construct the Coulomb and exchange operators is discussed. We also present results using three GPUs in parallel, combining coarse and fine-grained parallelism.

Introduction

Graphical processing units (GPUs) and related massively parallel processors are rapidly emerging as a promising architecture for many methods in computational chemistry, including molecular dynamics[1,2] and quantum chemistry.[3−9] Previously, we demonstrated[5,6] that gains in efficiency of up to 130× over conventional CPUs could be achieved for two-electron Coulomb repulsion integral (ERI) evaluation using NVIDIA graphics cards with algorithms well-tuned to the stream processing[10] architecture underlying modern GPUs. Here, we extend our initial implementation to complete formation of the Fock matrix, which forms the dominant computational step in both Hartree−Fock and density functional self-consistent-field calculations. Our implementation as described here is limited to s- and p-type atom-centered Gaussian basis functions, although extension to higher angular momenta is underway. We assess the efficiency of the resulting code by comparison to GAMESS,[11] a mature third-party quantum chemistry package. Because of the observed 2 order-of-magnitude increase in efficiency, we are able to carry out "on-the-fly" ab initio Monte Carlo simulations of water clusters, and the resulting equilibrium structures are in good agreement with previous studies using local optimization. We also carry out direct SCF calculations on a variety of molecules containing up to 453 atoms (2131 basis functions). These calculations show that the GPU can be as much as 650× faster than conventional CPUs, and calculations on molecules with hundreds of atoms can be completed in minutes. Precision remains a critical issue that limits the size of molecules that can be treated accurately. The newest generation of NVIDIA cards includes double-precision (DP) support, and we comment on this briefly.

We begin with a brief overview of the GPU architecture, focusing on the issues that are most relevant to the current work. More detailed descriptions can be found elsewhere.[12] We then describe the *J*- and *K*-matrix formation algorithms, followed by results and discussion. All numerical results were first obtained on a single GeForce 8800GTX card. However, as we were preparing the manuscript, NVIDIA released a new generation of GPUs (G200) that supports DP floating point operations in hardware. Thus, we also present results obtained on one GeForce 280GTX card as well as on a multi-GPU system containing three such cards running in parallel. The hardware overview and performance sections refer to the G80 GPU hardware. Some parts of our code invoke DP calculations on the GPU, and we comment on this briefly. Everywhere in this Article we use NVIDIA's CUDA[12] (Compute Unified Device Architecture) programming interface that allows control of the GPU via an extension of the standard C language.

Overview of Graphics Hardware

GPUs are an example of a stream processing architecture,[10] emphasizing fine-grained parallelism and efficient utilization

[†] Present address: Department of Chemistry, Stanford University, Stanford, CA 94305.

* Corresponding author phone: (650) 736-8860; e-mail: Todd.Martinez@stanford.edu.

of the memory hierarchy. Typical GPUs contain many arithmetic units, which are arranged in groups that share fast access memory and an instruction unit. For example, the GeForce 8800GTX graphics card, which was released in late 2006, has 128 arithmetic units (also referred to as streaming processors, or SPs) organized into 16 streaming multiprocessors (SMs). Each SM (consisting of 8 SPs in the 8800GTX) has its own instruction unit and thus operates independently of the other SMs. The SPs comprising an SM operate together in single instruction multiple data (SIMD) fashion. In other words, at any moment, all 8 SPs on an SM are executing exactly the same instruction on different data streams.

Because the computational units of the GPU operate in parallel, problems should be split into a set of smaller tasks (threads) that are distributed among the different processors. In NVIDIA's CUDA paradigm, all threads are organized into a 1D- or 2D-grid of 1-, 2-, or 3D blocks with up to 512 threads in each block. The choice of grid topology is dictated by the particular problem under consideration; in our case, we use a 2D grid of 2D blocks. Threads in the same block are always executed on the same multiprocessor, providing them access to the SM's shared memory for fast data exchange. In contrast, distinct blocks are not guaranteed to be executed on the same SM. This means that distinct blocks must communicate through the main GPU memory, which is much slower (by 2 orders of magnitude) than the shared memory. In addition, block execution cannot be efficiently synchronized. Therefore, any efficient GPU algorithm should avoid expensive interblock data transfer, having all interthread communication, if any, occur within blocks. The hardware splits the thread blocks into warps with 32 threads in a warp, which are processed by SMs in SIMD fashion with all 32 threads executed by 8 streaming processors in 4 clock cycles. This SIMD execution model requires additional effort when developing GPU-based algorithms, because all threads in the same warp must execute exactly the same set of instructions. A typical case demonstrating violation of this requirement and subsequent deleterious consequences is a simple loop where the number of iterations is determined at runtime. If there are threads in a warp that complete their loop as the others continue iterating, the idle threads will still execute the unnecessary loop instructions until the last thread in the warp is done. Such idle threads, thereby, occupy the hardware resources, which otherwise would be used for useful work, and degrade the algorithm performance.

Several basic strategies for efficient execution on the GPU architecture can be delineated.

(1) The first is data localization. Contiguous threads should access contiguous DRAM addresses to coalesce multithread memory instructions into one memory load/store operation. Otherwise, the overall memory bandwidth can drop by a factor of 10 or even more, completely obviating the computational advantage of the GPU.

(2) The second is little interthread communication. Most streaming-type architectures (including NVIDIAs) do not provide efficient tools for fast interthread data exchange, except when threads belong to the same block. All communication must be restricted to threads within the same block, and ideally there is no communication between threads.

(3) The third is elimination of memory access collisions. Neither CUDA nor the NVIDIA hardware support floating-point atomic memory operations. In other words, if two or more threads write to the same memory address at the same instant of time, the result is unpredictable. It is therefore critical to ensure that potential memory access collisions are avoided, which requires careful attention to the relationship between a thread and the memory it will write to.

(4) The fourth is dense computing. Because thread warps are processed in SIMD fashion, it is imperative to keep the whole warp busy. If some threads in a warp are not carrying out useful work, performance degradation will occur. Because of the efficient warp scheduler, there is little penalty if all threads in a warp are idle; in this case, the entire warp will be removed from execution scheduling.

Self-Consistent-Field Calculations

The evaluation of a large number of two-electron repulsion integrals (ERIs) over N atom-centered basis functions is one of the most computationally intensive tasks in any ab initio Molecular Orbital (MO) or Density Functional Theory (DFT) approach. These integrals are given by:

$$(\phi_\mu\phi_\nu|\phi_\lambda\phi_\sigma) = \int\int \phi_\mu(\vec{r}_1)\phi_\nu(\vec{r}_1)\frac{1}{|\vec{r}_1 - \vec{r}_2|}\phi_\lambda(\vec{r}_2)\phi_\sigma(\vec{r}_2)\, d\vec{r}_1\, d\vec{r}_2 \quad (1)$$

where the ϕ are linear combinations (contractions) of atom-centered Gaussian basis function primitives. In our previous work,[6] we explored three different algorithms for ERI evaluation on GPU. The algorithms possess different granularity of the problem decomposition and cover a wide range of possible thread-integral mapping strategies. It was found that the so-called "1 Thread − 1 Contracted Integral" (1T1CI) algorithm was the most efficient at generating the set of all contracted integrals. Another approach, "1 Thread − 1 Primitive Integral" (1T1PI), was the fastest in primitive integral evaluation. However, the subsequent integral transformation step, leading to the final set of contracted integrals, was not very efficient and sometimes required so much labor as to mitigate the advantage of the 1T1PI mapping strategy. We also observed that the time required to transfer calculated ERIs from GPU to CPU memory can be comparable to the ERI calculation time due to limited CPU−GPU bus bandwidth.[6]

This removes the need for the problematic primitive→ contracted integral transformation and limits the GPU−CPU communication to matrices of size $O(N^2)$, where N is the number of primitive basis functions. In this case, GPU−CPU communication requires a negligible amount of the total time. We also use an incremental Fock matrix formation procedure ("Fock matrix updating"),[13] where the GPU computes only the difference between Fock operators in successive iterations. This has the usual advantage of minimizing the number of Fock matrix elements that need to be computed, especially as convergence is reached in later iterations. However, an

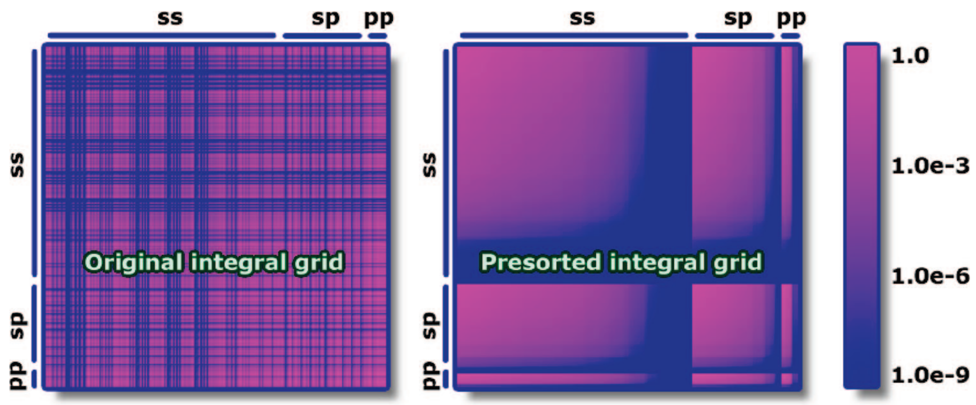


Figure 1. Illustration of ERI organization for *J*- and *K*-matrix construction, using data obtained from a representative calculation on ethane. The row and column indices correspond to *bra* and *ket* pairs of a given primitive integral, and the coloration indicates the magnitude of the Schwartz upper bound for a given integral. After the *ket* and *bra* pairs are ordered by angular momentum, the grid on the left is obtained. The sparse structure of the resulting grid hampers effective parallelization because of load imbalance. Further sorting of the *ket* and *bra* pairs within each angular momentum class according to the magnitude of each pair's contribution to the Schwartz upper bound leads to the grid on the right, which is well-suited for computations on the GPU.

additional advantage in the present context is that the incremental Fock matrix procedure is numerically more accurate when integrals are generated with limited, that is, 32-bit, precision.

In Hartree−Fock (HF) calculations for molecules with up to a few thousand basis functions, most of the computational time is spent on formation of the Fock matrix, comprised of one-electron, Coulomb, and exchange contributions:

$$\mathbf{F}(\mathbf{C}) = \mathbf{H}_{\mathbf{core}} + \mathbf{J}(\mathbf{C}) - \frac{1}{2}\mathbf{K}(\mathbf{C}) \qquad (2)$$

where **C** is the matrix of the molecular orbital (MO) coefficients in the atomic orbital (AO) basis, and $\mathbf{H}_{\mathbf{core}}$ includes all one-electron terms. We construct the *J*- and *K*-matrices entirely on the GPU and transfer the resulting matrices to the CPU, where they are augmented with the one-electron term to yield the final Fock matrix. Subsequent SCF operations, such as the transformation of **F** to an orthogonal basis set and its subsequent diagonalization, are performed either on the GPU (dgemm calls to the NVIDIA CUBLAS library) or on the CPU (dsyevd calls to the Intel MKL 10.0.3.020 library). All operations performed on the CPU are carried out in full 64-bit arithmetic, while the operations on the GPU may be carried out in 32-bit, 64-bit, or hybrid of 32-bit and 64-bit arithmetic as discussed below.

Organization and Prescreening of Integrals

Efficient algorithms on the GPU require careful attention to the relationship between data storage and computation. We view the fundamental data structure in the direct SCF problem as an $N^2 \times N^2$ matrix of integrals, where the row and column indices correspond to $[\mu\nu|$ (*bra*) and $|\lambda\sigma]$ (*ket*) pairs; two such pairs, when combined together, index a primitive $[\mu\nu|\lambda\sigma]$ integral. For each *bra* and *ket* pair, one can define a Schwartz upper bound[14] such that the product of the bounds for *bra* and *ket* provides an upper bound to the corresponding [*bra*|*ket*] integral. The left panel of Figure 1 depicts this structure graphically for the specific case of the ethane molecule, using blue−pink coloration to denote the magnitude of the corresponding bound. The *bra* and *ket* pairs are sorted according to $\mu\nu$ and $\lambda\sigma$ basis function types (our code currently supports s- and p-type basis functions, although implementation of higher orbital momentum functions is underway; and there are three possible combinations if s- and p-functions are involved, *ss*, *sp*, and *pp*), resulting in an ERI grid that is well ordered with respect to $[\mu\nu|\lambda\sigma]$ integral types. As we have discussed previously,[6] this grouping makes it possible to efficiently calculate the different angular momentum classes of integrals. However, incorporating prescreening into the algorithm (ignoring integrals that are known by the Schwartz bound to be below a given numerical threshold) leads to load balancing problems because the integrals that still need to be computed are scattered in irregular fashion throughout the integral matrix. Thus, we further presort the *bra* and *ket* pairs (within each angular momentum class) according to the pair's contribution to the Schwartz bound. This leads to a well-ordered integral grid as shown in the right panel of Figure 1. This arrangement of the integral matrix leads to efficient algorithms on the GPU, as discussed below.

J-Matrix Algorithm

The *J*-matrix is formed directly from primitive ERIs using the 1T1PI algorithm, in which each GPU thread evaluates one primitive ERI (or, as we have discussed previously,[6] a batch of ERIs when higher than *s* angular momentum basis functions are involved). The elements of the *J*-matrix are given as:

$$J_{(\mu\nu)} = \sum_{(\lambda\sigma)} P_{(\lambda\sigma)}(\mu\nu|\lambda\sigma) \qquad (3)$$

Following conventional usage,[15−17] we use parentheses or brackets throughout to indicate that the Greek indices refer to contracted or primitive basis functions, respectively. We build the *J*-matrix directly from primitive integrals as:

$$J_{[\mu\nu]} = \sum_{[\lambda\sigma]} P_{[\lambda\sigma]}[\mu\nu|\lambda\sigma] \tag{4}$$

which is then contracted into the final *J*-matrix:

$$J_{(\mu\nu)} = \sum_{i,j} J_{[\mu_i\nu_j]} \tag{5}$$

where μ_i denotes the *i*th primitive function contributing to the μth contracted function. Following Ahmadi and Almlof,[18] we first expand the Cartesian Gaussian primitive pair products in terms of a Hermite Gaussian basis set and preprocess the corresponding density matrix elements $P_{[\lambda\sigma]}$ accordingly. For example, each *ss* pair can be expressed with a single (Λ_0) Hermite basis function, while the three products in an *sp* pair (sp_x, sp_y, and sp_z) are expanded over four Hermite basis functions (Λ_0, Λ_x, Λ_y, and Λ_z). Denoting *bra* and *ket* primitive Hermite products as [*p*| and |*q*], eq 4 can be written as:

$$\begin{aligned} J_{[\mu\nu]} &= \sum_{[pq\lambda\sigma]} E_{[p]}^{[\mu\nu]} E_{[q]}^{[\lambda\sigma]} P_{[\lambda\sigma]}[p|q] \\ &= \sum_{[pq]} E_{[p]}^{[\mu\nu]} P_{[q]}[p|q] \\ &= \sum_{[p]} E_{[p]}^{[\mu\nu]} J_{[p]}^{\text{Hermite}} \end{aligned} \tag{6}$$

where $E_{[p]}^{[\mu\nu]}$ are the expansion coefficients of the Hermite pair primitive functions in terms of the primitive Cartesian Gaussian basis functions, and the second line defines the preprocessed density matrix in terms of Hermite primitive pair products, $P_{[q]}$. Generation of all required information about the Hermite primitive pair products and preprocessing of the density matrix is carried out on the CPU. The [*p*|*q*] integrals are calculated using the McMurchie−Davidson algorithm[19] on the GPU and contracted with the preprocessed density matrix elements to generate $J_{[p]}^{\text{Hermite}}$. Although the integrals are calculated with 32-bit arithmetic, their accumulation is done with double precision accuracy to minimize numerical error accumulation during the summation.[20] The $J_{[p]}^{\text{Hermite}} \rightarrow J_{[\mu\nu]}$ postprocessing (transformation from Hermite to Cartesian) and subsequent $J_{[\mu\nu]} \rightarrow J_{(\mu\nu)}$ contraction are done on the CPU, leading to the final *J*-matrix.

It is well-known that efficient evaluation of two-electron integrals requires different subroutines for differing angular momentum classes, for example, (*ss*|*ss*) or (*ss*|*sp*). Because efficient execution on the GPU requires maximizing the amount of work for a given kernel (which can loosely be regarded as a subroutine), we have previously described[6] a supergrid representation of the two-electron integrals that allows one to easily treat integrals of the same type together (see Figure 1). For purposes of computational organization, we imagine the primitive integrals [*p*|*q*] arranged in a square grid, shown in Figure 2. The $\mu\nu \leftrightarrow \nu\mu$ and $\lambda\sigma \leftrightarrow \sigma\lambda$ index permutation symmetries for the $[\mu\nu|\lambda\sigma]$ integrals are easily incorporated by pruning the list of [*p*| and |*q*] pair products, which comprise the rows and columns and the usual postprocessing (e.g., doubling off-diagonal contributions). Thus, the final two-dimensional grid of primitive integrals is of dimension $N(N + 1)/2 \times N(N + 1)/2$. The index-symmetry pruned list of [*p*| and |*q*] pair products are sorted according to their Schwartz upper bounds:

$$[p|_{\text{Schwartz}} = \sqrt{[p|p]} \tag{7a}$$

$$|q]_{\text{Schwartz}} = |P_{\lambda\sigma}|^{\max}\sqrt{[q|q]} \tag{7b}$$

where we note that the order of the *bra* and *ket* arrays may be different because we include the density matrix element contribution in the *ket* bound. When the angular momentum of one or both of the basis functions involved in a pair product is greater than *s*, we follow the usual practice of using a single [*p*|*p*] in eq 7 corresponding to the pair product with $l = 0$. We also use the maximum of the density matrix elements over all angular momentum components (in the Cartesian representation) in the second line of eq 7. If one considers a basis set containing at most $l = 1$ angular momentum basis functions, the procedure thus boils down to first grouping the [*p*| and |*q*] lists by the angular momentum of the Cartesian Gaussians ($\mu\nu$ and $\lambda\sigma$) leading to *ss*, *sp*, and *pp* groups for each of [*p*| and |*q*]. Each of the resulting six lists is then sorted according to the Schwartz bounds given in eq 7. Nine GPU kernels are then called, one for [*ss*|*ss*], [*ss*|*sp*], [*ss*|*pp*], [*sp*|*ss*], and so on. All of the GPU kernels were hand-coded; however, an automatic GPU code generation tool for basis sets with higher angular momentum is currently under development in our group.

At this point, we simplify the description of our implementation by considering only one of the integral classes, for example, [*ss*|*ss*]. All other integral classes are treated similarly. Figure 2 depicts the primitive integral grid for a representative integral class, with the pink/blue color scheme denoting the magnitude of the primitive integral Schwartz upper bounds as in Figure 1. The triangles above and to the left of the grid represent the Schwartz bounds for *ket* and *bra*, respectively. Each small cyan-bordered square in Figure 2 represents a single [*p*|*q*] primitive integral. It is easily seen that each row of this integral matrix contributes to a single matrix element of J^{Hermite}. As mentioned above, each primitive integral will be computed by a single GPU thread; that is, each small square will translate to a GPU thread. The organization into thread blocks is indicated by the orange squares; that is, each orange square translates to a thread block. We show a 2 × 2 arrangement of threads into thread blocks for illustration in Figure 2, but the actual implementation uses an 8 × 8 arrangement, which we found to be optimal. To utilize memory effectively, the integral matrix is processed by columns, assigning thread blocks to each column. Each thread in each thread block calculates the appropriate primitive integral multiplied by the appropriate preprocessed density matrix element. When all threads in these thread blocks are complete, the next group of columns in the integral matrix is processed. The same thread blocks will treat the integrals in a given row, which ensures that they already have the partially formed matrix elements of J^{Hermite} in shared memory. Once all columns have been processed, reduction along the threads in each row of the

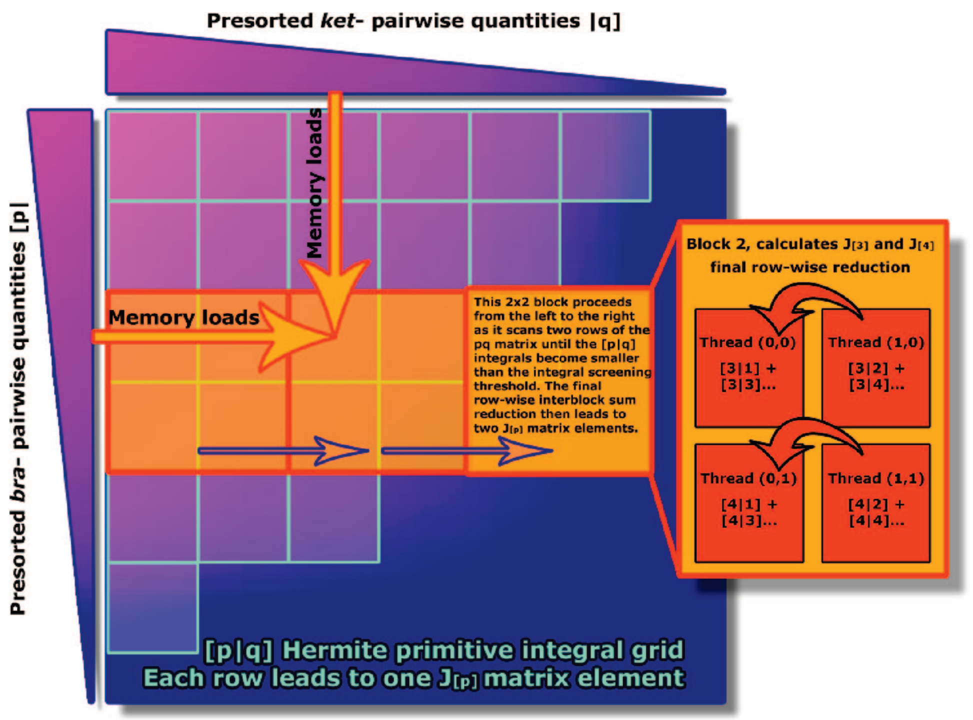

Presorted *ket*- pairwise quantities |q]
Presorted *bra*- pairwise quantities [p|
Memory loads
Memory loads
This 2x2 block proceeds from the left to the right as it scans two rows of the pq matrix until the [p|q] integrals become smaller than the integral screening threshold. The final row-wise interblock sum reduction then leads to two $J_{[p]}$ matrix elements.
Block 2, calculates $J_{[3]}$ and $J_{[4]}$ final row-wise reduction
Thread (0,0) [3|1] + [3|3]...
Thread (1,0) [3|2] + [3|4]...
Thread (0,1) [4|1] + [4|3]...
Thread (1,1) [4|2] + [4|4]...
[p|q] Hermite primitive integral grid
Each row leads to one $J_{[p]}$ matrix element

Figure 2. $J_{[p]}$-Matrix formation algorithm for a given angular momentum block, e.g., [*ss*|*ss*]. Cyan-bordered squares depict primitive integrals that need to be evaluated and further contracted into the matrix elements. The *bra*- and *ket*-pair quantities (left and upper triangles) are presorted and lead to a well-organized grid of integrals, whose upper bound is represented by the blue−pink (with pink being largest) color scheme. The orange-bordered squares show those integrals that are evaluated by one block as it scans the integral grid from the left to the right and accumulates the results. Those integrals with Schwartz bound lying below the prescreening threshold (lower-right part of the grid) are ignored. Yellow arrows illustrate the GPU memory load operations.

thread block leads to the final matrix element of J^{Hermite}. These elements are sent to the CPU for postprocessing.

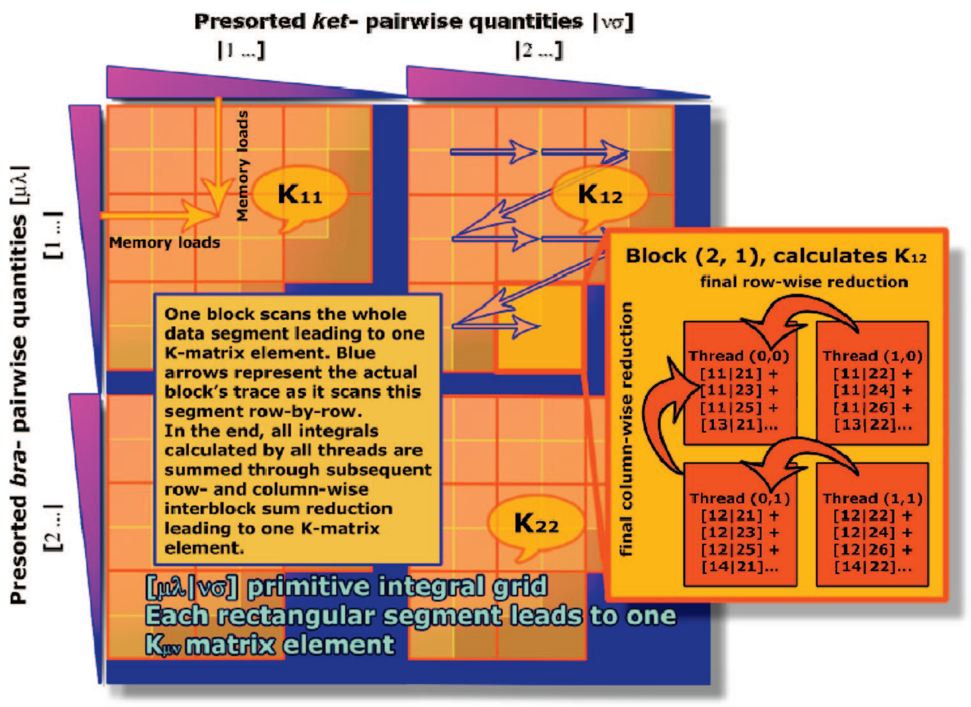
This approach evaluates approximately twice as many integrals as minimally required, because the same integrals are present in distinct rows due to the [*p*|*q*]↔[*q*|*p*] index permutation symmetry that we do not account for. Our choice was dictated by the interblock communication and memory access collision requirements discussed above, and our implementation completely satisfies both of these requirements. Additionally, this row-wise model is perfectly suitable for multi-GPU parallelization because distinct rows can be sent to different GPUs. Expensive internode communication is avoided because each GPU has all of the data needed to complete its share of the work (these data consist of the pair quantities corresponding to the [*p*| and |*q*] lists discussed above).

K-Matrix Algorithm

The computation of the *K*-matrix has very different memory access requirements as compared to the *J*-matrix. Additionally, it is not possible to easily split the work using the pair product representation as outlined for the *J*-matrix above. Therefore, we make no attempt to use the integrals calculated during *J*-matrix construction in the computation of the *K*-matrix. Instead, the *K*-matrix calculation is completely independent of the *J*-matrix calculation. As will be seen below, the data organization is also completely different, and therefore the pair quantities are generated and sorted completely separately. The basic organizational strategy that we use eliminates both interblock communication and memory access collisions by having just one block of GPU threads calculate one $K_{\mu\nu}$ matrix element.

Unlike the *J*-matrix algorithm, the $\mu\lambda\leftrightarrow\lambda\mu$ and $\nu\sigma\leftrightarrow\sigma\nu$ symmetries cannot be exploited because, for example, [*bra*|$\nu\sigma$] and [*bra*|$\sigma\nu$] integrals contribute to distinct *K*-matrix elements and thus reside in different GPU blocks. Exploiting this symmetry would necessitate expensive interblock communication. All [*bra*| and |*ket*] pair-quantities are thus grouped into four rather than three segments (assuming that only s- and p-type basis functions are employed), *ss*, *sp*, *ps*, and *pp*, producing 16 distinct ERI subgrids instead of the nine distinct subgrids depicted in Figure 1. Exploiting the [*bra*|*ket*]↔[*ket*|*bra*] symmetry reduces this to 10 subgrids, because only the upper triangle needs to be considered.

Figure 3 provides more details on the *K*-matrix algorithm using one of the 10 angular momentum subgrids as an example. For each of the angular momentum ERI subgrids, we further group the basis function pairs by the first primitive index. The pair quantities are then sorted by the Schwartz upper bound within each of these classes (angular momentum class and first primitive index), where we do not include the density matrix estimation in the bound computed for either [*bra*| or |*ket*]. Thus, we have a series of [μ...|ν...] blocks (with size $N \times N$) of the integral matrix, each of which contributes to one $K_{\mu\nu}$ matrix element. One GPU thread block is tasked with the evaluation of each $K_{\mu\nu}$ matrix element and winds


Presorted *ket*- pairwise quantities |νσ]
|1 ...]
|2 ...]
Presorted *bra*- pairwise quantities [μλ|
[1 ...|
[2 ...|
Memory loads
Memory loads
K11
K12
K22
Block (2, 1), calculates K12
final row-wise reduction
final column-wise reduction
Thread (0,0) [11|21] + [11|23] + [11|25] + [13|21]...
Thread (1,0) [11|22] + [11|24] + [11|26] + [13|22]...
Thread (0,1) [12|21] + [12|23] + [12|25] + [14|21]...
Thread (1,1) [12|22] + [12|24] + [12|26] + [14|22]...

One block scans the whole data segment leading to one K-matrix element. Blue arrows represent the actual block's trace as it scans this segment row-by-row. In the end, all integrals calculated by all threads are summed through subsequent row- and column-wise interblock sum reduction leading to one K-matrix element.

[μλ|νσ] primitive integral grid Each rectangular segment leads to one $K_{\mu\nu}$ matrix element

Figure 3. *K*-Matrix formation algorithm. Similar to the *J*-matrix algorithm, but in this case, each GPU thread block scans its own square segment of the integral grid rather than several rows of the whole grid. Each square segment ([1...|1...], [1...|2...], or [2...|2...]) leads to one *K*-matrix element.

its way through the corresponding [μ...|ν...] block as shown for K_{12} in Figure 3. As for the *J*-matrix above, we depict a 2×2 configuration of threads in each thread block, but we use an 8×8 configuration in the code. Even though the Schwartz bounds for neither [*bra*| nor |*ket*] are weighted with the density matrix elements $P_{\lambda\sigma}$, we do utilize the *P*-matrix sparsity. As a thread block scans its way through the [μ...|ν...] block, we monitor the product of the Schwartz upper bound and the density matrix element $P_{\lambda\sigma}$. Once this product falls below the standard ERI threshold (10^{-11} au) multiplied by a "guard" parameter (chosen to be 10^{-5}) for all threads in a warp, we abort the scan of the row and the warp proceeds to the next one. We verified that this procedure does not affect the final SCF energies, while it leads to considerable computational savings. In addition, the integrals are accumulated with double precision accuracy, as described above for Coulomb matrix formation. The McMurchie–Davidson algorithm was used to calculate the ERIs with pre- and postprocessing steps performed inside the GPU kernel. In our *K*-matrix implementation, $O(N^4/2)$ integrals are evaluated. Together with $O(N^4/4)$ ERIs calculated to form the *J*-matrix, the number of integrals calculated at each Fock matrix formation step is $O(3N^4/4)$, that is, 6 times more than the number of unique ERIs. Despite this apparent computational inefficiency, the GPU is still able to achieve significant performance gains as compared to traditional CPU implementations. In addition, the *K*-matrix algorithm possesses the same level of parallelism as does the *J*-matrix algorithm, making the whole Fock matrix formation step suitable for implementation on multi-GPU platforms.

Multi-GPU Parallelization

We have also investigated the possibility of using multiple GPUs in parallel. Our parallel *J*-matrix algorithm maps different rows of the [*p*|*q*] Hermite integral matrix to different GPUs cyclically. Each GPU thus computes its own subset of the $J_{[p]}^{\mathrm{Hermite}}$ matrix that is then copied to the host. In the parallel *K*-matrix algorithm, different GPUs calculate different rows of the *K*-matrix, which are also mapped to the devices cyclically. Tests performed on a 3-GPU (GTX280) system, vide infra, show reasonable speedups of 2.0–2.8× over a single GPU board for both *J*- and *K*-matrix algorithms. We have not made any attempt at dynamic load balancing in our current implementation, and this is expected to improve the scalability even more, bringing it closer to the ideal 3× speedup.

Table 1. Specifications of Several GPU Kernels Invoked during *J*-Matrix Formation on GeForce 8800GTX[a]

kernel	FLOPs	MOPs	registers per thread	active threads per SM	performance (GFLOPS)	bandwidth (Gbytes/s)
ssss	30	12	20	384	88 (175)	131
sssp	55	15	24	320	70 (174)	71
sspp	84	21	24	320	69 (227)	64
pppp	387	21	56	128	97 (198)	20

[a] Four out of nine kernels are presented. The performance values in parentheses correspond to the test code where some memory load operations were removed as described in the text.

Results and Discussion

A detailed analysis of the performance obtained during *J*-matrix formation for a representative test case is presented in Table 1. Statistics are shown for four of the two-electron integral kernels, which are grouped by the angular momentum of the basis functions involved. The number of floating-point and memory operations (FLOPs and MOPs, respectively) for each kernel were determined by hand-counting instructions in the intermediate assembly level code generated by the compiler. All floating-point instructions (including division and exponentials) were counted as a single FLOP with the exception of fused multiply add (MAD), which was counted as two FLOPs. For memory operations, we have counted 32-, 64-, and 128-bit memory loads/stores as 1, 2, and 4 MOPs, respectively. Because there is limited local memory (registers) on each SM, the amount of local memory required by each thread executing a kernel impacts the number of threads that can execute simultaneously on a single SM. Thus, we show the number of registers required for each thread by the different kernels in Table 1. As this number increases, the number of threads that can be active on a single SM decreases. This is generally expected to reduce the ability to hide memory access latency effectively.

The GPU hides on-board memory access latency (typically constituting hundreds of clock cycles) by time-sliced execution of a large number of parallel threads (up to 768 parallel threads can run on each SM, and one GeForce 8800GTX card has 16 SMs). Once a thread issues a memory access instruction, the warp scheduler switches to other warps, thus keeping the whole device busy. If the number of active warps is not large enough, the computation can stall because all active warps will eventually be waiting for data to return from memory accesses. In fact, the number of active warps is solely determined by the amount of resources (registers, shared memory, etc.) the GPU kernels consume, and it is highly desirable to develop "light" kernels to ensure that many GPU threads can execute in parallel. We present both the number of registers allocated per thread and the consequent number of active threads running on one SM in Table 1. Obviously, the more registers a kernel consumes the less threads are active (i.e., run in parallel), and memory latency hiding becomes less efficient. This adversely affects the sustained bandwidth[21] (last column in Table 1).

A single GeForce 8800GTX card can provide up to 350 GFLOP/s (GFLOPS) computational performance. How much of this power can be harnessed in quantum chemistry calculations? The results from our current implementation are shown in the "performance" column in Table 1. The kernels sustain close to 30% of the theoretical peak performance. Although we consider this level of performance to be quite reasonable for a complex task like Fock matrix formation, one might wonder how to further improve. To determine the extent to which memory bandwidth is controlling the observed performance, we removed the fundamental integral interpolation table and density matrix element loads, replacing the corresponding variables by arbitrary constants. The resulting performance is represented in parentheses in the "performance" column of Table 1. Even though this code transformation does not change the number of FLOPs executed by each kernel, the sustained performance increases by a factor of 2 or even more. This demonstrates that our current implementation is limited by memory bandwidth. Presumably, some part of the memory latency effect is generated by the fundamental integral interpolation table lookup routines that access the device memory in random fashion, making it difficult for the hardware to coalesce such random loads into one memory operation. We are currently developing a scheme that better meets the GPU memory access requirements.

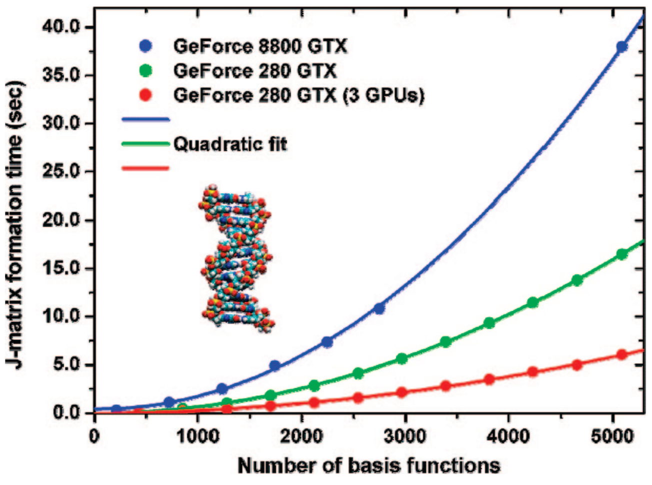


Figure 4. Timings for *J*-matrix construction on a GPU for a series of DNA duplexes using the 3-21G basis set. The solid line is a quadratic fit, showing that our GPU algorithm exhibits ideal quadratic scaling with as few as 500 basis functions. The performance of the algorithm on a multi-GPU system is also shown, indicating its efficient parallel performance.

A series of *J*-matrix formation benchmarks has been performed on duplex-strands of DNA, containing an increasing number of base pairs using the 3-21G basis set. The *J*-matrix formation timings are presented in Figure 4 as a function of the total number of basis functions. Significant acceleration is achieved, allowing the calculation of electronic structure for molecules with up to a thousand basis functions in a matter of seconds. Moreover, the timings are accurately fitted by a quadratic polynomial showing that our algorithm exhibits perfect quadratic scaling already for a few hundred basis functions. This is because of the efficient treatment of prescreening arising from the sorting of the [*bra*| and |*ket*] lists described above. The timings reported in Figure 4 include CPU-side work, for example, sorting and calculation of pair quantities, as well as GPU−CPU data exchange. One can see from Figure 4 that the latest GeForce 280GTX card is significantly faster than the GeForce 8800GTX and furthermore that our parallel implementation achieves a 2.8× speedup using three 280GTX cards.

As pointed out earlier, the GPU architecture is very different from that of CPU, and one might therefore expect that programs optimized for CPU architectures will not fully benefit from the computational power provided by modern streaming processors. Recently, Yasuda[7] reported GPU acceleration of *J*-matrix formation in a development version of the Gaussian electronic structure code. Comparison with the present results allows one to get a sense of the performance gains that can be achieved by careful choice of "stream-friendly" algorithms, as opposed to retrofitting existing CPU algorithms. Thus, we obtained timings on the same GPU hardware used by Yasuda (8800GTX) for *J*-matrix formation for taxol and valinomycin using the 3-21G and 6-31G basis sets. The comparison with timings from our code (including pre- and postprocessing steps performed on the CPU and CPU−GPU data transport overhead) is shown in Table 2. Yasuda's implementation uses

Table 2. *J*-Matrix Formation Time (in seconds) for Taxol and Valinomycin Molecules Using 3-21G and 6-31G Basis Sets[a]

	J-matrix formation time (s)			
	Gaussian on GPU[7]		present work	
molecule	3-21G	6-31G	3-21G	6-31G
taxol	16.8	31.9	0.60	1.65
valinomycin	23.8	57.4	1.05	3.05

[a] Previously reported GPU-accelerated timings based on modifying an existing code (Gaussian) are compared to timings from our code, which has been completely redesigned for stream processors. This comparison emphasizes the gains that can be achieved by redesigning an entire SCF code for stream processing, as opposed to porting an existing CPU algorithm. All timings were obtained on one GeForce 8800GTX card.

multipole methods on the CPU for all far-field contributions (not included in the timings in Table 2), while we evaluate all two-electron integrals explicitly. Thus, this comparison is strongly tilted in favor of Yasuda's implementation. Nevertheless, we find that our code is between 20 and 30 times more efficient than the previous implementation. Of course, neither of these implementations is fully optimized, but the results suggest that there is compelling reason to rethink electronic structure algorithms in the context of streaming processor architectures.

To further demonstrate the effectiveness of quantum chemistry on the GPU, we performed ab initio Monte Carlo (AIMC) simulation of two molecular systems on one GeForce 8800GTX card using the 3-21G basis set: (1) a cluster of 10 rigid water molecules, and (2) the neutral photoactive yellow protein (PYP) chromophore[22] solvated by 45 rigid water molecules. In both cases, the temperature was set at 300 K. Two AIMC runs were performed for the water cluster, and one AIMC run for the solvated PYP chromophore with fixed geometry. Following Wales and Hodges,[23] all MC iterations were grouped into two sets: displacements and rotations, each containing 30 successive MC steps. In the former set, all water molecules were randomly displaced along the three axes, while in the latter set, each water molecule was randomly rotated about the oxygen atom. The displacement and rotation parameters were independently adjusted during the simulations to provide an acceptance ratio of 30−40% in both sets. All water molecules were displaced or rotated in each MC iteration. The initial positions of the water molecules in all three runs were generated randomly; that is, we did not use any pre-equilibration techniques like simulated annealing. The left panel in Figure 5 shows two of the $(H_2O)_{10}$ configurations found after the MC procedure. When further optimized to the nearest energy local minimum, both of these form well-known structures that have been previously characterized.[24] The right panel of Figure 5 shows a snapshot from the solvated PYP chromophore simulation. On average, one MC step required 0.5 and 14 s for the water cluster and solvated chromophore simulations, respectively. Out of these 14 s for the solvated chromophore, a noticeable part of the time was spent on the one-electron integral evaluation (3.4 s) and Fock matrix diagonalization (2.4 s) steps that are currently performed on the host CPU. For bigger systems, the impact of linear algebra (matrix multiplication and diagonalization) will grow in significance because of its cubic scaling as compared to the quadratic scaling of Fock matrix formation. A similar problem emerged as linear scaling electronic structure methods[25] were developed, and we are currently investigating the use of alternatives to diagonalization such as pseudodiagonalization and density matrix purification.[26]

Furthermore, we performed a series of restricted Hartree−Fock direct-SCF benchmarks on caffeine ($C_8N_4H_{10}O_2$), cholesterol ($C_{27}H_{46}O$), buckyball (C_{60}), taxol ($C_{45}NH_{49}O_{15}$), valinomycin ($C_{54}N_6H_{90}O_{18}$), CLN025 (a recently reported[27] "artificial protein," $C_{62}N_{11}H_{97}O_{32}$), and olestra ($C_{156}H_{278}O_{19}$) test molecules, depicted in Figure 6, using the 3-21G and 6-31G basis sets and incremental Fock matrix formation procedure.[13] These tests are run on three different GPU systems: one GeForce 8800GTX, one GeForce 280GTX, and three GeForce 280GTX cards that operate in parallel. Because the GeForce 280GTX supports 64-bit DP floating point operations in hardware, we modified the original *J*-matrix code to perform DP accumulation of the J_p^{Hermite} and $K_{\mu\nu}$ matrix elements to avoid additional error introduced by summation of two-electron integrals of widely varying magnitudes. In this case, the ERIs themselves are still evaluated in 32-bit arithmetic on the GPU. All GPU calculations on the 8800GTX card were performed with 32-bit single precision (SP) accuracy. In all of the benchmarks, an integral screening threshold of 10^{-11} au was used.

Tables 3 and 4 summarize the performance results. The "speedup" column represents the speedups achieved in the first (and generally most time-consuming) direct SCF iteration, as compared to GAMESS ver. 11 Apr 2008 (R1) linked to Intel MKL ver. 10.0.3.020 running on a single Intel Pentium D 3 GHz core. Even though our algorithm calculates a significant number of redundant two-electron integrals (6 times more than the number of unique ERIs), the resulting performance is remarkable. In many cases, the GPU code using a single GTX280 card is more than 100× faster than the CPU. Numbers in parentheses in the "GPU" column of Table 3 represent the fraction of one SCF iteration time required to construct the Fock matrix. For small- and medium-sized molecules, most of the SCF time is devoted to Fock matrix formation. However, for large molecules, the linear algebra calls (we use standard dgemm and dsyevd routines from the NVIDIA CUBLAS and Intel MKL library, correspondingly) become a bottleneck. The olestra molecule is a typical example that experiences such performance degradation (Table 5); the linear algebra functions (on the GPU and CPU) take as much as 50% of the Fock matrix computation time (on the GPU). Because general matrix multiplication and diagonalization algorithms scale cubically with the number of contracted basis functions, this impact will be even larger for larger molecules.

In Table 5, we provide detailed timing information for different parts of the SCF procedure. For small and medium molecules, most of the SCF iteration time is spent on the Fock matrix construction that is done on the GPU. However, for the larger molecules, we find that the linear algebra (performed on the GPU and CPU) begins to constitute a major part of the SCF iteration time. We also

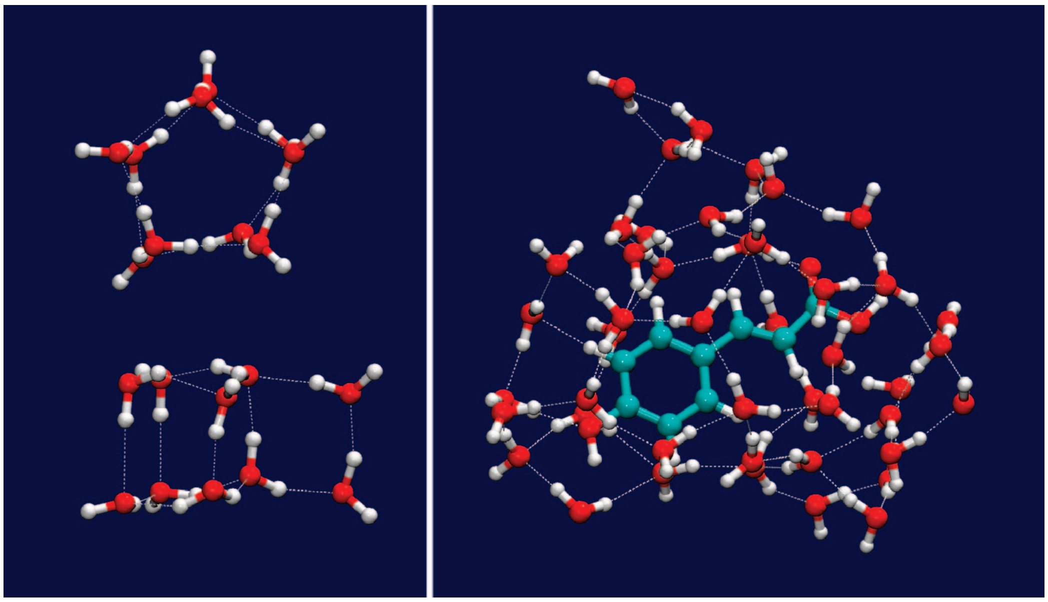


Figure 5. Equilibrated structures from ab initio Monte Carlo simulations at 300 K on the GPU. (Left) Two (upper and lower) representative structures of a cluster with 10 water molecules. (Right) Representative equilibrium structure of the neutral form of the photoactive yellow protein (PYP) chromophore microsolvated with 45 water molecules.

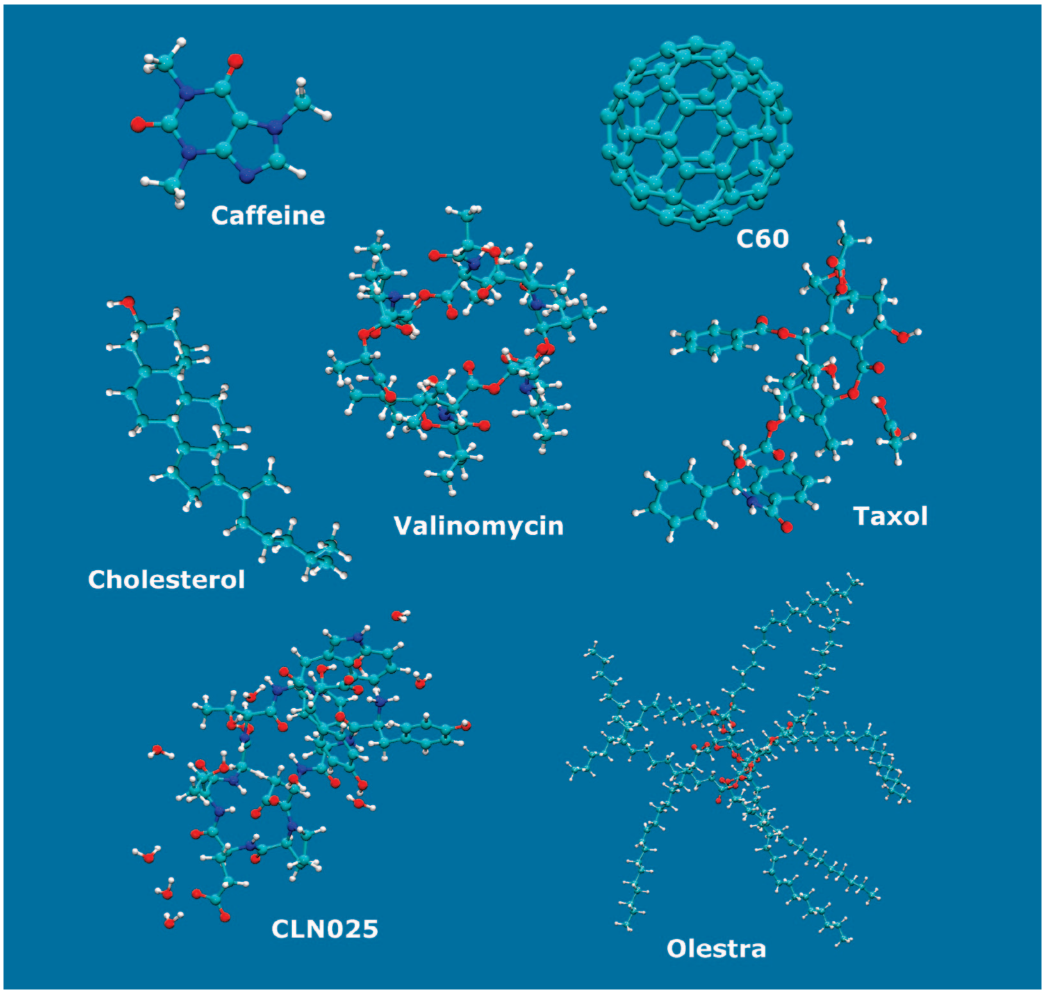


Figure 6. Structures of seven molecules used in the direct-SCF benchmark.

demonstrate that the amount of time required for the presorting of pair quantities is a completely negligible portion of the SCF iteration time. We have pointed out that the dense integral grid resulting from this presorting is critical for efficient GPU computation. However, this step may also be beneficial for traditional CPU imple-

Table 3. Accuracy and Performance of the Direct-SCF Algorithm on One GeForce 280GTX GPU with DP Accumulation of 32-bit Integrals Using the 3-21G Basis Set[a]

	time for the first direct-SCF iteration (s)		electronic energy (au)		
molecule (atoms; basis functions)	GPU	GAMESS	GPU (32 + 64 bit)	GAMESS	speedup
caffeine (24; 146)	0.10 (98%)	4.4	−1605.91830	−1605.91825	44
cholesterol (74; 344)	0.78 (94%)	66.8	−3898.82172	−3898.82189	86
buckyball (60; 540)	3.87 (96%)	353.9	−10 709.08339	−10 709.08392	91
taxol (110; 647)	2.99 (92%)	282.0	−12 560.68286	−12 560.68278	94
valinomycin (168; 882)	5.45 (91%)	729.9	−20 351.98981	−20 351.99038	134
CLN025 (202; 1139)	10.20 (90%)	1405.2	−30 763.37801	−30 763.37882	138
olestra (453; 2131)	19.09 (67%)	12408.1	−49 058.97636	−49 058.97814	650

[a] Reference calculations are performed on a single core of Intel Pentium D 3 GHz processor with full 64-bit DP accuracy. Numbers in parentheses represent the fraction of SCF computation time devoted to Fock matrix formation.

Table 4. Similar to Table *3*, but for 6-31G Basis Set[a]

					GPU speedup		
molecule	electronic energy GPU$_{32+64}$	ΔE_{32} (mH)	ΔE_{32+64} (mH)	GAMESS CPU time (s)	G80	280GTX	3 × 280GTX
caffeine	−1609.37202	0.06	0.10	7.6	19	28	42
cholesterol	−3904.55935	0.35	0.04	113.9	40	56	120
buckyball	−10 721.02377	−2.84	0.80	589.9	37	57	155
taxol	−12 575.56452	0.07	0.29	476.8	46	64	145
valinomycin	−20 371.17351	3.32	0.61	1226.3	67	90	222
CLN025	−30 791.12220	5.94	1.11	2274.7		92	225
olestra	−49 097.16184	36.55	0.44	14 079.2	199	352	696

[a] Errors in the electronic energy (in millihartree) are shown for both 32-bit (ΔE_{32}, G80) and mixed 32 + 64-bit (ΔE_{32+64}, 280GTX) accumulation of the Fock matrix. In all cases, the two-electron integrals are generated on the GPU using 32-bit arithmetic. Speedups (wall clock time, referencing the first SCF iteration) are quoted for the older 8800GTX card, the newer 280GTX card, and a parallelized implementation using three 280GTX cards (3 × 280GTX). All timings refer to the first SCF iteration.

Table 5. Timing Data (in seconds) for Different Steps of the SCF Procedure (the First Iteration, 3-21G Basis Set)[a]

				Fock matrix (first iteration)			
molecule	density	1e Ints	initial guess	pair sort	*J*	*K*	linear algebra
caffeine	0.00	0.01	0.96	0.00	0.02	0.06	0.02
cholesterol	0.00	0.09	0.99	0.04	0.14	0.55	0.05
buckyball	0.01	0.21	1.05	0.09	0.58	3.04	0.15
taxol	0.02	0.30	1.10	0.11	0.43	2.21	0.22
valinomycin	0.03	0.71	1.25	0.19	0.78	3.97	0.48
CLN025	0.06	1.13	1.48	0.29	1.26	7.66	0.93
olestra	0.28	2.88	3.20	0.67	2.54	9.65	5.95

[a] The Fock matrix formation consists of three steps: presorting of the pair-quantities (done on the host CPU), and the *J*- and *K*-matrix formation steps (done on the GPU) as described in the text. These timing data were obtained using a 2.66 GHz Intel Core2 Quad CPU and a GeForce 280GTX GPU.

mentations, because it will improve cache utilization and the efficiency of integral prescreening.

Another important question under consideration is the accuracy one can attain on the GPU. The 8800GTX hardware used in our first calculations natively supports only 32-bit arithmetic operations; thus only 6−7 significant figures of accuracy should be expected in the final result. This may not be enough for large molecules where the absolute total energy exceeds 10^4 hartree, because one wants energy differences to be accurate at the millihartree level or better. The absolute energy error generated by 32-bit GPU arithmetic (and 32-bit accumulation of the Fock matrix elements) as compared to 64-bit calculations on the CPU (GAMESS) using 6-31G basis sets is shown in the column labeled ΔE_{32} of Table 4. The GPU provides submillihartree accuracy for the taxol molecule, but the absolute errors exceed the millihartree level for larger molecules.

The major contribution to the resulting energy error is made by two-electron integrals with relatively large magnitudes, because the mantissa of a 32-bit number is not capable of holding enough significant figures to provide the required accuracy (typically, 10^{-11} hartree). Yasuda has suggested the computation of the largest ERIs on the CPU with DP accuracy to avoid this problem. Both our *J*- and *K*-matrix algorithms already use presorted integral grids, and splitting the grids into DP and SP subgrids, according to the ERI Schwartz upper bound, is straightforward and will not cause any additional computational overhead. This will be investigated in future work. A second contribution to the precision error lies in the summations that lead to the final *J*- and *K*-matrices. Using the DP capability of the 280GTX, we have implemented these summations in DP, while the integrals themselves are still calculated in 32-bit precision. This leads to the errors listed in the column of Table 4 labeled ΔE_{32+64}. The use of DP accumulation improves the accuracy sometimes by better than an order of magnitude (at negligible computational cost), and calculations on molecules as large as olestra are within "chemical accuracy", typically defined as 1 kcal/mol.

Additionally, because many of the large two-electron integrals are produced by core orbitals that are chemically inert, their corresponding contributions may be expected to cancel (at least partially) when energy differences are computed, as previously shown by Kermes et al. in calculations of the correlation energy.[4] Direct tests performed on a GeForce 8800GTX card (32-bit integrals and 32-bit accumulation) on a cholesterol molecule confirm this. We

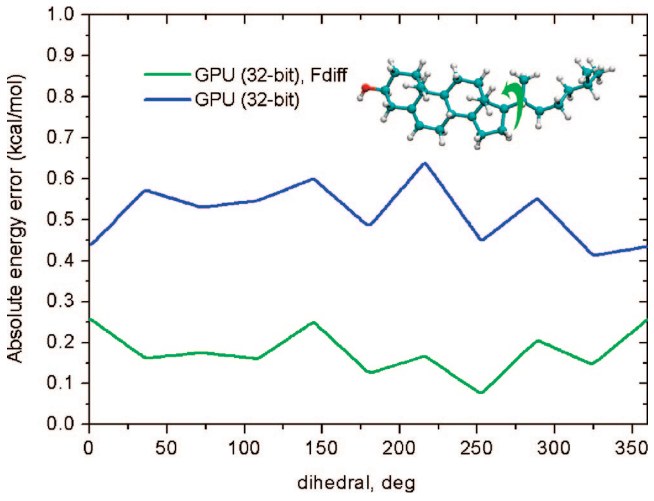


Figure 7. The absolute energy error produced by 32-bit calculations of the cholesterol (3-21G basis set) molecule on the 8800GTX GPU, as the molecule is twisted around one of its bonds. Conventional (blue) and incremental (green) Fock matrix formation approaches are presented. The reference 64-bit data were generated by GAMESS.

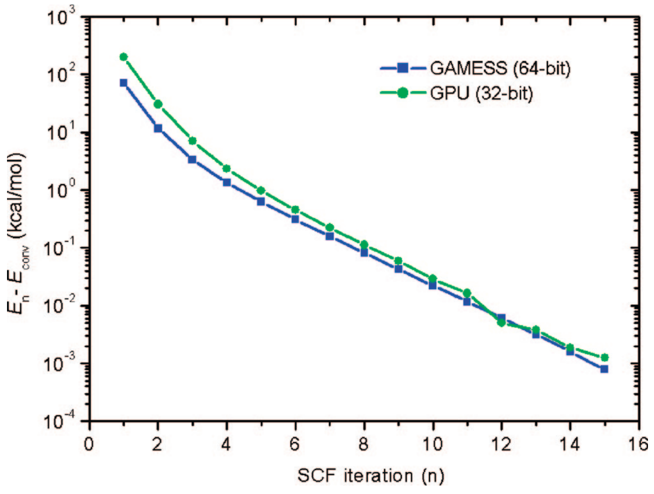


Figure 8. Convergence of the SCF energy for the cholesterol molecule (3-21G basis set) using the 8800GTX GPU and CPU. The convergence behavior is nearly identical, showing that the single precision evaluation of the integrals has no adverse effect on SCF convergence.

twisted the molecule around one of the bonds (see inset in Figure 7) and optimized the molecular geometry with DP accuracy, keeping the corresponding dihedral fixed. The corresponding error generated by 32-bit arithmetic for conventional (blue) and incremental (green) Fock matrix formation techniques is displayed in Figure 7. Two characteristic features can be seen in this figure: (1) the incremental approach much better represents 64-bit results, and (2) in both curves, the deviation from the mean is smaller than the absolute values of the errors (presumably due to cancelation of large integrals).

Finally, we investigated the impact of SP accuracy on the SCF convergence behavior using the cholesterol molecule as an example and GeForce 8800GTX GPU, comparing to the 64-bit GAMESS calculation results. Figure 8 presents the SCF energy error, $E_n - E_{conv}$, as it gradually declines at each subsequent SCF iteration, where E_{conv} is the converged energy, which is different for GPU and CPU. The slight initial offset is due to the less accurate initial guess algorithm used in our program. We disabled SCF acceleration both in our code and in GAMESS. The curves in Figure 8 track each other nearly perfectly, meaning there is no convergence instability introduced by 32-bit GPU arithmetic, up to $\Delta E \approx 0.1$ kcal/mol. Of course, eventually unavoidable precision noise will produce less regular behavior if one attempts to converge the energy beyond the 7−8 significant figures that can be obtained with SP arithmetic.

Conclusions

We have demonstrated that it is possible to achieve two order-of-magnitude increases in efficiency for an entire quantum chemistry calculation using GPUs. The speedups we report here (up to 650×) are for the entire direct SCF iteration, and thus representative of the speedup that can be expected for the full SCF calculation. Because of the exceptional performance, we were able to carry out ab initio Monte Carlo simulations for molecular systems with over a hundred atoms. We obtained this level of performance by developing the algorithms from the ground up for use on stream processing architectures. For example, our algorithm calculates many redundant ERIs in recognition of the fact that it can be more efficient to do redundant computation than random memory accesses and/or interthread communication on GPUs. We have further demonstrated the possibility of parallelization over multiple GPUs, which leads to even more significant performance gains.

We showed that the accuracy achieved for energy differences can be significantly higher than that expected from the absolute energies because of systematic cancelation of precision errors largely due to integrals involving the chemically inert core electrons. We demonstrated that using double precision only to preserve accuracy during summation of the density-ERI products provides “chemical accuracy” in all of the absolute energies reported here (as compared to full double precision calculations on the CPU) at negligible computational cost.

The advances reported here will make large-scale quantum chemistry calculations routine and will be especially useful in increasing the utility of ab initio molecular dynamics (AIMD) simulations. We are currently working on an implementation of analytic gradients that is a prerequisite for efficient use in the AIMD context.

Acknowledgment. This work was supported by the National Science Foundation (CHE-06-26354 and DMR-03025939).

References

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

(2) Stone, J. E.; Phillips, J. C.; Freddolino, P. L.; Hardy, D. J.; Trabuco, L. G.; Schulten, K. *J. Comput. Chem.* **2007**, *28*, 2618.

(3) Anderson, A. G.; Goddard, W. A., III.; Schroder, P. *Comput. Phys. Commun.* **2007**, *177*, 265.

(4) Kermes, S.; Olivares-Amaya, R.; Vogt, L.; Shao, Y.; Amador-Bedolia, C.; Aspuru-Guzik, A. *J. Phys. Chem.* **2007**, *112A*, 2049.

(5) Ufimtsev, I. S.; Martinez, T. J. *Comput. Sci. Eng.* **2008**, *10*, 26.

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

(7) Yasuda, K. *J. Comput. Chem.* **2008**, *29*, 334.

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

(9) Brown, P.; Woods, C.; McIntosh-Smith, S.; Manby, F. R. *J. Chem. Theory Comput.* **2008**, *4*, 1620.

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

(11) 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.

(12) NVIDIA CUDA. Compute Unified Device Architecture Programming Guide Version 2.0; NVIDIA Developer Web Site; http://developer.download.nvidia.com/compute/cuda/2_0/docs/NVIDIA_CUDA_Programming_Guide_2.0.pdf (acccessed September 15, 2008).

(13) Almlof, J.; Faegri, K.; Korsell, K. *J. Comput. Chem.* **1982**, *3*, 385.

(14) Whitten, J. L. *J. Chem. Phys.* **1973**, *58*, 4496.

(15) White, C. A.; Head-Gordon, M. *J. Chem. Phys.* **1996**, *104*, 2620.

(16) Gill, P. M. W. *Adv. Quantum Chem.* **1994**, *25*, 141.

(17) Shao, Y.; Head-Gordon, M. *Chem. Phys. Lett.* **2000**, *323*, 425.

(18) Ahmadi, G. R.; Almlof, J. *Chem. Phys. Lett.* **1995**, *246*, 364.

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

(20) This double precision accumulation is only implemented on the 280GTX card. Results reported using the 8800GTX carry out the accumulation in single precision arithmetic.

(21) The bandwidth reported in Table 1 is an effective bandwidth that may be higher than the actual bandwidth because of caching (of texture memory).

(22) Ko, C.; Levine, B.; Toniolo, A.; Manohar, L.; Olsen, S.; Werner, H.-J.; Martinez, T. J. *J. Am. Chem. Soc.* **2003**, *125*, 12710.

(23) Wales, D. J.; Hodges, M. P. *Chem. Phys. Lett.* **1998**, *286*, 65.

(24) Day, P. N.; Pachter, R.; Gordon, M. S.; Merrill, G. N. *J. Chem. Phys.* **2000**, *112*, 2063.

(25) Scuseria, G. E. *J. Phys. Chem.* **1999**, *103A*, 4782.

(26) Daniels, A. D.; Scuseria, G. E. *J. Chem. Phys.* **1999**, *110*, 1321.

(27) Honda, S.; Akiba, T.; Kato, Y. S.; Sawada, Y.; Sekijima, M.; Ishimura, M.; Ooishi, A.; Watanabe, H.; Odahara, T.; Harata, K. *J. Am. Chem. Soc.* **2008**, *130*, 15327.

CT800526S