New Multithreaded Hybrid CPU/GPU Approach to Hartree−Fock

Andrey Asadchev and Mark S. Gordon*

Department of Chemistry, Iowa State University, Ames, Iowa 50011, United States

ABSTRACT: In this article, a new multithreaded Hartree−Fock CPU/GPU method is presented which utilizes automatically generated code and modern C++ techniques to achieve a significant improvement in memory usage and computer time. In particular, the newly implemented Rys Quadrature and Fock Matrix algorithms, implemented as a stand-alone C++ library, with C and Fortran bindings, provides up to 40% improvement over the traditional Fortran Rys Quadrature. The C++ GPU HF code provides approximately a factor of 17.5 improvement over the corresponding C++ CPU code.

I. INTRODUCTION

As computer hardware becomes more sophisticated and complex and programming languages, compilers, and software patterns mature, it becomes necessary to re-engineer software written during the 1980s or earlier in order to take advantage of modern hardware and language features. Unlike older hardware, modern processors have more and more cores, multithreading becomes more and more important, and novel architectures such as graphical processor units (GPU) enter mainstream scientific computing.

"Legacy" programs often do not take into account low-level details of modern processors such as multilayer cache organization, pipelines, and SIMD (single instruction, multiple data) units.[1] As a result of poor cache performance, programs waste CPU cycles, moving data at the expense of actual computations. Failure to take advantage of the SIMD architecture, due for example to unfavorable control structures and memory access patterns, can lead to as much as a 50% drop in performance. Parallel execution within a single node presents a challenge as well: computational tasks in legacy codes usually run as processes within a message passing environent, rather than as threads, limiting the utility of shared memory and fast interthread communication offered by a multithreaded environment,[2] resulting in replicated memory which puts additional strain on memory cache and bus. OpenMP can at times solve the problem of multithreading in legacy codes, provided that internal subroutines are thread-safe, which is not always the case.

There are several projects that aim to address shortcomings of legacy code, implementing the entire suite of quantum chemistry algorithms using new programming techniques, typically in C++, for example, PSI[3] and MPQC.[4]

This paper describes a new approach to the Hartree−Fock method that is meant to address the requirements of modern hardware and software, from a low-level two-electron Rys Quadrature[5] implementation to multithreaded parallel Fock matrix construction and GPU implementation. The method described here does not aim to replace an entire software package but rather to provide an independent library that can be used to replace or augment existing Hartree−Fock and integral implementations. This paper is organized as follows: Section II presents the developments associated with the Rys quadrature algorithm, including automatically generated code and the requirements for quartets that contain low and high angular momentum quantum numbers. Section III considers various aspects of the Fock matrix construction. The C++ CPU implementation is presented in section IV, while the corresponding GPU implementation is discussed in section V. Section VI considers the performance of the new algorithms, and conclusions are drawn in section VII.

II. RYS QUADRATURE IMPLEMENTATION

Modern computers have complex architectures and pipelines, making it difficult for an application programmer to write efficient assembly code. Fortunately, modern compilers are able to produce efficient code if several constraints are met:

- memory access has a favorable alignment, for example, 16 bytes for the current Intel Core architecture
- nonoverlapping segments of memory are flagged as such, using a special type declaration or compiler pragmas, e.g., the C99 restrict keyword
- innermost loops do not have control statements, such as if or equivalent
- short innermost loops have bounds that are known at compile time
- innermost memory accesses are contiguous, i.e., they have a stride of one

Of course, most application programmers (e.g., computational chemists) would not endeavor to write assembly code. However, nontrivial algorithms, such as the Rys Quadrature that is used for two-electron integrals in quantum chemistry codes,[5] still require a significant amount of code to accommodate the compiler requirements. Writing such codes manually can be time-consuming and error-prone, regardless of the language used. However, there are a number of code generators that can greatly simplify the task through automation. Using code generators to implement integral routines is not new; for example, the excellent LIBINT[6] library was implemented using a code generator. For this project, the Python Cheetah code generator[7] was chosen for the following reasons:

Received: June 23, 2012
Published: September 10, 2012

- Generator statements are embedded directly into the source code template, regardless of language, which, for example, can be C++, C, or Fortran.
- The generator statements are just regular Python statements.
- Any Python module can be imported and used in the generator environment, including several symbolic algebra packages, such as sympy[8] and Sage,[9] which provide an interface with Mathematica[10] and other computer algebra systems.

The strategy toward implementing the Rys Quadrature algorithm is as follows:[5b]

- Certain integrals, particularly those over basis functions with low angular momentum quantum numbers, e.g., $L = 0$ (s) and $L = 1$ (p), and consequently small shell quartet block sizes (e.g., there 64 integrals in a (sp sp|sp s) quartet) and short polynomial expressions are best computed directly using the entire polynomial expression at once, rather than via two-dimensional intermediates.
- General integrals over basis functions with higher angular momentum quantum numbers have prohibitively long polynomial expressions and must be assembled from two-dimensional intermediate integrals via so-called recurrence and transfer relations.[5]

II.A. Rys Quadrature. The main idea in the Rys Quadrature is to represent a six dimensional integral

$$(ij|kl) = \iint \varphi_i(1)\, \varphi_j(1)\, \frac{1}{r_{12}} \varphi_k(2)\, \varphi_l(2)\, \mathrm{d}r_1\, \mathrm{d}r_2$$

as a product of three two-dimensional integrals I_x, I_y, and I_z

$$\sum_a^N I_x(a)\, I_y(a)\, I_z(a)\, W(a)$$

summed over an exact N-point numerical quadrature with roots a and weights W. The two-dimensional integrals I_x, I_y, and I_z are evaluated using recurrence and transfer equations. The exact formulation of the equations can be found in the original Rys paper.[5]

Each primitive integral above corresponds to a single contraction. When evaluating contracted shells, the full expression becomes

$$(ij|kl) = \sum_a^A \sum_b^B \sum_c^C \sum_d^D C_a C_b C_c C_d\, I(a, b, c, d)$$

where the bounds of the summation are shell contraction orders, C are the contraction coefficients, and $I(a,b,c,d)$ are primitive uncontracted integrals.

II.B. Small Angular Momentum Integrals. If an integral expression $(ij|kl)$ is simple enough, it can be expanded directly into a polynomial, removing the need to compute and store two-dimensional integrals. Doing this also has the benefit of providing the compiler with enough information to enable aggressive optimization. Furthermore, expanded expressions can be filtered through a computer algebra system, like Mathematica, simplified, and organized together arbitrarily. The above strategy is not, however, computationally favorable if the integral expression is large, since the large amount of produced code tends to overflow the data and program cache and can adversely impact performance.

The polynomial expressions are expanded from recurrence and transfer formulas as follows:

- The symbolic algebra Python package, sympy, is used to build a raw polynomial expression from terminal terms, the starting and ending values in the Rys recursive formulas, using recurrence and transfer formulas.
- The raw polynomial expressions are piped into Mathematica through Sage, a Python package that provides interfaces with popular computer algebra systems. Mathematica simplifies the raw polynomial expressions and performs a common subexpression elimination (CSE) to pull out common terms.
- The number of common terms can be quite large, generally larger than the number of registers (16 for the current generation of Intel x86−64 processors). Simplified expressions are reordered to maximize register reuse.
- Simplified expressions are stored as a plain text Python dictionary dump, together with the terminal terms and common terms expressions.
- Since the expression order may have changed, values might have to be permuted to restore the original integral order

In the expression dictionary dump, each integral block expression has a lookup key, which is a collection of four strings, corresponding to shell symbols. The first entry is the dictionary of terminal symbols (those with empty expressions) and common terms (those with nonempty expressions). The next entry is the list of individual functions in the integral block, specified by their l, m, n angular momentum quantum numbers. Each function has a polynomial expression as a string and a list of required terms, both terminal and common. Once they have been loaded, the expressions can be read from the dictionary and implemented inside the loop over quadrature roots.

The algorithm is fairly straightforward: the primitive integrals, depending on individual contractions of the basis functions i, j, k, and l and the corresponding roots and weights a and w of the integral shells P, Q, R, and S, are evaluated inside the four nested loops corresponding to primitives. The actual integral construction and summation over the roots is handled by a function specialized for the shell types (e.g., s, sp, d, etc.) of the shells P, Q, R, and S, i.e., the actual implementation of the polynomial expressions. The bra and ket primitives are precomputed to reduce the number of exponent computations. Once the integral is assembled for all contractions, it is then reordered to restore the correct order. Finally, the amount of memory required is determined by the integral quartet size. For small integral blocks, this amount of memory is small enough to completely fit in L1 cache.

Through some experimentation, it was found that integral blocks with approximately 160 functions, e.g., (fsp|sps), where sp refers to a hybrid sp shell, and below tend to have the best balance between performance and code size. Large integral quartets, for example a full SP quartet, tend to increase code size and compilation time dramatically, without a noticeable performance benefit.

II.C. General Integrals. General integrals with high angular momentum quantum numbers are best computed using a traditional approach via two-dimensional intermediates. However, the details of the present implementation are significantly different from others and are best described using the pseudo algorithm in the C++ Listing 1. The lines in the pseudocode after "//" are comments.

```
// P,Q,R,S are the Shell objects that contain all
// information such as contracted Gaussians, angular //
momentum L, etc.

N = (P+Q+R+S)/2 + 1 // number of quadrature roots
bra (P,Q); // bra primitives

for k,l in (R,S) { // ket contractions
    ket (k,l); // ket primitives
    for i,j in (P,Q) { // bra contractions
        // contraction factor
        C = bra(ij)*ket;
        if (C < cutoff) continue; // screening
        // roots and weights
        (a,w) = roots(bra(ij), ket);
        (Gx,Gy,Gz) = recurrence(bra(ij), ket);
        (Ix(K),Iy(K),Iz(K)) = transfer(Gx,Gy,Gz);
        ++K;
    }
}

for r,s in (R,S) { // R,S functions
    Ix = Ix(:,:,x(r),x(s),:)
    Iy = Iy(:,:,y(r),y(s),:)
    Iz = Iy(:,:,z(r),z(s),:)
    for k in K { // contractions
      for a in N { // roots
        // form integrals
        G(0) += Ix(Li,Lj,k)*Iy(0,0,k)*Iz(0,0,k)
        G(1) += Ix(0,0,k)*Iy(Li,Lj,k)*Iz(0,0,k)
        ...
        G(M-1) += ...
      }
      I(0:M) += C*G
      for a in N { // roots
        ...
      }
      I(M,...) += C*G
      ...
    }
    transform(G)
}
```

Listing 1. Bra Quadrature

The main ideas of the pseudocode are as follows:

- The bra, $\langle PQ|$, exponential factors are precomputed, to avoid a quartic number of exponent computations.
- Inside the individual primitive loops, the roots are computed to form recurrence intermediates that in turn are used to generate the final two-dimensional integral via transfer relations for a given contraction K.
- Once all of the two-dimensional integrals are formed, they are transformed into the final electron repulsion integral (ERI). Details of the implementation are somewhat involved and are explained below.

II.D. Bra Kernel. In calculating the shell functions, there is not a simple *runtime* relationship between a numerical index i of a function in a shell, i.e., $f(i) = (l,m,n)$, and the individual angular momentum quantum numbers l, m, and n. For example, in a d-shell, (2,0,0), (0,2,0), (0,0,2), (1,1,0), ..., it is not possible to resolve a general loop index, say $i = 3$, into its components (1,1,0) without some sort of lookup table. Therefore, the angular momentum components could be tabulated and looked up during runtime. However, indirect indexing due to the use of a lookup table prevents effective optimization by the compiler. In the outer loops, there is little overhead due to indexing, but for the innermost loops, corresponding to the bra part, the indexing overhead becomes significant. In order to avoid lookup tables in the bra loops, all of the indexes on the bra side must be available during compilation. This is fairly easy to accomplish using a code generator, for example, the same Python Cheetah code generator described above.

Different kernels, corresponding to different numbers of roots, can also be generated using the code generator. However, since the code described here was written using C++, this becomes unnecessary, since the C++ template meta language can be used to accomplish the same result much more effectively. The number of functions computed in any given block may be too large for the compiler to handle effectively, primarily because there is only a small number of registers. Therefore, the entire list of bra functions is broken up into blocks of M functions each. After some experimentation, an M value of 10 was found to be the most effective.

It should be noted that for a given integral block, the bra subsection is evaluated entirely for each given ket index, for all contractions. This allows the code to generate the entire integral block piecewise and transform individual bra blocks one by one, without forming the entire integral. The utility of this approach is described in terms of the Fock matrix construction in more detail below.

Throughout the entire computation, the three innermost indices correspond to roots and bra indices that are known at compile time, delegating the task of the actual optimization to the compiler.

III. FOCK MATRIX CONSTRUCTION IMPLEMENTATION

The construction of the Fock matrix[11] from the integrals and the density matrix can be split into two parts: higher level iterations over the shell quartets and lower-level contraction of the density matrix with the integrals to produce a Fock matrix block that corresponds to a particular integral quartet (i,j,k,l).

The general approach to contracting an integral I with the density matrix D is outlined in Listing 2. The coefficient C refers to Coulomb term coefficients, and X refers to exchange term coefficients. For plain Hartree–Fock (HF) using 8-fold symmetry, those coefficients would be 4 and −1 respectively, but for methods that modify the Fock operator, e.g., density functional theory (DFT), those coefficients may be different.

```
// I(i,j,k,l) are already computed: ints
// D(i,j) is density matrix
for l in S { // ket indices
  for k in R {
    for j in Q { // bra indices
      for i in P {
        F(i,j) += C*D(k,l)*I(i,j,k,l)
        F(k,l) += C*D(i,j)*I(i,j,k,l)
        F(i,k) += X*D(j,l)*I(i,j,k,l)
        F(i,l) += X*D(j,k)*I(i,j,k,l)
        F(j,k) += X*D(i,l)*I(i,j,k,l)
        F(j,l) += X*D(i,k)*I(i,j,k,l)
      }
    }
  }
}
```

Listing 2. Fock contraction

The following modifications are made to improve performance:

- The density and Fock matrix blocks, corresponding to a particular combination of two shells, are stored contiguously to optimally use cache locality. This is addressed in more detail in the next subsection.
- The innermost loops are relatively short, and for the best performance the loop sizes are known at compile time.

- The memory usage is dominated by integral storage. However, since the integrals are being formed block by block, the entire integral never needs to be stored. Instead, each bra tile is contracted with the appropriate density tile to form a Fock tile piece by piece.
- Since small angular momentum integrals are formed at once, a specialized version to handle that case is implemented as well.

The kernel version of the code specialized for the entire bra—ket, i.e. small angular momentum integrals, is essentially Listing 2 with loop bounds that are known at compile time, to provide the compiler with the information needed to enable aggressive optimization. For example, when compiling a Fock kernel corresponding to a (ss|ss) quartet, all the loop bounds are 1, and the compiler will optimize out the loops altogether.

The kernel version specialized for partial Fock contraction is implemented as a function object that "remembers" indices *k* and *l* (see the pseudocode in Listing 3). For each integral bra tile being formed, the apply function is called. With each transformation, the internal indices are updated to maintain the correct state.

```
class Fock {
    k,l = 0 // initial state
    apply(I(P,Q)) {
      for j in Q { // bra indices
        for i in P {
          F(i,j) += C*D(k,l)*I(i,j)
          ...
          F(j,l) += X*D(i,k)*I(i,j)
        }
      }
      ++k // update state indices
      ++l
    }
}
```

Listing 3. Tiled Fock contraction

III.A. Blocking Fock/Density Matrix. The utility of block partitioning matrix computations is well understood.[12] However, partitioning the Fock matrix into blocks is not straightforward since the block nature of the Fock matrix is determined by the shell order in the basis set. However, the basis set may be sorted in such a way as to group same-size shells together. Reorganizing the basis set alone does not give the Fock matrix a uniform block structure since the basis set typically contains s, p, ... shells. This can be overcome by considering the entire Fock matrix to be a meta-matrix consisting of submatrices, each with a uniform block structure, determined by the corresponding shells. Consider a graphical depiction of such a matrix, as shown in Figure 1, showing a hypothetical meta-matrix with a nonuniform block structure organized as uniform matrices. The black lines designate the individual shell block boundaries, with all of the elements inside the block being in a contiguous memory segment. The red graphs show the consecutive layout of blocks in memory, with connected blocks being in the same memory segment in that given order. The blue lines designate the borders of submatrices, in which all blocks *within* those submatrices are of uniform dimensions.

If the programming language constructs allow, the meta-matrix can be given the usual matrix semantics that map individual element access to a specific block in the appropriate submatrix. In C++, this can be accomplished by defining operator () (*i*,*j*). The effect is that a complex meta-matrix can

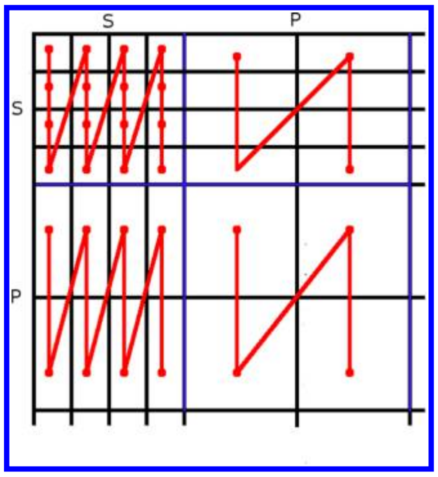

Figure 1. Meta-matrix with block structure.

have all three characteristics: submatrix, block, and element-wise access.

The second benefit of organizing the basis set according to shells is to allow efficient evaluation of multiple similar shell quartets on highly parallel architectures, such as graphical processing units (GPUs). If the shells are grouped together according to coefficients and exponents, as well as the angular momentum quantum numbers, then evaluation of such a block is guaranteed to have the same data except for the Cartesian centers.

If the Fock matrix needs to be sorted for computational efficiency, the density matrix can be permitted to reflect the desired order. Likewise, if other parts of the program expect the Fock matrix to be in a different order, once formed, the Fock matrix can be unsorted. This is especially relevant if the Fock matrix is to be used by external programs which may not necessarily sort the basis set.

III.B. Collapsing Fock Algorithm Loops. The regular Fock matrix algorithm, Listing 4, becomes cumbersome if the work has to be divided among different parallel domains and different processors/accelerators. To make the work distribution easier to implement and more efficient, the four nested loops of the Fock algorithm can be collapsed into a single queue-like generator, as illustrated in Listing 5. The basic idea is to map a single index back to four loop indices.

The advantage of using a queue rather than nested loops is that a queue can be transparently and easily parallelized. For the Fock algorithm, the queue tuples are generated on the fly, rather than stored at the expense of N^4 tuples.

The internal counter employed in the queue can be a generic counter, for example, a distributed read—modify—write counter, which allows one to easily transform a seemingly single-node queue into a distributed queue.

```
for l in N {
  for j in N {
    // loop bounds account for 8-fold symmetry
    for k in max(l,j):N {
      for i in j,k+1 {
          ...
      }
    }
  }
}
```

Listing 4. Fock looping

```
class Queue {
    // initial values
    counter = 0;
    last = 0;
    (i,j,k,l) = (0,0,0,0);
    next() {
        next = (i,j,k,l);
        end = (counter++)+1; // advance counter
        for last:end {
            if (empty()) throw exception; // signal if empty
            next = (i,j,k,l);
            i += 1; // i loop
            advance = (i >= (k+1)); // k loop
            if (advance) {
                k += 1;
                i = j;
            }
            advance = advance and (k == N); // j loop
            if (advance) {
                j += 1;
                k = max(j,l);
                i = j;
            }
            advance = advance and (j == N); // l loop
            if (advance) {
                l += 1;
                j = 0;
                k = max(j,l);
                i = j;
            }
        }
        last = end;
        return next;
    }
}
...
while (true) {
    try: (i,j,k,l) = queue.next(); // get next tuple to evaluate
    catch: break; // the end, break from the loop
    ...
}
```

Listing 5. Fock task queue

III.C. Exchanging bra/ket Order. Most of the integral algorithms, including the Rys Quadrature, prefer the general integral (pq|rs) over shells P, Q, R, and S to be sorted such that $P \geq Q$, $R \geq S$, and $P \geq R$. Exchanging the order inside the integral code adds complexity and has a performance penalty. But for the purposes of a Hartree−Fock code, exchanging the order of the quartet indexes alone and of the corresponding submatrices is sufficient. However, the screening must be done before changing the order if one is using an unmodified screening loop structure.

III.D. Normalization Coefficients. Integrals over functions with angular momentum higher than the P shell must be normalized. The normalization can either be done in the integrals themselves or by absorbing the normalization coefficients into other terms. The advantage of removing normalization coefficients from the integrals is that the integral code is simpler when it is devoid of normalization coefficients.

For the purposes of the HF algorithm, the following approach can be used to shift the normalization coefficients N_i from the integrals to the Fock (F) and density (D) matrices to form normalized matrices F^* and D^*:

$$F_{ij} = (N_iN_jN_kN_l(ij|kl))D_{kl} \tag{1}$$

$$D_{kl}^* = (N_kN_l)D_{kl} \tag{2}$$

$$F_{ij}^* = (ij|kl)D_{kl}^* \tag{3}$$

$$F_{ij} = (N_iN_j)F_{ij}^* \tag{4}$$

Therefore, normalization can be handled by first normalizing the density matrix, then performing the regular Fock algorithm and normalizing the resulting Fock matrix.

III.E. Multithreaded Implementation. A multithreaded Fock algorithm allows one to reduce the memory overhead by maintaining only a single copy of the Fock and density matrices per node. The density matrix, which is read-only, does not need to be protected from conflicting updates. However, the Fock matrix is subject to conflicting simultaneous updates from multiple threads, known as race conditions. For example, evaluating integral quartets (i,j,k,l) with values (1,1,4,4) and (1,1,3,3) requires an update to the Fock elements $F(k,l) = F(1,1)$ in both cases. If the two integral quartets are to be evaluated by two distinct threads, the access to the Fock elements must be synchronized so as to avoid race conditions.

There are a number of ways this can be accomplished. For the best performance, an approach using a matrix block lock/mutex (mutual exclusion object) was chosen. Since the entire Fock matrix can be arbitrarily partitioned into blocks, each block can be given its own mutex that is locked when a thread is ready to update the corresponding block. However it is wasteful to lock the entire Fock matrix block while the integrals are being computed and contracted. A better alternative is for each thread to maintain up to six Fock buffers, $F(i,j)...F(j,l)$, which can then be accumulated into the main shared Fock matrix. The algorithm outline is in Listing 6.

```
for (i,j,k,l) in ERI {
    // thread buffers
    Submatrix f(i,j), ..., f(j,l)
    (f(i,j), ..., f(j,l)) = Contract(Integral(i,j,k,l), D)
    // accumulates submatrix
    for f(m,n) in ((f(i,j), ..., f(j,l))) {
        F.lock(m,n)
        F(m,n) += f(m,n)
        F.unlock(m,n)
    }
}
```

Listing 6. Shared Fock updates

IV. C++ IMPLEMENTATION DETAILS

Since the approach detailed in the current work is written in C++, the following libraries and techniques are available:

- Boost libraries[13]
- C++ meta-programming,[14] including boost::enable_if[15] and boost::mpl[16]
- C99 preprocessor and Boost Preprocessor[17]
- OpenMP[18]

The code relies heavily on template meta-programming to accommodate compile time requirements of the integral and Fock kernels and to reduce the amount of boiler-plate copy/paste. Various preprocessor tricks of the Boost Preprocessor are used heavily as well. For example, to "transform" a runtime value into a compile time value, the Boost Preprocessor can be used to generate the transformation, e.g., Listing 7. BOOST_PP_SEQ_FOR_EACH_PRODUCT will apply a macro ERI for each Cartesian quartet of shell types, automatically creating all possible handlers for a quartet followed by a special case if the quartet is invalid, i.e., not one of the TYPES in the listing below.

The multithreading was implemented using OpenMP. While the Boost Thread library is much more powerful and versatile than OpenMP, only a subset of the multithreading constructs was needed to make the code multithreaded, primarily the loop counter synchronization and mutex constructs. In addition to

```
#define TYPES (SP)(S)(P)(D)(F)//...

void runtime(Quartet quartet) {
    type a = quartet[0];
    type b = quartet[1];
    type c = quartet[2];
    type d = quartet[3];

#define ERI(r, types) \
    if (a == BOOST_PP_SEQ_ELEM(0, types) && \
        b == BOOST_PP_SEQ_ELEM(1, types) && \
        c == BOOST_PP_SEQ_ELEM(2, types) && \
        d == BOOST_PP_SEQ_ELEM(3, types)) { \
        typedef shell_pair<BOOST_PP_SEQ_ELEM(0, types),      \
                           BOOST_PP_SEQ_ELEM(1, types)> bra; \
        typedef shell_pair<BOOST_PP_SEQ_ELEM(2, types),      \
                           BOOST_PP_SEQ_ELEM(3, types)> ket; \
        eri<bra,ket>(quartet);
    }
    BOOST_PP_SEQ_FOR_EACH_PRODUCT(ERI, (TYPES)(TYPES)(TYPES)(TYPES))
    {
        throw invalid_quartet();
    }
}
```

Listing 7. Using preprocessor

the above-mentioned libraries, other miscellaneous components from the Boost and Standard Template Library are used throughout.

V. GPU IMPLEMENTATION

There have been various GPU implementations for electron repulsion integrals, for example, the McMurchie-Davidson[19,20] and Rys Quadrature[5b,21,22] approaches. Early on, the GPU implementations primarily targeted single precision computations with s and p functions only, using either CUDA C or accelerator statements. The current generation of GPU hardware has a much smaller time difference for single vs double precision, making the case for single precision less obvious.

The authors have utilized double precision exclusively to reproduce the CPU results and to go well beyond s and p functions. The GPU implementation was done using NVIDIA CUDA technology. In developing the GPU implementation of the Hartree–Fock method, the following factors are considered:

- High angular momentum and low angular momentum/highly contracted integrals are different in nature and warrant different implementation approaches.
- The integral kernels must be able to evaluate many batches of integrals in one launch. By sorting according to the basis set, a large number of quartets, differing only in the atom centers, but not in shell primitives, can be generated.
- The integrals must be contracted with the density D as soon as possible to reduce the memory overhead from n^4 to n^2 where n is the shell size order, e.g., $n = 6$ for a Cartesian d shell. Therefore, the entire integral quartet is never written into the device memory.
- Contracting integrals with the density directly results in race conditions which must be accounted for.
- Integral batches which cannot be evaluated on the device must be done on the host.

The current Fermi hardware has 32 768 registers and 48 KB of shared memory. The number of concurrent thread blocks is eight. A typical integral kernel will use ~60 registers per thread and 6 KB of shared memory. Therefore, up to eight thread blocks can be executed simultaneously, 64 threads each. The 64 threads are executed in warps, with 32 threads per warp. The threads in each warp are implicitly synchronized, but their execution is not implicitly synchronized with the other warp. In essence, a warp can be thought of as an independently executing unit. This fact can be used to partition work along the warp or subwarp boundaries.

The development of the integral kernels closely follows the CPU version: the implementation is split into general and low angular momentum kernels. The low angular momentum kernels are parallelized over the contraction loop. In both cases, an efficient implementation requires that the type of integral be known at compile time. This is handled by implementing integrals using C++ templates, with the bra–ket type being a compile time parameter and the shell exponents and coefficients a runtime parameter. The shells, centers, and quartet lists are stored in the device memory. Regardless of the implementation, each kernel loads all three sets of data and forms the corresponding bra–ket primitives in shared memory.

V.A. General Integral Kernel. The general integral kernel is applicable to most combinations of contraction order and bra–ket types. While the general kernel may not perform equally well for some combinations, these combinations can be handled by specialized kernels chosen at runtime.

There are multiple ways one can approach the problem of implementing a general Rys Quadrature algorithm on the GPU architecture. The approach taken here is as follows:

- All roots and weights are computed first and stored in the shared memory first.
- Each thread is assigned a 3-D index corresponding to the recurrence and transfer computations it will perform, where the x index maps to an angular momentum, the y index maps to one of the three Cartesian coordinates, and the z index maps to root.
- The x index corresponds to either a bra or a ket index. Let L_{ab} be the total bra angular momentum $L_a + L_b$ and N the number of roots. In general, $(L_{ab} + 1)3N$ threads are needed to evaluate recurrence, and $(L_{ab})3N$ threads are needed to evaluate transfer, with the higher value being the total number of threads required.

In certain cases, e.g., if one or more of the shells are S ($L = 0$), not all of the recurrence and transfer computations are needed; then, the number of threads will be smaller than $(L_{ab} + 1)3N$. The computations are independent of one another in the y and z indices but are dependent on the previous results of a thread with a different x index (and the same y and z indices). Consider the graphical depiction (Figure 2) of a transfer relation to form a $(fd|$ bra intermediate from a $(hs|$ bra. The y axis corresponds to the first center of the bra, and the x axis corresponds to the second center of the bra. Each index (p,q) depends on $(p + 1, q - 1)$ and $(p, q - 1)$. For example, the

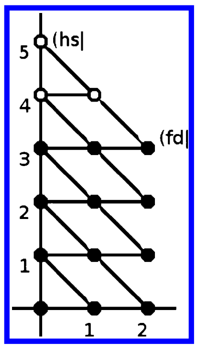

Figure 2. Transfer diagram to form $(fd|$ bra.

index (3, 2) depends on (3, 1) and (4, 1) which in turn depend on (3, 0), (4, 0), and (5, 0). The intermediate (4, 2), computed by thread 4, depends on the value of (5, 1) computed by thread 5. To ensure correctness, the work of both threads must be synchronized. If the threads are aligned to 2^n boundaries, such that they all fall within the same warp, the synchronization is implicit. In other words, if the overall number of threads needed is $(L_{ab} + 1)3N$, padding $(L_{ab} + 1)$ to a power of 2 will ensure that all threads with the same y and z indices are in the same warp at the negligible expense of some idle threads.

There are three ways the mappings can be aligned to a warp:

(1) The entire recurrence/transfer computation (if small enough) is mapped to a warp (or, a half-warp or a quarter-warp, etc). This holds if $(L_{ab} + 1)3N \leq$ warp.

(2) The xy dimension is aligned to a 2^n boundary. For example in the transfer figure above $L_{ab} = 5$, the xy boundary is therefore 16 threads since $L_{ab} \times 3 = 15$ and the next power of 2 is $2^n = 16$.

(3) The x dimension is aligned to a 2^n boundary. For example, if $L_{ab} = 7$, the x boundary is eight threads: since the next power of 2 is $2^n = 8$.

Option (1) is preferred. If the first condition fails, the choice between options 2 and 3 depends on which one minimizes the number of threads needed to perform recurrence/transfer computations. For example, if $L_{ab} = 4$, recurrence/transfer option 2 needs 16 threads, while option (3) requires 24 threads per root (since the number shown for (3) is per one Cartesian index, it must be multiplied by 3). If $L_{ab} = 7$, option (2) needs 32 threads, while option (3) needs 24 threads.

Once the intermediate 2D integrals are in shared memory, each thread computes a subset of integrals. The mapping between a thread/integral index and the corresponding 2D integral index is stored in the main memory and looked up for each element. The index is stored in a four-element vector, with the fourth index containing the coefficient index for hybrid SP functions.

Once all of the integrals are formed, they are transferred into the shared memory space previously used to store roots and intermediates. The exact number of integrals each thread computes depends on the size of the integral quartets and the number of threads launched. The number of threads depends mostly on the dimensions of the recurrence/transfer computations and the amount of shared memory used by the kernel. To accommodate those two requirements, a number of kernels are available with two, three, four, or eight multiples of a warp and the corresponding number of integrals per thread. During runtime, the kernel that maximizes the device occupancy is chosen.

For the case in which the entire recurrence/transfer computation can be mapped to a single warp, the integrals can be partitioned to warps rather than to an entire thread block, with each warp assigned to evaluate a unique contraction.

As implemented, the above approach is able to handle any quartet with a total angular momentum of 9 or less, for example, $(fd|dd)$, including shells with hybrid sp coefficients. The limit of 9 is imposed by the Rys roots program.

V.B. Low Angular Momentum Integrals. The most natural way to evaluate low angular momentum integrals is to assign individual quartets to a thread block and a single contraction to a thread, with each thread evaluating all integral elements corresponding to that contraction. However, this scheme becomes inefficient if the number of contractions is smaller than the number of threads in a block. This problem can be partially solved by assigning individual roots, rather than contractions, to a thread. For example, for a $(ps|ps)$ quartet, this effectively doubles the number of tasks to distribute since for each contraction there are two roots generated.

The low angular momentum kernels reuse the CPU kernel verbatim, with each device thread evaluating an individual root and all of the corresponding integrals, subsequently reduced into shared memory.

Once implemented, the above approach does not saturate the threads. The above implementation was therefore modified to handle an individual quartet per warp, in essence assigning two quartets per thread block. As an additional benefit, shell primitive loads decrease by half.

V.C. GPU Hartree–Fock Implementation. It is not possible to implement a parallel version of the Fock contraction within a thread block in which all six Fock contributions can be evaluated in the single inner loop. The approach taken here is to split the six updates onto separate loops, such that each Fock element can be computed independently. The implementation is as follows:

- One of the six integral/density loops is mapped to a warp. Hence, one thread block can contract and store concurrently one or more Fock tiles corresponding to the integral batch.
- The individual Fock matrix elements are mapped uniquely to a thread in a warp.
- The warp loads the density tile into shared memory.
- The density tile is contracted with the integral batch, and the Fock matrix element is stored in a register.
- The Fock matrix is locked with an exclusive read/write lock, and a Fock matrix element is added to the device memory
- The mutex is unlocked, and the warp proceeds to contract the next tile.
- Both the density and Fock tiles are stored in a block manner, such that all elements of a tile are continuous in memory.

```
lock(i,j) {
  while (atomicCAS(mutex(i,j),1)) {}
}
unlock(i,j) {
  mutex(i,j) = 0;
}

fock (i, j, k, l) {
  shared G; // integrals
  shared d(k,l); // density tile
  d(k,l) = D(k,l); // load density tile
  f = contract(g,d); // contract
  lock(i,j); // obtain lock
  F(i,j) += f; // add to main memory
  unlock(i,j); // release lock
}

if (do_ij) fock(i, j, k, l);
if (do_kl) fock(k, l, i, j);
if (do_ik) fock(i, k, k, l);
if (do_il) fock(i, l, k, l);
if (do_jk) fock(j, k, i, l);
if (do_jl) fock(j, l, i, k);
```

Listing 8. GPU HF kernel

Only one contraction out of six has a simple indexing; the other five contractions traverse the integrals with a non-contiguous stride, which must be accounted for.

Table 1. C++ Rys Method CPU Performance vs GAMESS[a]

system	GAMESS[b]	GAMESS/Rys[c]	C++[d]	improvement[e] (%)
cocaine 6-31G	21.3	52.4	37.2	−74.6/29.0%
cocaine 6-31G(d)	65.0	112.9	75.2	−15.7/33.4%
cocaine 6-31++G(d,p)	402.7	592.0	405.1	−0.60/31.6%
cocaine 6-311++G(2df,2p)	3424.4	3686.4	2356.3	31.2/36.1%
taxol 6-31G	310.2	691.6	474.1	−52.8/31.4%
taxol 6-31G(d)	1104.2	1729.2	1040.0	5.8/39.8%
taxol 6-31++G(d,p)	11225.9	15380.5	10288.0	8.4/33.1%
valinomycin 6-31G	853.6	1700.7	1104.4	−29.3/35.3%
valinomycin 6-31G(d)	2285.0	3445.7	2104.8	7.9/38.9%

[a]All times are in seconds on a single core [b]GAMESS using various ERI methods (default). [c]GAMESS using only Rys method. [d]Newly implemented C++ Rys method. [e]Improvement over default GAMESS/improvement over Rys-only GAMESS.

The current CUDA implementation does not provide a built-in device memory mutex; however, the mutex can be implemented with the atomic compare and swap operation, atomicCAS. The mutex implementation, summarized in Listing 8, will spin until a zero is read. Rather than locking the entire Fock matrix, only the individual tiles are locked at a time.

To achieve performance in the presence of the mutex, the quartets must be traversed so that the indices are not too similar; otherwise one would encounter mutex contention. For example, processing quartets (0,0,0,0), (1,0,0,0), ... would result in a high number of collisions as integral quartets are prescreened sequentially. This problem can be avoided by traversing the quartet list in non-one strides: for example, in strides of 32 in a round-robin manner, provided the quartet lists are on the order of thousands of entries. Since the basis set is sorted to begin with, the generated integral lists are typically well into the thousands.

V.D. Host/GPU Integration. The GPU device is driven by a separate host thread. First, the density matrix is copied into the device memory, and the Fock matrix is initialized to zeros. The GPU thread will then request a task from the task queue. If the quartet task can be evaluated by a device kernel, the quartets are prescreened on the host and asynchronously copied to the device, and the kernel is launched, asynchronously. This leaves the host thread to either prescreen the next batch or to evaluate those quartets that cannot be handled on the device. This approach allows for the overlap of the CPU/GPU execution. As will be shown in the performance section, the number of unhandled quartets is small, even with a high angular momentum basis set. Once the tasks are exhausted, the Fock matrix on the device is merged into the host.

VI. PERFORMANCE

The newly implemented HF algorithm was compared against the standard GAMESS[23] code, using the Rys Quadrature method only, as well as the default GAMESS option, which chooses the optimal integral package according to the integral types.[24]

The GAMESS code was compiled with the following command:

```
gfortran -O3 -msse3
```

The new implementation was compiled with the following:

```
g++ -O3 -msse3
```

The gcc version was 4.4.3 for both gfortran and g++. The benchmarks were executed on two Intel Xeon E5405 2.00 GHz CPUs.

The timing comparisons of the new C++ CPU code with the GAMESS code are listed in Table 1. All of the timings are given in seconds, with C++ and GAMESS runs set to utilize a single core. The following should be kept in mind when interpreting the results:

- The rotated axis algorithm and its variations are algorithmically much less complex than the Rys Quadrature algorithm for contracted shells, like those typically found in low angular momentum basis sets, so GAMESS calculations that use only the Rys algorithm (for comparison purposes) will naturally take longer than the GAMESS default (optimal) option noted above.
- The rotated axis code[24] in GAMESS has been reimplemented to take some advantage of modern processors.
- The Rys Quadrature algorithm is advantageous for small contraction/high angular momentum basis sets. The implementation of the Rys Quadrature algorithm in GAMESS is the original implementation from the HONDO[25] package and does not take into account modern processor architecture.
- For large basis sets with *f* functions, the relative number of shell quartets handled by the Rys Quadrature algorithm is significantly higher than for smaller basis sets.

The test computations were performed on the molecules cocaine, taxol, and valinomycin using basis sets that incorporate a different number of s, p, sp, d, and f shells. Cocaine is the smallest of the three molecules, and valinomycin is the largest. The improvement over the original Rys Quadrature is on the order of 30−40% for all cases. When compared to the default integral option in GAMESS, which picks the Rys Quadrature only if f and higher angular momentum functions are present, the performance is either higher, lower, or the same, depending on the number of d functions, the size of the basis set, and correspondingly the memory requirement of the density and Fock matrices.

The rewritten Rys Quadrature algorithm is still much slower than the rotated-shell axis code when only s and p functions are involved. The difference is most pronounced when the total basis set is small. The difference diminishes with increasing Fock and density matrix sizes as memory locality becomes more important. For example, for the cocaine 6-31G computation, the rotated shell axis code is 75% faster, but only 30% faster with the much larger valinomycin 6-31G computation.

When d functions are present, the C++ Rys Quadrature code performs better than the current packages as the basis set size increases. For taxol and valinomycin, the new CPU approach outperforms the current GAMESS codes by a few percent. The new code clearly becomes faster if *f* functions are present. In the best case scenario, it is 31% faster than the GAMESS integral packages, due to both better memory locality and the higher fraction of quartets with higher angular momentum. Overall, the new Hartree–Fock implementation is scalable and efficient, improving the overall performance by as much as 30%.

The comparison between the C++ CPU and GPU codes is summarized in Tables 2, 3, and 4, broken down by the relative time a particular shell quartet takes. A quartet size is the product of the shell sizes in a quartet. For example, (*ps*|*ss*) quartets are of size 3 ($3 \times 1 \times 1 \times 1$) and (*dd*|*dd*) quartets are of size 1296 ($6 \times 6 \times 6 \times 6$). The benchmark molecule is taxol, and the three basis sets are cc-pVDZ, cc-pVTZ, and 6-31G(d).[26] The correlation consistent basis sets have contraction orders as high as 4096, while the Pople basis sets rely heavily on hybrid sp shells. Note that a large fraction of integral time is spent computing the multitude of integrals with p shells. In fact, for the cc-pVDZ basis set, 60% of the total time is spent evaluating the smallest (in terms of quartet size) four integrals.

Table 2. Taxol/cc-pVDZ GPU Performance

quartet size[a]	CPU % by time[b]	GPU speed-up (*x*)[c]
1	14.2	35.2
3	22.8	23.0
6	6.6	18.5
9	19.3	17.4
18	9.6	14.5
27	7.0	9.6
36	1.6	11.4
54	8.7	12.6
81	1.9	12.8
108	3.3	17.2
162	2.5	16.0
216	0.4	14.3
324	1.6	16.7
648	0.4	17.9
1296	0.1	15.0
overall[d]	5068.66 s	17.5

[a]Product of four shell sizes, e.g., s = 1, p = 3, sp = 4, d = 6. [b]Fraction of total time computing quartet of this size. [c]GPU speed-up (relative to C++ CPU) for quartets of this size. [d]Total time and total speed-up.

The GPU speed-ups over the single CPU core times (Tables 2–4) vary from 17.5× to 12× for the cc-pVTZ basis set. The specialized low-angular momentum quartet kernels perform fairly well, with the lowest speed-up for the last specialized kernel with two sp shells, size 16. The speed-up consequently drops for the general kernel. The performance improves as the quartet gets bigger. The number of slower kernels in the shell size 16–100 range is rather high, and it tends to lower the overall speed-up.

CPU and GPU execution can run together to occupy all available resources on the nodes.

Table 5 shows the wall clock time required to perform a single SCF iteration of fairly large computations. To showcase various points of performance and comparability, the times are given for combinations of serial and parallel execution with or without GPU.

Table 3. Taxol/cc-pVTZ GPU Performance

quartet size[a]	CPU % by time[b]	GPU speed-up (*x*)[c]
1	4.3	25.9
3	8.5	17.5
6	4.6	15.1
9	8.4	13.8
10	1.7	14.6
18	8.0	11.6
27	3.7	8.2
30	3.7	9.3
36	2.5	10.4
54	8.3	11.4
60	2.5	13.3
81	1.1	11.4
90	4.1	15.1
100	0.8	15.5
108	5.8	15.9
162	2.9	15.0
180	5.2	14.0
216	1.2	14.1
270	1.7	17.3
300	1.4	15.8
324	3.5	17.3
360	1.7	15.4
540	3.6	18.9
600	1.1	15.7
648	1.6	17.9
900	1.1	18.7
1000	0.2	15.0
1080	2.9	15.3
1296	0.4	15.8
1800	1.6	19.4
2160	0.7	20.1
3000	0.3	n/a
3600	0.7	n/a
6000	0.3	n/a
10000	0.0	n/a
overall[d]	35110.4 s	12.0

[a]Product of four shell sizes, e.g., s = 1, p = 3, sp = 4, d = 6. [b]Fraction of total time computing quartet of this size. [c]GPU speed-up (relative to C++ CPU) for quartets of this size. [d]Total time and total speed-up.

As can be seen, the multithreaded implementation is efficient, consistently achieving over 95% parallel efficiency even for the small computations. Although not shown, the implementation scales well beyond eight threads. In case of the largest valinomycin benchmark, combining CPU and GPU execution brought a calculation that took more than 2 h to just over 5 min.

VII. CONCLUSIONS

The newly implemented Rys Quadrature and Fock Matrix algorithms, implemented as a stand-alone C++ library, with C and Fortran bindings, provides on the order of a 40% improvement over the traditional Fortran Rys Quadrature and performance that is similar to that of less computationally intensive algorithms. The library is fully multithreaded and has favorable scaling across eight cores or more cores within a single node. The library has a simple interface to evaluate a block of integrals as well several compile time parameters to optimize performance. Although algorithmically much more expensive, the new Rys quadrature implementation uses a

Table 4. Taxol/6-31G(d) GPU Performance

quartet size[a]	CPU % by time[b]	GPU speed-up (x)[c]
1	1.7	28.5
4	6.5	20.9
6	1.9	18.8
16	12.3	13.1
24	6.6	10.6
36	1.1	11.7
64	13.9	13.7
96	16.8	15.8
144	5.9	19.5
216	0.6	15.4
256	12.4	23.5
384	11.5	20.9
576	7.0	20.2
864	1.7	21.3
1296	0.2	16.6
overall[d]	1031.94 s	16.6

[a]Product of four shell sizes, e.g., s = 1, p = 3, sp = 4, d = 6. [b]Fraction of total time computing quartet of this size. [c]GPU speed-up (relative to C++ CPU) for quartets of this size. [d]Total time and total speed-up.

Table 5. Combined CPU/GPU Performance[a]

system	one core	eight cores	one GPU	eight cores + one GPU
taxol 6-31G	474.1	60.2	37.4	26.5
taxol 6-31G(d)	1040.0	132.2	80.2	53.0
taxol 6-31G(2d,2p)	3429.8	442.3	290.0	178.1
taxol 6-31++G(d,p)	10288.0	1243.9	984.5	539.9
valinomycin 6-31G	1104.4	143.9	92.4	60.0
valinomycin 6-31G(d)	2104.8	270.7	189.6	116.9
valinomycin 6-31G(2d,2p)	7439.3	964.0	554.0	328.0

[a]All times are in seconds. The times include all steps to evaluate a single iteration energy, including diagonalization.

processor very effectively and is able to match and beat the performance of recently implemented algorithms, such as those found in GAMESS,[24] which have much less algorithmic complexity for small angular momentum integrals.

The GPU version, adopted from the CPU version, shows speed-ups as high as 17.5×. Importantly, this speedup is relative to the newly optimized C++ CPU code, not to the original legacy Fortran code. The Rys Quadrature however does not scale well in the midsize shell quartets. Part of a rotated-shell axis code is likely to increase the overall performance to 20× or higher.

■ AUTHOR INFORMATION

Corresponding Author

*E-mail: mark@si.msg.chem.iastate.edu.

Notes

The authors declare no competing financial interest.

■ REFERENCES

(1) Gerber, R. *The Software Optimization Cookbook: High-Performance Recipes for IA-32 Platforms*; Intel Press: Hillsboro, OR, 2006; pp 191−207.
(2) Gerber, R. *Programming with Hyper-Threading Technology How to Write Multithreaded Software for Intel IA-32 Processors*; Intel Press: Hillsboro, OR, 2004; pp 9−29.
(3) Turney, J. M.; Simmonett, A. C.; Parrish, R. M.; Hohenstein, E. G.; Evangelista, F. A.; Fermann, J. T.; Mintz, B. J.; Burns, L. A.; Wilke, J. J.; Abrams, M. L.; Russ, N. J.; Leininger, M. L.; Janssen, C. L.; Seidl, E. T.; Allen, W. D.; Schaefer, H. F.; King, R. A.; Valeev, E. F.; Sherrill, C. D.; Crawford, T. D. Psi4: An open source ab initio electronic structure program. *Comput. Mol. Sci.* **2011**, *2*, 556−565.
(4) Janssen, C. L.; Nielsen, I. B.; Leininger, M. L.; Valeev, E. F.; Kenny, J. P.; Seidl, E. T. *The Massively Parallel Quantum Chemistry Program (MPQC)*; Sandia National Laboratories: Livermore, CA, 2008.
(5) (a) Rys, J.; Dupuis, M.; King, H. F. Computation of electron repulsion integrals using the Rys quadrature method. *J. Comput. Chem.* **1983**, *4*, 154−157. (b) Asadchev, A.; Allada, V.; Felder, J.; Bode, B. M.; Windus, T. L.; Gordon, M. S. Uncontracted Rys Quadrature Implementation of up to g Functions on Graphical Processing Units. *J. Chem. Theory Comput.* **2010**, *6*, 696−716.
(6) Valeev, E. F.; Fermann, J. T. *Libint*. http://sourceforge.net/p/libint/ (accessed Aug. 6, 2012).
(7) *Cheetah - the Python-Powered template engine*. http://www.cheetahtemplate.org/ (accessed Aug. 6, 2012).
(8) *SymPy: Python Library for Symbolic Mathematics*; SymPy Development Team, 2009. http://www.sympy.org (accessed Sept. 2012).
(9) Stein, W. *Sage: Open Source Mathematical Software*, version 2.10.2; The Sage Group, 2008. http://www.sagemath.org (accessed Aug 6, 2012).
(10) Wolfram, S. *The Mathematica(R)book*, 5th ed.; Wolfram Media Inc.: Champaign, IL, 2003; pp 79−101.
(11) Furlani, T. R.; King, H. F. Implementation of a parallel direct SCF algorithm on distributed memory computers. *J. Comput. Chem.* **1995**, *16*, 91−104.
(12) Buttari, A.; Langou, J.; Kurzak, J.; Dongarra, J. Parallel tiled QR factorization for multicore architectures. *Proceedings of the 7th International Conference on Parallel Processing and Applied Mathematics*, PPAM'07
(13) *Boost C++ libraries*. http://www.boost.org/ (accessed Aug. 6, 2012).
(14) Abrahams, D. *C++ template metaprogramming: concepts, tools, and techniques from boost and beyond*; Addison-Wesley: Boston, 2005, pp 1−33.
(15) *boost::enable_if*. http://www.boost.org/doc/libs/release/libs/utility/enable_if.html (accessed Aug 6, 2012).
(16) *The Boost MPL library*. http://www.boost.org/doc/libs/release/libs/mpl/doc/index.html (accessed Aug 6, 2012).
(17) *The Boost Preprocessor*. http://www.boost.org/doc/libs/release/libs/preprocessor/doc/index.html (accessed Aug 6, 2012).
(18) *Boost thread*. http://www.boost.org/doc/libs/release/doc/html/thread.html (accessed Aug 6, 2012).
(19) Ufimtsev, I. S.; Martinez, T. J. Quantum chemistry on graphical processing units. 1. Strategies for two-electron integral evaluation. *J. Chem. Theory Comput.* **2008**, *4*, 222−231.
(20) Ufimtsev, I. S.; Martinez, T. J. Quantum chemistry on graphical processing units. 2. direct self-consistent-field implementation. *J. Chem. Theory Comput.* **2009**, *5*, 1004−1015.
(21) Yasuda, K. Two-electron integral evaluation on the graphics processor unit. *J. Comput. Chem.* **2008**, *29*, 334−342.
(22) Wilkinson, K. A.; Sherwood, P.; Guest, M. F.; Naidoo, K. J. Acceleration of the GAMESS-UK electronic structure package on graphical processing units. *J. Comput. Chem.* **2011**, *32*, 2313−2318.
(23) 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; pp 1167−1189.
(24) Ishimura, K.; Nagase, S. A new algorithm of two-electron repulsion integral calculations: a combination of Pople-Hehre and McMurchie-Davidson methods. *Theor. Chem. Acc.* **2008**, *120*, 185−189.
(25) Dupuis, M.; Rys, J.; King, H. F. HONDO. Quantum Chemistry Program Exchange, 11, 336338, 1977.

(26) Davidson, E. R.; Feller, D. Basis set selection for molecular calculations. *Chem. Rev.* **1986**, *86*, 681−696.