JCTC Journal of Chemical Theory and Computation

Uncontracted Rys Quadrature Implementation of up to G Functions on Graphical Processing Units

Andrey Asadchev,[†] Veerendra Allada,[‡] Jacob Felder,[†] Brett M. Bode,[‡] Mark S. Gordon,[*,†] and Theresa L. Windus[†]

Department of Chemistry and Department of Electrical and Computer Engineering Iowa State University and Ames Laboratory, Ames, Iowa 50011

Received September 25, 2009

Abstract: An implementation is presented of an uncontracted Rys quadrature algorithm for electron repulsion integrals, including up to **g** functions on graphical processing units (GPUs). The general GPU programming model, the challenges associated with implementing the Rys quadrature on these highly parallel emerging architectures, and a new approach to implementing the quadrature are outlined. The performance of the implementation is evaluated for single and double precision on two different types of GPU devices. The performance obtained is on par with the matrix−vector routine from the CUDA basic linear algebra subroutines (CUBLAS) library.

1. Introduction

The evaluation of two-electron ($2e^-$) repulsion integrals (ERI) is a major computational step in determining the electronic structure of molecules using ab initio quantum chemistry and density functional theory (DFT) methods.[1] Accelerating the integral calculations significantly reduces the overall runtime of the direct Hartree−Fock (HF)[2] and the post-HF methods, e.g., many body perturbation methods.[3]

In 1951, Boys[4] proposed using Gaussian functions as a standard atomic basis set for quantum chemistry computations because the integrals over the Gaussian functions can be evaluated efficiently in closed form. Since then, many different algorithms have been developed to evaluate ERIs over Gaussian functions. The Gauss quadrature method using orthogonal Rys polynomials, developed by Dupuis, Rys, and King (DRK),[5] is a general algorithm that is applicable to a wide range of integrals that arise in computational chemistry. Besides the original Rys quadrature, other ERI algorithms have been developed by, for example, Pople and Hehre (PH),[6] McMurchie and Davidson (MD),[7] Obara and Saika (OS),[8] and Head-Gordon and Pople (HGP).[9] Some of the modifications to the original DRK algorithm are due to Lindh, Ryu, and Liu (LRL)[10] and Dupuis and Marquez (DM).[11] Each of the developed schemes is more efficient for particular cases of integrals, while less efficient or inapplicable for other cases. In practice, quantum chemistry codes, such as the general atomic and molecular electronic structure system (GAMESS),[12] include several different ERI methods in order to take optimal advantage of the best method for particular integral and angular momentum types. The focus of the present work is on ERIs over higher (e.g., d, f) angular momentum functions.

Computationally the ERI calculations scale as $\sim M^3$ to $\sim M^4$, where M is the number of basis functions used in the calculation, and the scalability range depends on the amount of integral prescreening that can be applied. For the most common calculations, M is typically less than a thousand, while larger calculations could require thousands of basis functions.

The ERI calculations are specific to the domain of computational chemistry and the related fields. They are much less common than general methods, such as Fourier transforms and linear algebra kernels, and typically are not as optimized as the basic linear algebra subroutines (BLAS) libraries. A typical HF or DFT calculation requires both ERI and linear algebra computations. However, ERI computations tend to dominate the overall time, since they require more floating point operations (flops). Moreover, unlike numerical linear algebra kernels that exhibit well-defined memory access patterns and simple long loop structures, ERI calculations have to account for many types of integral classes, and therefore, iteration variables do not have a simple linear

[*] Corresponding author. E-mail: mark@si.msg.chem.iastate.edu.
[†] Department of Chemistry.
[‡] Department of Electrical and Computer Engineering.

relationship to the data elements which must be accessed. To help speed up the time needed to complete the ERI calculations, a general graphical processing unit (GPU) programming model has been implemented. The goal is to implement high angular momentum uncontracted ERIs in this scheme.

2. Electron Repulsion Integrals

Gaussian functions are taken as the standard basis for most ab initio methods. The Cartesian form of a primitive uncontracted one-electron Gaussian basis function with the center located at the origin takes the form of eq 1:

$$\phi(r) = x^{a_x} y^{a_y} z^{a_z} \exp(-\alpha r^2) \qquad (1)$$

where α is the Gaussian exponent that governs the spatial extent of the function, r measures the distance from the atomic origin, and a_x, a_y, and a_z are local quantum numbers that determine the net angular momentum L_a by eq 2:

$$L_a = a_x + a_y + a_z \qquad (2)$$

Individual Gaussian functions, like those described in eq 1, are generally called “primitive” functions. Especially for lower angular momentum functions (e.g., s and p functions), the actual basis functions are taken to be linear combinations (“contractions”) of primitive Gaussians:

$$\phi_a(r) = \sum_k^K D_{ka}\phi_k(r) \qquad (3)$$

On the other hand, functions with higher angular momentum (e.g., d, f, and g functions) are typically uncontracted—the focus of this work.

An uncontracted ERI in terms of these one-electron functions can be expressed as:

$$(ab|cd) = \int\int \phi_a(1)\phi_b(1)\frac{1}{r_{12}}\phi_c(2)\phi_d(2)dr_1dr_2 \qquad (4)$$

A contracted ERI can be constructed from a series of uncontracted ERIs (eq 5):

$$(ij|kl) = \sum_a^K \sum_b^L \sum_c^M \sum_d^N D_{ai} D_{bj} D_{ck} D_{dl} (ab|cd) \qquad (5)$$

The angular momentum L_a specifies the shape of the function and is denoted by the letters s, p, d, f, etc., for angular momentum values of 0, 1, 2, 3, etc., respectively. Functions with the same angular momentum that differ only in a_x, a_y, and a_z indices belong to the same shell. Grouping functions into shells allows the ERIs to be evaluated more efficiently. The size (i.e., the number of functions) of a shell with angular momentum L_a is

$$\binom{L_a + 2}{2}$$

and the size of an ERI shell block is

$$\binom{L_a + 2}{2}\binom{L_b + 2}{2}\binom{L_c + 2}{2}\binom{L_d + 2}{2}$$

where

$$\binom{n}{2}$$

is the binomial coefficient evaluated as $n(n - 1)/2$, and L_a, L_b, L_c, and L_d are the angular momenta of the four atomic orbitals in the ERI.

Note that the individual ERIs have an eight-fold symmetry, since $(ij|kl) = (ji|kl) = (ij|lk)$, etc.; however, the ERIs are computed as shell blocks, rather than individual integrals. So generally, the eight-fold symmetry is only relevant between blocks, not within a block.

2.1. Rys Quadrature. The Rys quadrature proposed by DRK is efficient for higher order integrals (integrals with a higher order angular momentum) that are required for very accurate calculations that include electron correlation. However, it is less efficient for lower order highly contracted integrals. An attractive feature of the Rys quadrature is that it is very stable numerically, an important advantage for higher order integrals. Unlike other methods mentioned in the Introduction Section, it has a very low memory footprint, making it amenable for architectures with smaller caches, such as the GPUs of interest in this work.

The basic idea of the Rys quadrature is to evaluate the integral using a numerical Gaussian quadrature based on a set of orthogonal Rys polynomials. Equation 4 can be expressed, using i, j, k, l to denote functions of a primitive uncontracted ERI shell block $(ab|cd)$, in the form

$$(ij|kl) = \sum_{m=0}^L C_m F_m(X) \qquad (6)$$

$$F_m(X) = \int_0^1 t^{2m}\exp(-Xt^2)\mathrm{d}t \qquad (7)$$

$$\begin{aligned} X &= \rho(r_A - r_B)^2 \\ r_A &= (\alpha_i r_i + \alpha_j r_j)/A \\ r_B &= (\alpha_k r_k + \alpha_l r_l)/B \end{aligned} \qquad (8)$$

$$\begin{aligned} \rho &= AB/(A + B) \\ A &= \alpha_i + \alpha_j \\ B &= \alpha_k + \alpha_l \end{aligned} \qquad (9)$$

$$L = L_a + L_b + L_c + L_d \qquad (10)$$

As suggested in eq 8, X depends on the Gaussian exponents and centers. Equation 6 can be written as eq 11:

$$(ij|kl) = \int_0^1 \exp(-Xt^2)P_L(t)\mathrm{d}t \qquad (11)$$

where P_L is a polynomial of degree L, eq 10, with the coefficients C_m in eq 6. Equation 11 can be evaluated exactly by an N-point (where N is an integer greater than $L/2$) Gaussian quadrature:

$$(ij|kl) = \sum_{\omega=1}^N W_\omega P_L(t_\omega) \qquad (12)$$

$$N = L/2 + 1 \tag{13}$$

The W_α and t_α are weights and roots of the Rys polynomial, respectively. For example, a $(dd|dd)$ block will have $L = 2 + 2 + 2 + 2 = 8$ and $N = 8/2 + 1 = 5$, and a $(gg|ff)$ block will have $L = 4 + 4 + 3 + 3 = 14$ and $N = 14/2 + 1 = 8$. Separation of variables allows the terms of the P_L polynomial, which are integrals over dr_1dr_2 (see eq 4) to be written as a product of three two-dimensional (2-D) integrals I_x, I_y, and I_z over dx_1dx_2, dy_1dy_2, and dz_1dz_2, respectively.

$$P_L(t) = 2(\rho/\pi)^{1/2} I_x I_y I_z \tag{14}$$

The overall ERI formula becomes

$$(ij|kl) = 2(\rho/\pi)^{1/2} \sum_\omega I_x(t_\omega) I_y(t_\omega) I_z(t_\omega) W_\omega \tag{15}$$

The 2-D integrals of a shell block have array dimensions (in FORTRAN/MATLAB notation, where commas delimit dimensions and colons specify the range) as shown below:

$$I_{q(=x,y,z)}(N, 0{:}L_a, 0{:}L_b, 0{:}L_c, 0{:}L_d) \tag{16}$$

The first or leading dimension in eq 16 corresponds to the number of roots, and the last four dimensions correspond to Cartesian exponents for each function in a shell block. When constructing the ERI block, the 2-D integrals will be reused multiple times, hence, the computational and memory advantage of calculating ERIs as a block. For example, to construct the first six integrals of the $(pp|pp)$ shell block, the following 2-D Cartesian integrals are used (multiplication by a constant factor is implied):

$$\begin{aligned}
(p_xp_x|p_xp_x) &= \sum_\omega I_x(\omega,1,1,1,1) I_y(\omega,0,0,0,0) I_z(\omega,0,0,0,0)\\
(p_yp_x|p_xp_x) &= \sum_\omega I_x(\omega,0,1,1,1) I_y(\omega,1,0,0,0) I_z(\omega,0,0,0,0)\\
(p_zp_x|p_xp_x) &= \sum_\omega I_x(\omega,0,1,1,1) I_y(\omega,0,0,0,0) I_z(\omega,1,0,0,0)\\
(p_xp_y|p_xp_x) &= \sum_\omega I_x(\omega,1,0,1,1) I_y(\omega,0,1,0,0) I_z(\omega,0,0,0,0)\\
(p_xp_y|p_xp_x) &= \sum_\omega I_x(\omega,0,0,1,1) I_y(\omega,1,1,0,0) I_z(\omega,0,0,0,0)\\
(p_yp_y|p_xp_x) &= \sum_\omega I_x(\omega,0,0,1,1) I_y(\omega,0,1,0,0) I_z(\omega,1,0,0,0)
\end{aligned}$$

The roots and weights of the Rys polynomials can be evaluated by polynomial approximations[13] or by using a general Stieltjes procedure.[14] The 2-D Cartesian integrals are evaluated efficiently using recurrence and transfer relationships. The recurrence relationships generate 2-D integrals with all angular momenta shifted to centers i and k from $(ss|ss)$ 2-D integrals, and transfer relationships shift the angular momentum to centers j and l to generate the desired 2-D integrals. For the details of these relationships, the reader is referred to the original DRK paper.[5]

The quadrature step itself, i.e., the summation over the roots, eq 15, is the time-consuming step of the ERI shell calculation, requiring:

$$3N\binom{L_a+2}{2}\binom{L_b+2}{2}\binom{L_c+2}{2}\binom{L_d+2}{2}$$

flops, where the factor of 3 is from the two multiplications and an addition in each step. The transfer relationships scale as $N(L_a + 1)(L_b + 1)(L_c + 1)(L_d + 1)$, requiring many fewer operations than the quadrature step as the angular momentum increases. The recurrence relationships and root evaluation require even fewer flops than the transfer relationships for higher order integrals. Therefore, since an efficient implementation of the quadrature step determines the overall performance of the algorithm, the main topic of this paper is the efficient parallel implementation of the quadrature. Unlike the recurrence and transfer equations, which have predictable memory access patterns and can be expressed as simple vector operations, the quadrature step has complex memory access patterns which span a large data set and depend on the particular ERI class being evaluated. For example, the evaluation of the $(ff|ff)$ ERI block requires $3N(L_a + 1)(L_b + 1)(L_c + 1)(L_d + 1) = (3)(7)(4)(4)(4)(4) = 5376$ floating point (FP) numbers for three 2-D integral arrays (X, Y, Z) and $10^4 = 10000$ FP numbers for the final integral. For double precision numbers, the overall memory would be 123008 Bytes, well beyond the size of a typical L1 data cache.

Algorithm 1 outlines the basic structure of the Rys quadrature. Its simplicity obscures the fact that the Cartesian indices do not have a simple relationship to the iteration variables and must be either tabulated or each case must be programmed specifically for a particular ERI class.

Algorithm 1: Rys Quadrature

```
Algorithm 1 Rys Quadrature
for all l do
  for all k do
    for all j do
      for all i do
        $I(i,j,k,l) = \sum_\omega I_x(\omega,i_x,j_x,k_x,l_x) I_y(\omega,i_y,j_y,k_y,l_y) I_z(\omega,i_z,j_z,k_z,l_z)$
      end for
    end for
  end for
end for
```

3. Graphical Processing Units

GPU technology has emerged as a viable computing platform for general purpose application programming, also known as general purpose computation on graphical processing units (GPGPU). The GPUs offer high-density arithmetic units at the expense of larger cache sizes and control units. In terms of linear algebra kernels, the GPUs can approach 20 and 70 giga floating point operations per second (GFLOPS) for matrix−vector and matrix−matrix routines, respectively, on current double precision (DP) capable devices[15] that have a theoretical peak of around 90 GFLOPS.

3.1. Compute Unified Device Architecture. Among the current GPGPU technologies, the NVIDIA compute unified device architecture (CUDA)[16] language environment is available for several GPU devices and is the target implementation choice. CUDA is a unified hardware computing architecture and programming model for graphics as well

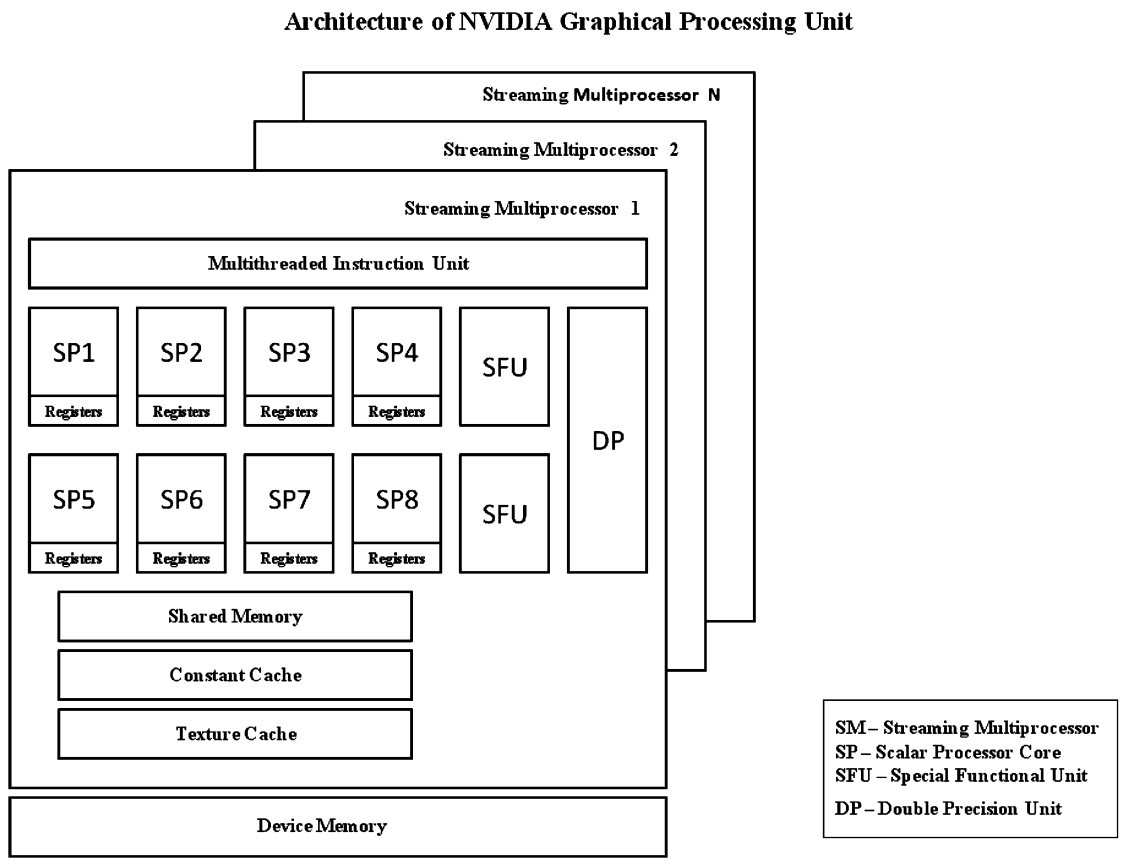

Architecture of NVIDIA Graphical Processing Unit
Streaming Multiprocessor N
Streaming Multiprocessor 2
Streaming Multiprocessor 1
Multithreaded Instruction Unit
SP1
SP2
SP3
SP4
SFU
DP
SP5
SP6
SP7
SP8
SFU
Registers
Shared Memory
Constant Cache
Texture Cache
Device Memory
SM – Streaming Multiprocessor
SP – Scalar Processor Core
SFU – Special Functional Unit
DP – Double Precision Unit

Figure 1. High-level architecture of a GPU.

as general-purpose processors. The current CUDA device architecture consists of a scalable array of streaming multiprocessors (SM), see Figure 1. Each SM consists of eight scalar processors (SP), a multithreaded instruction unit, on-chip shared memory, one double precision unit, and two special purpose transcendental functional units. Under the CUDA programming model, the GPU is viewed as a highly multithreaded compute device capable of executing many threads in parallel. The threads execute a sequence of instructions in a data parallel fashion—single-instruction multiple threads (SIMT).

Computationally demanding code paths of an application are isolated into functions (*kernels* in NVIDIA terminology) that are compiled into the instruction set architecture of the GPU device. The CUDA programming interface is designed with a minimal set of extensions to the C/C++ language. A runtime library provides functions to manage the compute device, to perform memory operations, and to run the device-specific functions. The main goal of the programming environment is to develop scalable and efficient parallel programs.

A computational kernel is launched from the host and executed by T threads (T is application specific) on the device. The threads are hierarchically arranged as a grid of blocks and as a block of threads, as shown in Figure 2 (adopted from the programmers manual).[17] Each thread within a thread block has a unique set of (x, y, and z) indices that allow three-dimensional (3-D) data to be mapped onto

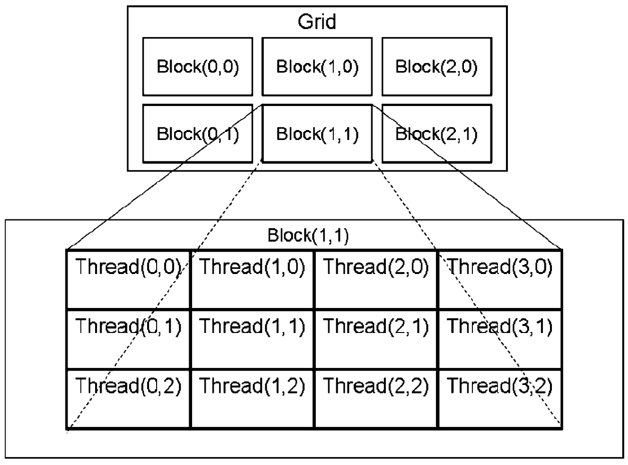


Figure 2. Grid of blocks and block of threads (z-dimension is implied).

a thread block. Each thread block has unique x and y coordinates, which map all the thread blocks onto a 2-D grid of blocks.

The logical memory space seen by the threads can be hierarchically arranged based on the data visibility (see Figure 1). Each thread has access to its local registers on the processor. Threads in a block can access and share data via the parallel data cache, called shared memory. The registers and the shared memory have a low latency and are limited resources available to the threads. One of the biggest challenges in designing the kernels lies in optimizing the per-thread register and the shared memory usage. Each thread also has access to a private local memory and a global memory space that are both part of the device memory and have high data access latencies. Apart from these, an application can also use the read-only constant and the texture memories that are cached.

Access to the main memory has a high latency, on the order of hundreds of cycles. To achieve full bandwidth, accesses to the main memory must be coalesced, meaning consecutive threads access consecutive memory elements to achieve full memory bandwidth. Coalescing ensures that multiple memory requests are being served simultaneously rather than served sequentially. Although only one thread block can execute at any given time on a SM, multiple thread blocks can be *active*, thereby hiding the memory latency by overlapping the computations and the communications. An active block in CUDA terminology is a block that is ready for execution whenever a SM becomes free, e.g., when the current executing thread block starts fetching the memory. The number of active thread blocks is limited by the register and the shared memory usage.

The execution of a thread block is further batched into a series of *warps* that are consecutively arranged based on the thread number in batches of 32. To maximize the parallel performance, all threads in a warp must execute an identical GPU instruction, otherwise the warp is said to diverge, and the differing instructions are executed sequentially.

4. CUDA Rys Quadrature Implementation

4.1. Related Work. Ufimtsev and Martinez[18] evaluated the ERIs of *s* and *p* functions on GPUs in single precision using the MD algorithm. They later developed an entire Hartree–Fock code that runs on a GPU and showed improved performance[19] over the CPU code. Yasuda[20] implemented the Rys quadrature on a GPU in enhanced single precision for *s* and *p* integrals (i.e., some double precision computations were emulated in the software but still using single precision hardware). A new interpolation formula was proposed for the roots and the weights, and an error analysis for the quadrature was given. Some work has also been done to implement ERI algorithms on IBM CELL and FPGAs,[21] however only for the rather limited case of $(ss|ss)$ ERIs.

To our knowledge there has not been a reported implementation of an ERI algorithm for *d* or higher angular momentum functions on GPUs or on other accelerators. The main difficulty seems to have been the limited amount of fast memory and the amount of code that must be generated for many cases involving higher angular momentum functions. This is the focus of the present work.

4.2. Implementation Considerations. Since the ERI computations are memory bound, the main consideration in designing the CUDA Rys quadrature is to optimize the memory access patterns and the data reuse. The 2-D integrals are reused multiple times to construct different ERIs and should, therefore, be loaded into shared memory. This also implies that an ERI block should be mapped onto a single thread block, as shared memory access and synchronization is limited to thread block boundaries. The ERI blocks are mapped onto the grid so that each thread block computes one ERI block. For the purposes of discussion, block is used to refer to both thread and ERI blocks.

Device memory loads and stores should be coalesced to parallelize memory accesses with high latencies. Multiple thread blocks should be active on a single SM in order to hide memory latency by overlapping computation and communication. In order to have multiple active thread blocks, the shared memory and the registers should be used sparingly. To illustrate hardware constraints, if a GPU has only 1638 4-byte registers and 16 KB of shared memory, then a kernel using 32 registers per thread and 2688 bytes of shared memory per thread block has a limit of 512 threads imposed by the register use and 6 active thread blocks per SM due to shared memory availability.

Clearly, for the larger ERI classes, the entire set of 2-D integrals cannot be kept in shared memory all at once but must be loaded from the device memory as needed. The iteration through the ERIs should be done so as to minimize the number of device memory loads. If the ERIs are only computed on the GPU but are not contracted right away, e.g., to form the Fock operator, then there is no reuse of the final ERIs.

4.3. Implementation Design. The current CUDA capable hardware imposes a limit of 512 or 768 maximum threads per block, depending on the particular GPU device. Consider the $(dd|dd)$ ERI block case. The size of the entire block is $6^4 = 1296$ elements, exceeding the maximum number of threads. However, it is possible to map multiple elements to a single thread, e.g., by mapping *i*, *j*, and *k* indices, corresponding to the first three shells of the block, to a unique thread and iterating over the last index *l*. Since the thread block is 3-D, the mapping of the *i*, *j*, and *k* shell index to a thread is natural. Algorithm 2 outlines the general idea. The algorithm is in Python-like pseudo code, with ## signifying comments, and the indices and loops over the roots, *N*, are implied.

Algorithm 2: CUDA Rys Quadrature, *i*, *j*, and *k* Mapping

Algorithm 2 CUDA Rys quadrature, *i*, *j*, *k* mapping

```
## map threads to ERI elements threadIdx is the thread coordinate
$i = threadIdx.x$
$j = threadIdx.y$
$k = threadIdx.z$
## The arrays LX, LY, LZ map functions to exponents
$(ix, iy, iz) = (LX[i], LY[i], LZ[i])$
$(jx, jy, jz) = (LX[j], LY[j], LZ[j])$
$(kx, ky, kz) = (LX[k], LY[k], LZ[k])$
for all $l$ do
  sync threads
  ## load 2-D integrals to shmem
  if $LX[l] \neq LX[l-1]$ then
    $I_{x,shmem} = I_x(:,:,:,LX[l])$
  end if
  if $LY[l] \neq LY[l-1]$ then
    $I_{y,shmem} = I_y(:,:,:,LY[l])$
  end if
  if $LZ[l] \neq LZ[l-1]$ then
    $I_{z,shmem} = I_z(:,:,:,LZ[l])$
  end if
  sync threads
  $I(i,j,k,l) = \sum_N I_{x,shmem}(ix,jx,kx) I_{y,shmem}(iy,jy,ky) I_{z,shmem}(iz,jz,kz)$
end for
```

In terms of shared memory, i, j, and k mapping requires all the 2-D integrals of a specific l index. For the $(dd|dd)$ ERI case, this means that the shared memory overhead for each l iteration is $N(L_a + 1)(L_b + 1)(L_c + 1) = 5(3^3) = 135$ elements per 2-D integral block. Though three 2-D integral blocks are needed per iteration, it is most likely that one of the previous Cartesian indices will stay the same. This means that the corresponding 2-D integral is already in the shared memory, reducing the memory communication by a third. For example, the construction of a d shell is outlined below. The three rows correspond to I_x, I_y, and I_z Cartesian indices. The indices marked with an asterisk represent load operations. Though there are a total of 18 indices, only 13 indices must be loaded if the shell is arranged to minimize loads.

$$\begin{array}{c} I_x \\ I_y \\ I_z \end{array}\begin{pmatrix} 0* \\ 2* \\ 0* \end{pmatrix} \rightarrow \begin{pmatrix} 0 \\ 0* \\ 2* \end{pmatrix} \rightarrow \begin{pmatrix} 0 \\ 1* \\ 1* \end{pmatrix} \rightarrow \begin{pmatrix} 1* \\ 1 \\ 0* \end{pmatrix} \rightarrow \begin{pmatrix} 1 \\ 0* \\ 1* \end{pmatrix} \rightarrow \begin{pmatrix} 2* \\ 0 \\ 0* \end{pmatrix}$$

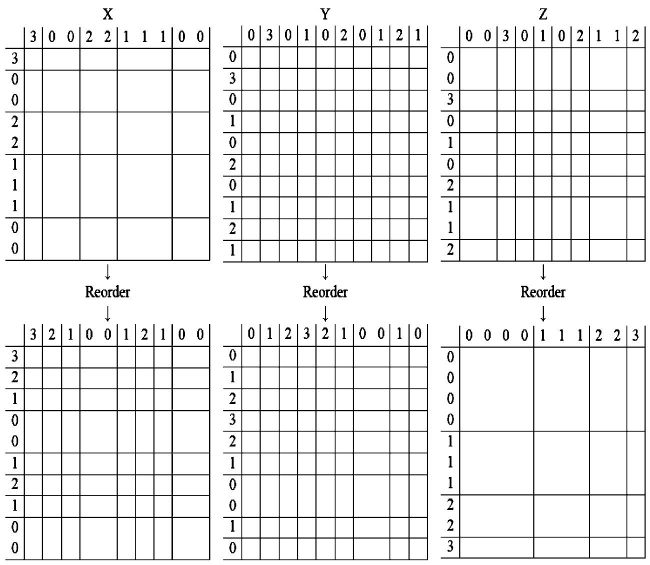
The above order may differ from the requirements of an application, however, restoring the desired ordering is trivial. In the $(ff|ff)$ ERI case, mapping three indices to threads is not possible, as it requires 1000 threads. However, we can map the i and j indices and loop over the k and l indices in a similar fashion, as outlined in Algorithm 3.

Algorithm 3: CUDA Rys Quadrature, *i* and *j* Mapping

```
Algorithm 3 CUDA Rys quadrature, $i, j$ mapping
  ## map threads to ERI elements
  $i = threadIdx.x$
  $j = threadIdx.y$
  ## lookup Cartesian exponents
  $(ix, iy, iz) = (LX[i], LY[i], LZ[i])$
  $(jx, jy, jz) = (LX[j], LY[j], LZ[j])$
  for all $kl_{z-block}$ do
    sync threads
    $I_{z,shmem} = I_z(:, :, LZ[k], LZ[l])$
    ## load 2-$D$ integrals to shmem
    for all $kl_{xy} \in kl_{z-block}$ do
      sync threads
      $I_{x,shmem} = I_x(:, :, LX[k], LX[l])$
      $I_{y,shmem} = I_y(:, :, LY[k], LY[l])$
      sync threads
      $I(i, j, k, l) = \sum_N I_{x,shmem}(ix, jx) I_{y,shmem}(iy, jy) I_{z,shmem}(iz, jz)$
    end for
  end for
```

The shared memory requirement for an $(ff|ff)$ ERI is $N(L_a + 1)(L_b + 1) = 7(4^2) = 112$ elements for each 2-D memory block. The memory access can likewise be reduced by a third, if the shells are reordered. Blocking of kl_z indices was used, as outlined in Table 1 for a $|ff)$ case. In the example, the number of memory loads is 216. The first row of Table 1 shows the data access pattern for x, y, and z 2-D integrals when the canonical ERI ordering is used. There are some block patterns that are visible in the x and z dimensions. But these blocking patterns are not optimal from the perspective of data reuse because the blocks are small. To improve the overall memory performance, the integrals can be reordered such that one of the 2-D integrals has a well-defined block structure, for example, the z integral.

Table 1. Index Ordering for $|ff)$ Case



4.4. Template-based Code Generation. If the cases described above are implemented in CUDA directly, then the register usage is high. To reduce the register use, the loops over the outer indices can be unrolled explicitly for each possible case, e.g. for $|pp)$, $|pd)$, $|pf)$, etc. Programming all of the cases by hand is prohibitive, as it requires a large amount of code. However, using a template-based approach, all of the cases can be generated automatically from a single template.

There exists a number of template engines, e.g., the venerable *m*4 macro processor,[22] but the Python-based[21] Cheetah template engine[24] is chosen for this project. In Cheetah templates, the Python statements that control the code generation are embedded directly in the source code, similar to the manner in which traditional C preprocessor directives are used. Other benefits of using Cheetah are the ability to write complex support modules in Python and to reuse existing Python utilities.

Since generating code from a template is straightforward, the root summation loops were also explicitly unrolled. This was done to allocate registers to store a single set of 2-D integrals in registers rather than in shared memory. The benefit of doing so is that the use of shared memory and the bank conflicts are reduced. All of the shared memory is arranged in banks; the number of banks for the current hardware is 16, i.e., half-warp size. A bank conflict arises when multiple threads in a half-warp access different memory locations mapped onto the same bank, simultaneously resulting in the serialization of threads in the half-warp.[17] The bank conflicts occur often if the leading dimension (in this case, the number of roots, N) is a divisor of the bank size; accesses to an array with a leading dimension of 8 causes 8 bank conflicts. For other cases, the bank conflicts occur much less often; a leading dimension of 7 causes only one bank conflict. The bank conflicts lead to warp serialization, where the warp threads execute the instructions sequentially rather than executing the same instruction in a single instruction multiple thread (SIMT) or a lock-step fashion. Warp serialization is highly undesirable, and bank conflicts are a serious performance issue, degrading the overall performance by about 25%. The bank conflicts are especially pronounced when the number of roots is even.

5. Results and Discussion

The performance of the implementation is evaluated on the NVIDIA GeForce GTX 275 and the Tesla T10 processors. The GTX 275 is a regular graphics card with a double precision support, an 1 GB of device memory, and a clock speed of 1.1 GHz. The Tesla T10 is a dedicated HPC accelerator with double precision (DP) support, 4 GB of device memory, and a clock speed of 1.35 GHz. The Tesla processor is capable of delivering approximately 20 GFLOPS on the level 2 BLAS double precision general matrix vector (DGEMV) routine and 70 GFLOPS on the level 3 BLAS double precision general matrix multiply (DGEMM) routine. The GTX 275 is approximately 25% slower than the Tesla.

The basis set used was purely synthetic with exponents of 1.5, which is representative of the values for higher angular momentum functions. The quoted timings do not include the GPU−CPU communication time.

The performance of the quadrature was evaluated by counting the total number of quadrature operations given by

$$\text{flops} = n_{\text{block}} 3N \binom{L_a + 2}{2} \binom{L_b + 2}{2} \binom{L_c + 2}{2} \binom{L_d + 2}{2}$$

where n_{block} is the number of ERI/thread blocks. The above metric also accounts for multiplication by a constant factor that incorporates contraction coefficients and normalization factors. For example, the (*ff*|*ff*) block requires a $21(10^4) = 210000$ flop count. The total flop count is divided by the execution time on the GPU to obtain the GFLOPS metric. The execution time does not include the memory transfer overheads between the host and the GPU. The transfer time latency of the ERIs from GPU to host is several times longer than that of the actual execution time.

The performance results on the GTX 275 and Tesla boards are presented in Tables 2 and 3, respectively. As can be seen from these tables, the performance depends to a large degree on the ERI class. The larger ERI classes (i.e., higher collective angular momentum) perform better on average than the smaller classes. The computations with an odd number of roots, cf., eq 13, e.g., (*gg*|*dd*), (*ff*|*ff*), etc., tend to have fewer bank conflicts than those with an even number of roots, e.g., (*gg*|*ff*), as discussed in Section 4. Consequently, the performance of ERI classes with an odd number of roots is higher, as much as 25% in an extreme case. The difference between the single and double precision performance is roughly a factor of 2, as previously predicted by Ufimtsev and Martinez.[25] One would expect this difference to favor single precision even more strongly, since the number of SP units is eight times the number of DP units. This suggests that the computations are memory bound rather than compute bound. The performance depends heavily on the mapping used, cf., Section 4.3. As one would expect, the "larger" *i*, *j*, and *k* mapping performs better than the *i* and *j* mapping for cases with lower *i* and *j* angular momenta (such as the (*pp*|*ff*) ERI block), since the shared memory reuse and parallelism is much higher. The difference between the two mappings for the same ERI class can be as high as a factor of 5. However, when the *i* and *j* angular momenta are higher (such as the (*ff*|*pp*) ERI block), the *i* and *j* mapping is only slightly outperformed by the *i*, *j*, and *k* mapping. Interestingly, comparing the best performance for the (*pp*|*ff*) and (*ff*|*pp*) ERI block gives very similar performance. One could use ERI index symmetry such as (*ij*|*kl*) = (*kl*|*ij*) in these cases to ensure that the first two indices are always lower, so the *i*, *j*, and *k* mapping algorithm could always be used when the memory is available.

The difference in performance between the generic GTX GPU and the Tesla T10, presented in Table 4, is 25−30% across the single and double precision performance. This is

Table 2. CUDA Rys Quadrature Performance on GeForce GTX 275

			GFLOPS$_{SP}$[c]		GFLOPS$_{DP}$[d]	
ERI	blocks[a]	flop count[b]	map$_{ijk}$[e]	map$_{ij}$[e]	map$_{ijk}$[e]	map$_{ij}$[e]
(*gg*|*gg*)	2000	2733750000	n/a	**45.23**	n/a	**22.55**
(*gg*|*ff*)	4000	2160000000	n/a	**34.42**	n/a	**15.32**
(*ff*|*gg*)	4000	2160000000	n/a	**30.91**	n/a	**14.11**
(*gg*|*dd*)	10000	1701000000	n/a	**43.08**	n/a	**21.05**
(*dd*|*gg*)	10000	1701000000	n/a	**23.63**	n/a	**16.35**
(*gg*|*pp*)	40000	1458000000	n/a	**36.53**	n/a	**17.08**
(*pp*|*gg*)	40000	1458000000	**34.23**	6.93	**18.20**	5.38
(*ff*|*ff*)	10000	2100000000	n/a	**40.43**	n/a	**20.11**
(*ff*|*dd*)	20000	1296000000	n/a	**37.54**	n/a	**18.29**
(*dd*|*ff*)	20000	1296000000	**37.69**	23.32	**16.53**	15.04
(*ff*|*pp*)	80000	1080000000	27.43	**31.46**	15.23	**17.05**
(*pp*|*ff*)	80000	1080000000	**32.23**	6.21	**17.45**	4.84
(*dd*|*dd*)	60000	1166400000	**31.10**	20.17	**16.38**	13.67
(*dd*|*pp*)	200000	777600000	19.71	**20.25**	11.54	**11.70**
(*pp*|*dd*)	200000	777600000	**20.18**	5.16	**11.11**	3.85
(*pp*|*pp*)	750000	546750000	**11.93**	4.79	**8.43**	3.76

[a] Blocks are the number of ERI blocks evaluated. [b] Flop count is the total floating point operations. [c] GFLOPS$_{SP}$ is the single precision performance. [d] GFLOPS$_{DP}$ is the double precision performance. [e] Map is the ERI to thread mapping; the best performing mapping is shown in bold.

Table 3. CUDA Rys Quadrature Performance on Tesla GPU

			GFLOPS$_{SP}$[c]		GFLOPS$_{DP}$[d]	
ERI	blocks[a]	flop count[b]	map$_{ijk}$[e]	map$_{ij}$[e]	map$_{ijk}$[e]	map$_{ij}$[e]
(*gg*|*gg*)	2000	2733750000	n/a	**55.97**	n/a	**27.34**
(*gg*|*ff*)	4000	2160000000	n/a	**42.07**	n/a	**18.67**
(*ff*|*gg*)	4000	2160000000	n/a	**37.70**	n/a	**17.19**
(*gg*|*dd*)	10000	1701000000	n/a	**53.39**	n/a	**25.34**
(*dd*|*gg*)	10000	1701000000	n/a	**31.71**	n/a	**19.87**
(*gg*|*pp*)	40000	1458000000	n/a	**45.15**	n/a	**20.65**
(*pp*|*gg*)	40000	1458000000	**42.42**	7.78	**22.09**	6.19
(*ff*|*ff*)	10000	2100000000	n/a	**50.19**	n/a	**24.46**
(*ff*|*dd*)	20000	1296000000	n/a	**46.15**	n/a	**22.44**
(*dd*|*ff*)	20000	1296000000	**45.71**	28.46	**19.71**	18.29
(*ff*|*pp*)	80000	1080000000	33.86	**39.38**	18.54	**20.10**
(*pp*|*ff*)	80000	1080000000	**40.33**	7.02	**21.46**	5.63
(*dd*|*dd*)	60000	1166400000	**38.74**	23.38	**19.78**	15.62
(*dd*|*pp*)	200000	777600000	24.67	**25.00**	14.20	**14.33**
(*pp*|*dd*)	200000	777600000	**25.22**	7.67	**13.73**	4.33
(*pp*|*pp*)	750000	546750000	**14.17**	5.37	**10.00**	4.30

[a] Blocks are the number of ERI blocks evaluated. [b] Flop count is the total floating point operations. [c] GFLOPS$_{SP}$ is the single precision performance. [d] GFLOPS$_{DP}$ is the double precision performance. [e] Map is the ERI to thread mapping; the best performing mapping is shown in bold.

Table 4. GTX 275, Tesla, and GAMESS Performance Comparison

			GFLOPS$_{SP}$[c]		GFLOPS$_{DP}$[d]		GLFOPS[e]
ERI	blocks[a]	flop count[b]	GTX 275	Tesla	GTX 275	Tesla	GAMESS
(*gg*|*gg*)	2000	2733750000	45.23	55.97	22.55	27.34	1.36
(*gg*|*ff*)	4000	2160000000	34.42	42.07	15.32	18.67	1.29
(*ff*|*gg*)	4000	2160000000	30.91	37.70	14.11	17.19	1.32
(*gg*|*dd*)	10000	1701000000	43.08	53.39	21.05	25.34	1.09
(*dd*|*gg*)	10000	1701000000	23.63	24.03	16.35	29.88	1.21
(*gg*|*pp*)	40000	1458000000	36.53	45.15	17.08	20.65	0.82
(*pp*|*gg*)	40000	1458000000	34.23	42.42	18.20	22.09	0.98
(*ff*|*ff*)	10000	2100000000	40.43	50.19	20.11	24.46	1.19
(*ff*|*dd*)	20000	1296000000	37.54	46.15	18.29	22.44	0.94
(*dd*|*ff*)	20000	1296000000	37.69	45.71	16.53	19.71	1.03
(*ff*|*pp*)	80000	1080000000	31.46	39.38	17.05	20.10	0.75
(*pp*|*ff*)	80000	1080000000	32.23	40.33	17.45	21.46	0.78
(*dd*|*dd*)	60000	1166400000	31.10	38.74	16.38	19.78	0.79
(*dd*|*pp*)	200000	777600000	20.25	25.00	11.70	14.33	0.63
(*pp*|*dd*)	200000	777600000	20.18	25.22	11.11	13.73	0.66
(*pp*|*pp*)	750000	546750000	11.93	14.17	8.43	10.00	0.48

[a] Blocks are the number of ERI blocks evaluated. [b] Flop count is the total floating point operations. [c] GFLOPS$_{SP}$ is the single precision performance. [d] GFLOPS$_{DP}$ is the double precision performance. [e] Map is the ERI to thread mapping; the best performing mapping is shown in bold.

consistent with the higher clock speed of the Tesla compute device. In terms of registers and shared memory, Tesla does not have an advantage over the generic GTX GPU. This is reflected in the same mapping on both GPUs having the best performance for a particular ERI case. The performance of the Rys quadrature is on par with the performance of the (S/G) DGEMV routines in the CUBLAS library. In terms of the peak theoretical performance, it is possible to achieve approximately 30% in the best case. The poor-performing lower angular momentum ERI classes utilize the hardware at 10% efficiency in the worst case. However, the two mapping implementations (*ijk* and *ij*) are not specifically optimized for these ERI classes.

Previous work in this field focused on *s* and *p* integrals, which produce small, highly contracted integral blocks that can be performed entirely in shared memory and registers. Therefore, the ratio of computation to memory traffic is high. Moreover, on Telsa and older architectures, there is a ratio of 8:1 for single vs double precision floating point units. For higher angular momentum integrals, computation cannot be done entirely in shared memory and registers, so partial values must be read and stored in global memory. This helps to explain the difference in performance between this work and previous work.

The difference in performance between the GPU and the CPU, which is also presented in Table 4, is very promising. The Rys quadrature used in GAMESS, which was also used as a benchmark by Ufimtsev and Martinez, is a legacy FORTRAN implementation that underperforms on modern CPUs. As can be seen from Table 4, the original Rys quadrature implementation is only 15% efficient at best on a modern 8 GFLOP processor. In order to achieve good performance on both CPU and GPU, the algorithm must be implemented in a way suitable for instruction level parallelism.

6. Conclusions and Future Work

This work has demonstrated the ability to obtain comparable or better performance to that of an optimized DGEMV routine for a Rys quadrature implementation of two-electron integral computations—a core computation for electronic structure algorithms. Since the focus of this work is on higher angular momentum integrals, the use of the double precision units on the GPU is also highlighted. In order to achieve the best performance, memory access patterns and data reuse have been optimized. The code implementation has been greatly facilitated by using the template-based code generator. Not only does it allow for fast prototyping of various algorithms, it also provides a developer-friendly framework for the developer to focus on the main issues associated with the algorithm and allows the details associated with the many angular momenta and single vs double precision cases to be handled automatically. The use of templates could eventually be taken one step further, so that the generated code could be optimized further, depending on the GPU architecture. However, this may prove to be impractical as the NVIDIA compiler is refined to take advantage of different GPU architectures.

Some improvements are still possible with respect to data reuse, but the gains are unlikely to be high. The improvements would be due to more aggressive memory caching and memory access pattern reordering. The greatest overall improvement will come from reusing the ERI blocks as soon as they are formed on the GPU, e.g., to construct the Fock matrix. The 2-D Fock matrix is formed from 4-D ERI blocks; so, if computed on the GPU device, then the memory transfer would just be that of the Fock matrix (of order M^2 where M is the number of basis functions) rather than those of all the ERI blocks. Therefore, the computation of the Fock matrix on the GPU increases the flop count and reduces the amount of memory to be transferred to the host, resulting in overall greater performance.[17] The direct use of the ERIs on the GPU device is necessary, as the memory transfer of the raw ERIs between GPU and host is several times longer than the computation itself. The amount of data that must be transferred from the host to the GPU to start the computation is small; moreover, since it is small, it can be transferred asynchronously while the computation is running.

The contracted ERIs and the ERIs of small angular momentum functions have not been addressed directly in this work. The implementation of the Rys quadrature roots recurrence and transfer relationships is also not discussed explicitly but will be presented in a future publication. The accuracy and utility of single precision vs double precision computations will be considered in future work. In addition, future work will include the incorporation of the ERIs into modern algorithms for full electronic structure theory calculations.

Acknowledgment. This work has been accomplished by Iowa State University of Science and Technology under contract no. DE-AC02-07CH11358 with the United States Department of Energy and under an National Science Foundation PetaApps grant. The GPU cluster was provided by a DURIP grant from the Department of Defense, with matching funds from the Iowa State University Frances M. Craig Professorship to M.S.G. and the NVIDIA Corporation. M.S.G. gratefully acknowledges an IBM Faculty Fellowship award. The authors have benefitted from many helpful discussions with Professor Todd Martinez and his students.

References

(1) Hohenberg, P.; Kohn, W. *Phys. Rev.* **1964**, *136*, B864.

(2) Almlöf, J.; Faegri, K., Jr.; Korsell, K. *J. Comput. Chem.* **1982**, *3*, 385–399.

(3) Bartlett, R. J.; Purvis, G. D. *Int. J. Quantum Chem.* **1978**, *14*, 561–581.

(4) Boys, S. F. *Proc. R. Soc. Lond. A* **1950**, *200*, 542.

(5) Rys, J.; Dupuis, M.; King, H. *J. Comput. Chem.* **1983**, *4*, 154–157.

(6) Pople, J.; Hehre, W. *J. Comput. Phys.* **1978**, *27*, 161–168.

(7) McMurchie, L. E.; Davidson, E. R. *J. Comput Phys.* **1978**, *26*, 218–231.

(8) Obara, S.; Saika, A. *J. Chem. Phys.* **1988**, *89*, 1540–1559.

(9) Head-Gordon, M.; Pople, J. A. *J. Chem. Phys.* **1988**, *89*, 5777–5786.

(10) Lindh, R.; Ryu, U.; Liu, B. *J. Chem. Phys.* **1991**, *95*, 5889–5897.

(11) Dupuis, M.; Marquez, A. *J. Chem. Phys.* **2001**, *114*, 2067–2078.

(12) 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.

(13) Rys, J.; Dupuis, M.; King, H. *J. Comput. Phys.* **1976**, *21*, 144.

(14) Sagar, R. P.; Smith, V. H., Jr. On the calculation of Rys polynomials and quadratures. *Int. J. Quantum Chem.* **1992**, *42*, 827–836.

(15) Volkov, V.; Demmel, J. W. Benchmarking GPUs to tune dense linear algebra. In *SC '08: Proceedings of the 2008 ACM/IEEE conference on Supercomputing*; IEEE Press: Piscataway, NJ, 2008.

(16) Nickolls, J.; Buck, I.; Garland, M.; Skadron, K. *ACM Queue* **2008**, *6*, 40–53.

(17) *NVIDIA CUDA Programming Guide*; NVIDIA Corporation: Santa Clara, CA, 2009; http://developer.download.nvidia.com/compute/cuda/2_3/toolkit/docs/NVIDIA_CUDA_Programming_Guide_2.3.pdf.

(18) Ufimtsev, I. S.; Martinez, T. J. *J. Chem. Theory Comput.* **2008**, *4*, 222–231.

(19) Ufimtsev, I. S.; Martinez, T. J. *J. Chem. Theory Comput.* **2009**, *5*, 1004–1015.

(20) Yasuda, K. *J. Comput. Chem.* **2008**, *29*, 334–342.

(21) Shi, G.; Kindratenko, V.; Ufimtsev, I. S.; Martinez, T. J. Two-Electron Integral Evaluation on FPGA, CELL and GPU accelerators. Poster presentation in Path to Petascale: *Adapting GEO/CHEM/ASTRO Applications for Accelerators and Accelerator Clusters Workshop*; National Center for Supercomputing Applications: Urbana, IL, 2009.

(22) *GNU m4 Macro Processor Manual*; Free Software Foundation, Inc.: Boston, MA, 2007; http://www.gnu.org/software/m4/manual/m4.html.

(23) Pilgrim, M. *Dive Into Python*; CreateSpace: Paramount, CA, 2009.

(24) Martelli, A. *Python in a Nutshell (In a Nutshell (O'Reilly))*; O'Reilly Media, Inc. Sebastapol, CA, 2006.

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

CT9005079