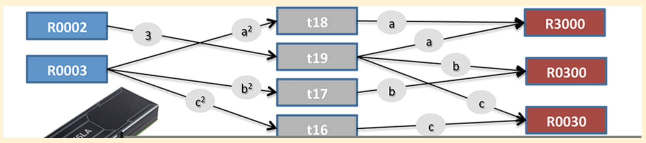
Generating Efficient Quantum Chemistry Codes for Novel Architectures

Alexey V. Titov,[†,‡] Ivan S. Ufimtsev,[‡] Nathan Luehr,[‡] and Todd J. Martinez*,[‡]

[†]National Center for Supercomputing Applications, University of Illinois at Urbana−Champaign, Urbana, Illinois 61801, United States

[‡]Department of Chemistry and the PULSE Institute, Stanford University, Stanford, California 94305, United States

Supporting Information



ABSTRACT: We describe an extension of our graphics processing unit (GPU) electronic structure program TeraChem to include atom-centered Gaussian basis sets with d angular momentum functions. This was made possible by a "meta-programming" strategy that leverages computer algebra systems for the derivation of equations and their transformation to correct code. We generate a multitude of code fragments that are formally mathematically equivalent, but differ in their memory and floating-point operation footprints. We then select between different code fragments using empirical testing to find the highest performing code variant. This leads to an optimal balance of floating-point operations and memory bandwidth for a given target architecture without laborious manual tuning. We show that this approach is capable of similar performance compared to our hand-tuned GPU kernels for basis sets with s and p angular momenta. We also demonstrate that mixed precision schemes (using both single and double precision) remain stable and accurate for molecules with d functions. We provide benchmarks of the execution time of entire self-consistent field (SCF) calculations using our GPU code and compare to mature CPU based codes, showing the benefits of the GPU architecture for electronic structure theory with appropriately redesigned algorithms. We suggest that the meta-programming and empirical performance optimization approach may be important in future computational chemistry applications, especially in the face of quickly evolving computer architectures.

■ INTRODUCTION

Quantum chemistry programs have long pushed the limits of code complexity, and the idea of using metacodes (codes which generate code) for computational chemistry can be traced back to the early 80s.[1] A computer-algebraic approach for molecular integrals was pioneered by Jones to automate the generation of overlap integrals over Slater-type orbitals.[2] Notably, this effort predated the widespread availability of general purpose computer algebra systems (CAS) such as Maple[3] and Mathematica.[4] As CAS software matured, a number of applications to the derivation of integrals for quantum chemistry applications appeared. Scott and coauthors demonstrated a way to optimize FORTRAN coding of certain molecular integrals by using built-in methods of Maple.[5,6] The same year, Bracken and Bartlett implemented and derived certain recursive relations for two- and three-center molecular integrals using Maple.[7] They also argued that these relations could potentially be useful to generate code in computationally efficient programming languages. Interest in the automatic generation of integrals code has been renewed by Fermann and Valeev with the integral generation library LIBINT.[8] This library provides an integrals module for CPU-based architectures, which can be embedded in electronic structure programs. To the best of our knowledge, besides the ease of integral generation, the performance gains of the chosen approach have yet to be quantified.

Automated computer algebra has also been used for other purposes in quantum chemistry. For example, at least two groups have used a CAS with FORTRAN code generation to efficiently implement the exchange-correlation functionals (and their functional derivatives) needed in density functional theory.[9,10] Additionally, the complexity of many body theories such as coupled cluster led to the development of the tensor contraction engine (TCE), which automates the derivation of many-body expressions and also the manipulation and storage of intermediates.[11,12] In fact, the TCE goes beyond simple leveraging of a CAS and may be viewed as a specialized standalone CAS combined with a domain-specific language and compiler.

Previous computational chemistry efforts in the area of code-automation have primarily dealt with the tedium and correctness of developed code. Our focus here is to demonstrate an effective metacode approach that allows both rapid development and efficient use of novel architectures for computational chemistry. Figure 1 shows a schematic of our implementation strategy. Symbolic metacodes embody the algebraic expressions in an abstract mathematical form. These expressions are manipulated and transformed using a CAS and emitted as source code fragments (in C or FORTRAN) that are translated (using shell scripts in our implementation) to the

Received: April 23, 2012
Published: October 30, 2012

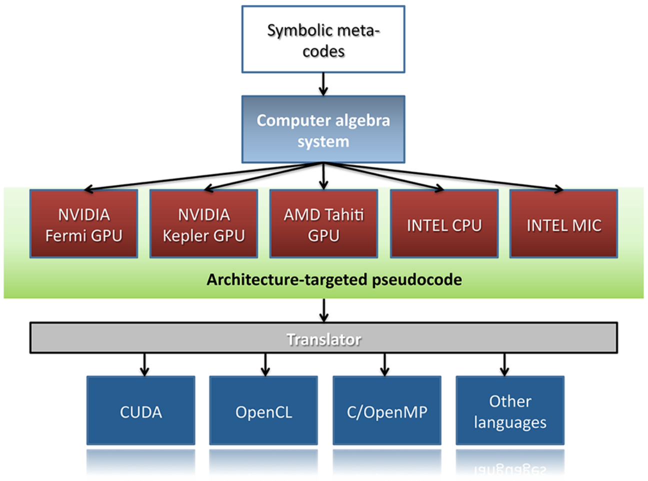


Figure 1. Layered structure comprising the metacode approach. The top consists of a set of analytical rules for the computational codes. The mathematical rules are mapped directly to computer algebra system (Maple in our case). The computer algebra tools are used to tailor the codes in the symbolic form to the specific hardware architecture. With the help of translators (e.g., *sed* scripts), the tailored code is transformed to meet the syntactic requirements of the source code language (e.g., CUDA). Using the right translator module, multiple target source codes can be generated from the same architecture specific meta-code, for example, CAS→NVIDIA GPU→Translator→CUDA and CAS→NVIDIA GPU→Translator→OpenCL.

desired language such as CUDA, C/C++, or OpenCL. By using a CAS we guarantee the correctness of the final expressions and also simplify the generation of multiple code variants accomplishing the same mathematical goal. With multiple code variants in hand, we can test the performance of each on a given architecture and select the best one. Manipulation of the implementation details through symbolic algebra followed by automated empirical testing allows us to effectively custom-derive algorithms to meet the strengths and limitations of specific architectures such as graphics processing units (GPUs). Although we focus on CUDA and NVIDIA GPUs in this article, we have also used the same system to generate OpenCL code for Intel CPUs and AMD and NVIDIA GPUs.

Accelerating molecular modeling on GPUs has recently shown remarkable potential for scientific applications. Provided with flexible application programming interfaces to GPUs and related architectures, the efficiency of such acceleration has already been convincingly demonstrated in many areas of computational chemistry.[13] These include electron repulsion integrals (ERIs),[14−17] self-consistent field (SCF) algorithms in Hartree−Fock (HF) and density functional theory (DFT),[18−21] low-order perturbation theory,[22,23] and quantum Monte Carlo.[24,25] A critical bottleneck in most implementations of HF and DFT is the evaluation of ERIs that formally scales as $O(N^4)$, where N is the total number of basis functions. Even though this scaling can be reduced to $O(N^2)$ for large systems with integral screening, ERI evaluation often dominates the computation time. Alternative techniques which decompose the integrals using density-fitting,[26−28] pseudospectral methods,[29,30] or Cholesky decompositions[31−33] have been explored to alleviate this bottleneck, but these have various disadvantages such as the introduction of numerical grids, fitting basis sets, or difficulties in the efficient treatment of exchange. We previously demonstrated that performance speedups of up to 650× over conventional processors can be achieved in HF and DFT calculations with atom-centered Gaussian basis functions of s- and p-type angular momentum if the ERI evaluation and subsequent formation of Coulomb and exchange operators is reformulated, parallelized, and accelerated on NVIDIA graphics cards.[14,18,34] The evaluation of two-electron integrals involving higher angular momentum functions was recently implemented on NVIDIA GPUs by Asadchev et al.[16] This ERI implementation was based on the Rys quadrature method,[35] which exhibits a relatively small memory footprint ideal for efficient code on modern stream processors. Unfortunately, the scheme became significantly less efficient as the angular momentum of the basis functions increased.

Apart from low memory requirements, various other algorithmic and hardware aspects need to be considered when designing efficient programs for modern accelerators. It is becoming evident that state-of-the-art and future high-performance computing hardware will be based on design principles similar to those used in GPUs: massively parallel processors supporting thousands (or more) physical execution threads (instruction streams executed in parallel) and a pronounced memory hierarchy for computation and data exchange. Therefore any efficient code developed for high-performance computers (and apparently also commodity hardware like consumer graphics cards) must embrace massive parallelism and explicit data management. In the present article we do not discuss architectures and programming techniques in detail, as these are well described elsewhere.[16,18,19] Instead we highlight important considerations for successful development of fast GPU codes that we believe will be transferable to future computing platforms.

In the context of streaming architecture execution[36] as exemplified by the GPU, threads are lightweight (i.e., they incur little overhead in terms of stack and other resources) and exchange data through either on-chip shared memory or global memory (DRAM), stressing the importance of fine-grained parallelism. Any efficient implementation should minimize the use of DRAM because of its high latency and limited bandwidth and instead should use fast shared memory to exchange data. When global memory accesses are unavoidable, threads should access DRAM contiguously because multiple memory operations can then be coalesced into one memory load/store instruction, maximizing instruction throughput. Otherwise, the application memory bandwidth drops significantly, undermining the high performance benefit of GPUs. It is always desirable to issue as many arithmetic instructions (FLOPs) as possible with respect to the number of global memory operations (mops), storing intermediates in a (necessarily limited) number of very fast local memory registers. Finally, it is vitally important to carefully orchestrate memory accesses to eliminate potential race conditions because of the large number of parallel threads and their undefined execution order. In many cases, it is more effective to recalculate intermediates as needed (instead of storing and reusing them) to reduce the required memory and/or eliminate dependencies that would hamper parallelization.[18]

The many hardware features and limitations make writing efficient quantum chemistry code for GPUs a daunting task. This problem is further amplified by the fact that different architectures can require customized code optimizations. Even successive hardware generations from the same manufacturer, such as NVIDIA G80, GT200, and Fermi can have profoundly different capabilities. For example, code optimized for GT200 could easily be far from optimal on Fermi GPUs. Furthermore, different manufacturers provide different programming languages for their products (e.g., Nvidia CUDA and AMD CTM). The recently introduced OpenCL standard aims to create a unified framework for cross-platform code development. However, OpenCL represents all existing accelerators as a single generic device with a standard ("lowest common denominator") set of capabilities. Therefore, OpenCL can solve the code portability problem but often cannot provide optimal performance across a variety of architectures.[37]

In this article we present a framework for development of efficient electronic structure codes for various computational platforms. We abandon the approach of hand-written and manually optimized code, and use a computer algebra system and shell scripting to automatically generate efficient GPU and CPU kernels for ERI evaluation with up to d-type basis functions. Both sets of kernels are generated from the same equations expressed in symbolic form, demonstrating the flexibility of our approach. The first part of the paper focuses on the correctness of the generated code, and the second part is on the generation and optimization of different code variants. The performance of the GPU and CPU-based ERI implementations is assessed through a number of HF benchmarks and comparison with GAMESS, a mature third-party quantum chemistry package.[38] We show that empirical performance testing of the autogenerated code variants (each of which corresponds to the same set of equations, but which differs in the intermediates used, recursion pathways, and other implementation details) can lead to an increase in computational performance by more than two times (as shown in Table 1), using the GPU kernel that evaluates [pp|pp] integrals, as an example.

Table 1. Performance Metrics for a Distribution of Various Generated Kernels, Evaluating Exchange Integrals for the Same [pp|pp] Batch

code variant	C1060 timing (ms)	C2050 timing (ms)	registers[a]	FLOPs[b]
1	1025.26	822.57	115	1049
2	1042.57	823.99	115	1083
3	1112.64	988.97	114	1218
4	1117.36	1151.79	120	2124
5	2303.17	2511.44	145	1185
6	2523.15	2780.31	171	2012
7	2077.94	2852.86	141	1931

[a]The register-per-thread count is based on the compilation for architecture with CC 1.3. Maximum number of registers per thread for CC 1.3 and CC 2.0 is 124 and 63, respectively. [b]Number of FLOPs inside the innermost loop.

■ OVERVIEW OF INTEGRAL AND OPERATOR GENERATION

We begin with an overview of the general strategy for evaluating the two-electron repulsion integrals, based on the McMurchie-Davidson algorithm.[39] These integrals are given by

$$(ij|kl) = \iint \frac{\phi_i(r_1)\phi_j(r_1)\phi_k(r_2)\phi_l(r_2)}{|r_1 - r_2|}\, dr_1\, dr_2 \tag{1}$$

where r_i labels the coordinates for the ith electron. The one-electron basis function ϕ on the Ath atomic center is given as:

$$\phi(r) = N_{lmn} x_A^n y_A^l z_A^m e^{-\alpha r_A^2} \tag{2}$$

where N_{lmn} is a normalization constant and (x_A, y_A, z_A) are the components of the vector $\vec{r}_A = \vec{r} - \vec{A}$ with norm r_A.

In practical calculations, the integrals themselves are only of secondary interest—the primary objective is the generation of the required operators, that is, Coulomb (J) and exchange (K) operators. These are contractions of the integrals with the density matrix elements, for example,

$$(J[P])_{ij} = \sum_{kl} P_{kl}(ij|kl) \tag{3}$$

and

$$(K[P])_{ij} = \sum_{kl} P_{kl}(ik|jl) \tag{4}$$

where P is the density matrix corresponding to the wave function. An efficient code does not separate the integral generation and density matrix contraction steps, but overlays them to some degree. This is most pronounced in the construction of the Coulomb operator, where Ahmadi and Almlof showed[40] that it was possible to carry out the contraction with density matrix elements during the generation of the integrals. This early contraction strategy was further refined in the "J-engine" algorithm.[41]

Usually, the one-electron basis functions are not single Gaussians (primitive basis functions) as shown in eq 2, but instead taken as fixed sums (contracted basis functions) over a set of Gaussians, all centered on the same atom. For simplicity, we limit our discussion to primitive basis functions. This follows our implementation, which generates J and K operators in the basis of primitive functions first, and then contracts these to generate the J and K operators in the basis of contracted functions. However, we note that several algorithms for integral generation have focused on generating the J/K operators in the contracted basis directly.[42−44] This leads to further flexibility that could be exploited within our CAS framework in the future.

McMurchie-Davidson Integration Algorithm. The first step of evaluating the integrals is to use the well-known Gaussian Product Rule to convert each four center integral $[ab|cd]$ into a two center integral $[\chi_P|\chi_Q]$ where (ignoring normalization):

$$\chi_P(r_1) = \phi_a(r_1)\phi_b(r_1) = x_A^n x_B^{n'} y_A^l y_B^{l'} z_A^m z_B^{m'} \tilde{E}_{ab} e^{-\alpha_P r_P^2} \tag{5}$$

and

$$\tilde{E}_{ab} = e^{-\alpha_a\alpha_b/\alpha_a+\alpha_b R_{AB}^2}$$
$$\alpha_P = \alpha_a + \alpha_b$$
$$\vec{P} = \frac{\alpha_a\vec{A} + \alpha_b\vec{B}}{\alpha_a + \alpha_b} \tag{6}$$

and χ_Q is defined analogously. The heart of the McMurchie-Davidson algorithm[39] is the exact expansion of the pair distributions, χ_P and χ_Q, in a basis of Hermite Gaussians Λ_i:

$$x_A^{n_A} x_B^{n_B} = \sum_{N=0}^{n_A+n_B} d_N^{n_A,n_B} \Lambda_N(x_P, \alpha_P) \tag{7}$$

$$\Lambda_j(x_p, \alpha_p)e^{-\alpha_p x_p} = \left(\frac{\partial}{\partial P_x}\right)^j e^{-\alpha_p x_p} \tag{8}$$

The expansion coefficients $d_N^{n_A,n_B}$ are obtained via the following recursion relations, where $PA_x = P_x - A_x$, $PB_x = P_x - B_x$, and $d_0^{0,0} = 1$:

$$d_N^{n_A+1,n_B} = (2\alpha_p)^{-1}d_{N-1}^{n_A,n_B} + PA_x d_N^{n_A,n_B} + (N+1)d_{N+1}^{n_A,n_B} \tag{9}$$

$$d_N^{n_A,n_B+1} = (2\alpha_p)^{-1}d_{N-1}^{n_A,n_B} + PB_x d_N^{n_A,n_B} + (N+1)d_{N+1}^{n_A,n_B} \tag{10}$$

Expansion in Hermite Gaussians allows the Cartesian polynomials in eq 5 to be written as a sum of derivatives with respect to the coordinate $\vec{P}$. These derivatives can then be taken outside of the integrals over electronic coordinates r_1 and r_2 in eq 1. This leaves only the auxiliary functions R_{NLMJ} which are defined in terms of $T = |\vec{P} - \vec{Q}|^2$ and the Boys function, defined as:

$$F_j(T) = \int_0^1 u^{2J}e^{-Tu^2}du \tag{11}$$

$$R_{NLMJ} = \left(\frac{\partial}{\partial P_x}\right)^N \left(\frac{\partial}{\partial P_y}\right)^L \left(\frac{\partial}{\partial P_z}\right)^M F_j(T) \tag{12}$$

The required set of auxiliary functions R_{NLMJ} can be conveniently evaluated according to the following recursion relations.

$$\begin{aligned} R_{0,0,M+1,j} &= c\cdot R_{0,0,M,j+1} + M\cdot R_{0,0,M-1,j+1} \\ R_{0,L+1,M,j} &= b\cdot R_{0,L,M,j+1} + L\cdot R_{0,L-1,M,j+1} \\ R_{N+1,L,M,j} &= a\cdot R_{N,L,M,j+1} + N\cdot R_{N-1,L,M,j+1} \end{aligned} \tag{13}$$

where the prefactors a, b, and c depend on the atomic centers and basis function exponents.

Construction of Coulomb Operator. Our focus here will be on generation of the Coulomb operator given above in eq 3. The exchange operator is more involved but follows the same principles elucidated here within the scheme described previously.[18] The Coulomb operator is constructed in the space of primitive Gaussian basis functions. This is expressed as follows where the E coefficients represent appropriate combinations of Hermite expansion coefficients (as in eqs 9−10), and P is the density matrix.

$$J_{[\mu\nu]} = \sum_{pq\lambda\sigma} E_{[p]}^{[\mu\nu]}E_{[q]}^{[\lambda\sigma]}P_{[\lambda\sigma]}[p|q] \tag{14}$$

We further divide eq 14 into three stages: preprocessing, integral evaluation, and post processing, as illustrated in Figure 2. Preprocessing handles the contraction over the $\lambda\sigma$ index, to combine ket expansion coefficients with elements of the density matrix. The results are denoted as "preprocessed density matrix elements" (PDMEs), represented as D in

$$J_{[\mu\nu]} = \sum_{pq} E_{[p]}^{[\mu\nu]}D_{[q]}[p|q] \tag{15}$$

Integral evaluation then further contracts the q index with $[p|q]$ integrals generated from R_{NLMJ} coefficients to produce $J^{Hermite}$ values. Finally, the postprocessing step contracts over the bra's Hermite coefficients to transform the result back to the space of Cartesian Gaussians:

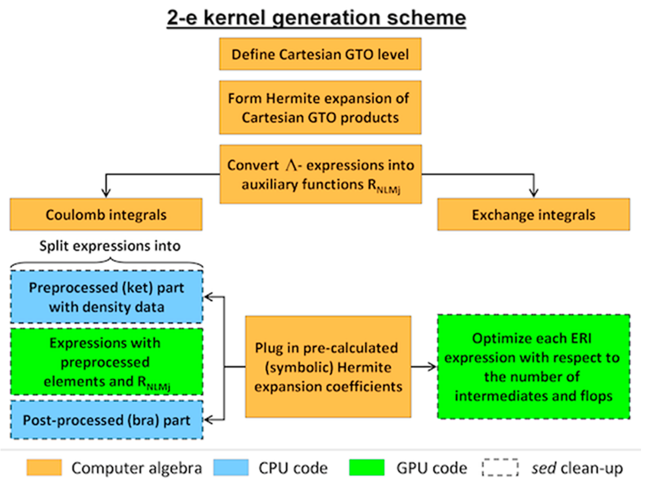

2-e kernel generation scheme
Define Cartesian GTO level
Form Hermite expansion of Cartesian GTO products
Convert Λ- expressions into auxiliary functions R_NLMJ
Coulomb integrals
Exchange integrals
Split expressions into
Preprocessed (ket) part with density data
Expressions with preprocessed elements and R_NLMJ
Post-processed (bra) part
Plug in pre-calculated (symbolic) Hermite expansion coefficients
Optimize each ERI expression with respect to the number of intermediates and flops
Computer algebra
CPU code
GPU code
sed clean-up

Figure 2. Algorithm diagram for generating symbolic ERI expressions to use with McMurchie-Davidson scheme. Orange blocks represent general algebraic expressions subsequently divided into Coulomb and exchange integrals. Green and blue blocks represent symbolic expressions converted to GPU and CPU C codes, respectively. Derivations of Coulomb integrals for different sets of basis functions follow the Ahmadi and Almlof scheme,[40] allowing to split the work between CPU and GPU (pre- and postprocessing parts on the left).

$$J_{[\mu\nu]} = \sum_{p} E_{[p]}^{[\mu\nu]}J_{[p]}^{Hermite} \tag{16}$$

METACODING APPROACH

An automated code generation approach is practically required when dealing with large numbers of complex kernels, as in electron repulsion integral evaluation on GPUs. However, once this approach is adopted, there are further benefits because many variants can be created and then tested for performance. Furthermore, this performance testing can in principle also be automated, ameliorating much of the burden of understanding the intimate details of how the hardware functions. This becomes a compelling advantage when the underlying hardware is subject to change, as may often happen for novel architectures such as the GPU. The idea of automated scanning over code variants to determine the highest performing ones can be traced back to the ATLAS project for numerical linear algebra.[45] In the case of linear algebra, it is possible to predict the optimal code variants if the size and number of caches and their associated latencies are all known. The ATLAS idea originated not so much because the performance could not be predicted but rather because the required hardware information is often difficult to obtain, so it is simpler to just generate all possibilities and empirically test each of them. A similar performance tuning approach has recently been reported in the context of tensor algebra related to coupled cluster and other many-body methods.[46] As will be shown below, the performance of integral generation kernels is harder to predict and simple metrics like minimizing the number of FLOPs or minimizing memory usage do not appear to be sufficient.

One might expect that the optimization phase of the compiler should be able to identify the best code transformations and permutations of data accesses. Although modern compilers can improve code performance, their capabilities in this regard are limited by the need to carry out source code to machine code translation efficiently. Full optimization is surely an exponentially difficult problem and the compiler can only consider a limited window of the source code at any given time if compilation is to complete quickly. Thus, complex codes often cannot rely on compilers to get maximal performance from the hardware; this is especially true in the case of stream processors such as the GPU. The metacoding approach centers on a high-level abstract model that allows empirical exploration of different realizations and transformations of the underlying equations.

Since the core computational relations are abstracted in algebra, we can generate code variants that are *guaranteed* to be correct by the CAS engine. As a result, we obtain code variants for the same kernel based on different principles and at different optimization levels, starting from basic simplifications of expressions to computational pathway decisions about which variables should be precomputed or recalculated as needed (discussed below). To find the optimal execution pathway, it is desirable to cycle through many of these code variants. It is then straightforward to pick the best performing one (see Supporting Information for examples of code variants). The approach is flexible enough to target different processing architectures, or even different releases of compiler tools, when the performance of kernels can change unexpectedly. There are strong similarities to the automated empirical optimization of software implemented in the ATLAS project, where the source code (choice of floating point, loop unrolling, register blocking, etc.) and parametrized adaptations (e.g., blocking size for linear algebra routines) are tested empirically for the best performance.[45] The main difference in our approach is in the depth of transformations applied to the code: the complexity of the integrals allows a wide variety of mathematically correct solution pathways. Similar to ATLAS, we generate many implementations and choose the best performing one.

We give a simple example of the type of code transformation we are discussing in Figure 3. The task that is detailed here is the calculation of a small set of the Hermite integrals. Three different pathways are shown, each starting with the values of two auxiliary functions (R_{0002} and R_{0003}) and three prefactors (a, b, and c) which depend on the location of the basis function centers and the corresponding exponents. The three depicted pathways have distinct memory requirements and floating-point operation counts. This is a highly simplified example, generating only three final integrals. A realistic case, for example, generation of [dd|dd] integrals, would generate as many as 1296 final integrals and thus would involve many starting values (auxiliary functions and prefactors in the recursion relations) and far more ways of reusing intermediates. Visualizing all of these pathways is likely to be nearly impossible. In fact, we go beyond the construction of the integrals (recurrence relations embodied by eq 13) within the CAS and incorporate the entire formation of the Coulomb and exchange matrices, including the construction of the Hermite-to-Cartesian transformation coefficients (eqs 9-10), the calculation of the PDMEs, and the transformation of the Hermite Coulomb and/or exchange matrix back to the Cartesian Gaussian basis (eqs 14–16). The details are tedious and are omitted here, since the basic concept is already apparent with the simple example of Figure 3.

Even though the example in Figure 3 encompasses only a handful of integrals, it already shows that the implementations can be structured very differently with ratios of explicit intermediates (temporary memory locations) to floating point multiplications ranging from 0 to 0.5. It is hard to know which of these pathways is preferred without very detailed knowledge of the underlying hardware *and* the optimization algorithms of the compiler. By empirically testing the performance of the different variants, we avoid this difficulty.

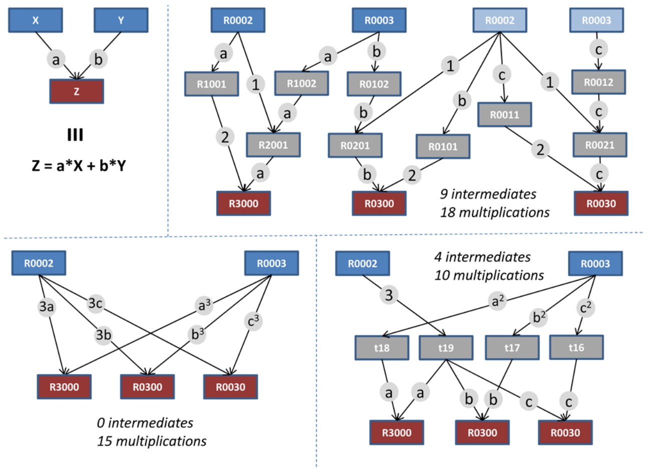


Figure 3. Examples of pathways that can be generated with CAS for the calculation of R_{3000}, R_{0300}, and R_{0030} Hermite integrals from auxiliary functions R_{0002} and R_{0003}. (Upper Left) A sample pathway with inputs X, Y and output Z, defining the meaning of the diagrams. Variables at arrow bases are multiplied by symbols inside gray circles. Inputs/outputs/intermediates are in red/blue/gray rectangles, respectively. (Upper Right) The classic McMurchie-Davidson recursion path. The hatched inputs are duplicated for clarity and do not contribute to the memory requirements. (Lower Left) Calculation of the final integrals without intermediates. (Lower Right) Calculation of the final integrals with shared intermediates. The number of multiplications and the number of intermediates required are shown for each example.

We apply transformations to the algebraic equations within the CAS using a number of simplification and transformation facilities. As we use the Maple CAS, we give the particular Maple commands used in italics below. The principles would nevertheless be equally applicable to other CAS programs. Once the algebraic equations are set up in CAS (these embody the basic recursions and prefactor manipulations that were described above in the overview of the McMurchie-Davidson integral generation scheme and its implementation), we transform the equations using combinations of the following procedures:

- Simplify algebraic expressions with *simplify*.
- Convert multivariate polynomials to Horner form (note that the choice and order of principal variables matters, and one can cycle through the various options here): $a_0 + x(a_1 + x(a_2 + ...+x(a_{n-1} + a_n x...))$ with *convert(.., 'horner', ...)*.
- Factor out and eliminate common subexpressions with *collect(..,'factor')*.
- Divide expressions into groups of intermediates with *algsubs* and *subs*.

Once the algebraic equations have been transformed analytically, optimized code is generated with Maple's built-in code optimizer, for example, *codegen[C]*. By choosing different combinations and/or variants of the above procedures, we can generate a number of different concrete realizations of a kernel. These still need to be processed to "glue" them into the main code, and this is accomplished with scripts as sketched in Figure 2.

As an example, consider a kernel that calculates a [pp|pp] batch of exchange integrals. This kernel is quite complex (typically 400+ lines with at least 1000 arithmetic instructions in its innermost loop), and its performance almost entirely relies on the prudent use of registers and arithmetic instructions. In Table 1, we show resource requirements (FLOPs and MOPs) and empirical timings for various [pp|pp] kernels that were generated within our framework. The computations are organized differently in these implementations, but the parameter space (i.e., variables with values known at runtime) stays the same. Given the predefined parameter space of R_{000j}, pair quantities, expansion coefficients, and density, our CAS framework produces correct computational pathways to Fock matrix updates. Some of these code variants compute the auxiliary functions R_{NLM} as needed and others precompute these from R_{000j} and store the precomputed values. Some of them transform the expressions to eliminate many common subexpressions and thus minimize floating point operations and others recompute many of the intermediate quantities. Three of these code fragments are shown in the Supporting Information to help clarify the differences between code variants generated through our procedure. All of these code variants were optimized for the shortest execution time (according to a simple heuristic model based on the number of FLOPs) by Maple's *codegen* facility. As can be seen in Table 1, the performance of these code variants varies greatly, with very steep performance penalties arising from the lack of registers for intermediates or suboptimal packing of arithmetic expressions. Not surprisingly, the fastest kernels in our sample distribution are found when the register usage and the number of arithmetic instructions turn out to be the smallest as seen for code variant 1 in Table 1. However, this dependence is not completely predictable. For example, the trade-off between an excessive number of FLOPs and register usage can be seen in the data sets corresponding to code variants 4 and 7. Here, more FLOPs with a smaller number of registers is more efficient than fewer FLOPs with more registers. For kernels with approximately the same number and type of arithmetic instructions, the difference in performance may be a result of instruction-level parallelism,[47] where dependencies in the calculated expressions determine different pipelining of arithmetic instructions (see Supporting Information, Figures S1 and S2). This strongly depends on the details of the hardware and the sophistication of the compiler and is hardly predictable.

■ ACCURACY AND PERFORMANCE

Calculation Correctness and Mixed Precision. Precision concerns have been thoroughly discussed previously and are no longer an issue, as GPUs now support robust, IEEE-compliant double precision (DP) arithmetic.[14−16,18] To maximize GPU performance, we continue to use a mix of single and double precision operations. Integrals with large Schwarz bounds are evaluated in double precision kernels while the rest are computed in single precision (with double precision accumulation into Fock matrix elements). This strategy requires separate single and double precision kernels, but provides nearly twice the performance and similar results as a full double-precision implementation.[14,18,34,48,49]

Table 2 summarizes the errors in dynamic precision total energies calculated with the HF method using the 6-31G* basis set with the integral single/double precision threshold dynamically changing from 1.0 to 1.0 × 10^{-5} atomic units (integrals with Schwartz bounds smaller than this threshold are computed in single precision and all others are computed in double precision), realized with our previously described

Table 2. Accuracy of the Direct-SCF Algorithm with Restricted Hartree-Fock Method Using 6-31G* Basis Set[a]

molecule	N_{atoms}; N_{bfs}	E_{TeraChem}, au	E_{GAMESS}, au	$\lvert\Delta E\rvert$, kcal/mol
caffeine	24; 230	−676.3387839	−676.3387824	0.0001
cholesterol	74; 512	−1124.0525208	−1124.0525198	0.0006
C_{60}	60; 900	−2271.7146008	−2271.7145719	0.0184
taxol	110; 1013	−2909.7293930	−2909.7293917	0.0009
valinomycin	168; 1350	−3771.1061020	−3771.1060932	0.0055
CLN025	163; 1522	−4410.4242761	−4410.4242687	0.0047
olestra	453; 3181	−7491.1432555	−7491.1432469	0.0054

[a]Reference calculations are performed with full 64-bit DP accuracy, while TeraChem calculations are done using dynamic precision.[48] N_{atoms} and N_{bfs} refer to the number of atoms and number of basis functions, respectively.

dynamic thresholding scheme.[48] This precision scheme allows us to obtain double precision quality results at nearly the cost of single precision calculations. We compare the fully converged final energies with reference calculations that utilize the General Atomic and Molecular Electronic Structure System (GAMESS).[38] Benchmarks are performed on a variety of systems starting from small molecules such as caffeine to large biomolecules such as a small protein (CLN025) and the olestra molecule. The obtained absolute energy differences shown in Table 2 are small and always well below the "chemical accuracy" threshold, that is, approximately 1 kcal/mol.

To maintain a manageable amount of generated source code, our CAS based metaprogram generates a single CUDA template for each angular momentum class. This is then instantiated into either single or double precision kernel variants as needed. We also leverage a similar template scheme to support long- and short-range Coulomb integrals used in modern range-corrected density functionals.[50−54] The required electron repulsion integrals differ from the usual full-range Coulomb integrals by a small adjustment of the function F_0,[55] which is easily encoded as an additional template parameter. Thus, each CAS optimized integral template is automatically expanded and further optimized by the CUDA compiler into as many as six custom kernels.

Integral Generation Performance. Since the case of low angular momentum kernels has been carefully addressed before,[56] we focus on the example of a computationally intensive kernel dealing with the [dd|dd] integral batch. The number of operations in ERI batches grows quickly with the angular momentum. While there are only 30 and 387 FLOPs in [ss|ss] and [pp|pp] kernels, respectively,[18] the generated [dd|dd] kernel has as many as 3582 FLOPs. The amount of memory (ideally in local registers) needed to process [ss|ss] and [pp|pp] kernels is 80 and 224 bytes per thread, respectively. In contrast, the amount of memory necessary to accommodate all the intermediates in the [dd|dd] kernel surges up to 1892 bytes per thread. Clearly, the lower angular momentum kernels fit into the small amount of very fast memory (4-byte registers) currently available on GPU hardware: 64 registers (256 bytes) and 128 registers (512 bytes) for Compute Capability (CC) 2.0 (Fermi) and CC 1.3 (GT200) architectures, respectively.[57] Although the amount of fast memory available per thread might be expected to increase in future generations, it will always be limited because the speed of stream processing largely derives from its reliance on many small sets of registers. Therefore, it is imperative to reduce the amount of memory used in kernels, so that the rapidly executed FLOPs do not suffer from high memory latency.

In Figure 4 we show the performance of the generated J [dd|dd] kernels. The performance of two different kernel types is depicted: for GPU and for CPU. To assess true speedups of the GPU-accelerated parts, we developed our own CPU-based integral code. The CPU code was backported from GPU kernels with the structure of the code mostly preserved. The backported CPU code was also further optimized with data remapping and regrouping of intermediates for multithreaded CPU execution using Streaming SIMD Extensions (SSE). The sustained performance of these CPU kernels is 1.25 and 13.3 GFLOP/sec on the same CPU for unoptimized and optimized code, respectively (compare with 0.79 GFLOP/sec reported[16] for GAMESS). Further manual tuning[58] resulted in only minor improvements of approximately 20%, suggesting that the generated code is highly efficient.

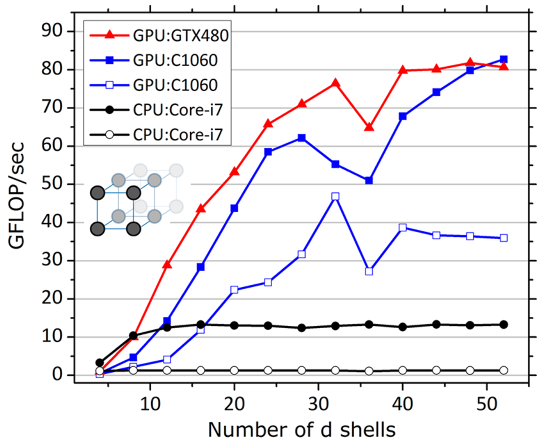


Figure 4. Performance of various [dd|dd] kernel implementations. Benchmarked molecule has one d shell on each atom and the atoms are organized in a regular 3D lattice (as shown in the inset). Code running on a single GTX 480 GPU (CC 2.0), a single Tesla C1060 (CC 1.3) and Intel's Core-i7 @ 3.33 GHz CPU is shown with triangles, squares, and circles, respectively. Optimized and unoptimized kernels both for GPU and for CPU are depicted with filled and empty symbols, respectively. One GFLOP/sec equals to 10^9 floating point operations per second.

As shown in Figure 4, the performance of the initial (unoptimized) GPU code is several times better than the heavily optimized CPU code. However, the large memory footprint carried by each GPU thread inhibits the effectiveness of the code. Forcing the GPU compiler to evaluate intermediates in-place[59] as needed (instead of inlining them many times in later expressions) reduces the local memory requirements and leads to considerable performance improvement. Specifically, for [dd|dd] kernels, this optimization reduces the memory requirements by almost 30% (from 1892 to 1340 bytes per thread) and more than doubles the performance of the GPU kernel as shown in Figure 4. The optimized kernel performs similarly or slightly better on CC 2.0 hardware than on CC 1.3, without any modifications to the code, as shown for the [dd|dd] kernel, because of the higher memory bandwidth and a faster GPU clock speed.

SCF Performance. Figure 5 summarizes the relative speedups obtained for HF calculations through the use of GPU-accelerated integrals. These values refer to the first iteration of the SCF procedure. The GAMESS code is used for the reference calculations on a state-of-the-art CPU.[38] While the multi-GPU version can be readily employed with up to 8 GPUs in parallel and with nearly 100% scaling efficiency across multiple GPUs, we use only one GPU for this benchmark. The geometries for the benchmarked molecules are given in the Supporting Information, so the interested reader can produce timings for other codes and architectures, if further comparisons are desired. We use the 6-31G* basis set and HF calculations for these benchmarks. The detailed timings of each step in the SCF procedure are given in the Table 3. As seen in Figure 5, our GPU-based code is only somewhat faster with smaller molecules (only 10−20×), since the amount of computation is relatively small. The advantage of the inherent massive parallelism on the GPU is seen more clearly when the device is heavily loaded, that is, with large biomolecular systems where the speedups can exceed 100×. Ultimately, the application to very large biomolecular systems (tens of thousands of basis functions) may be limited by GPU memory.

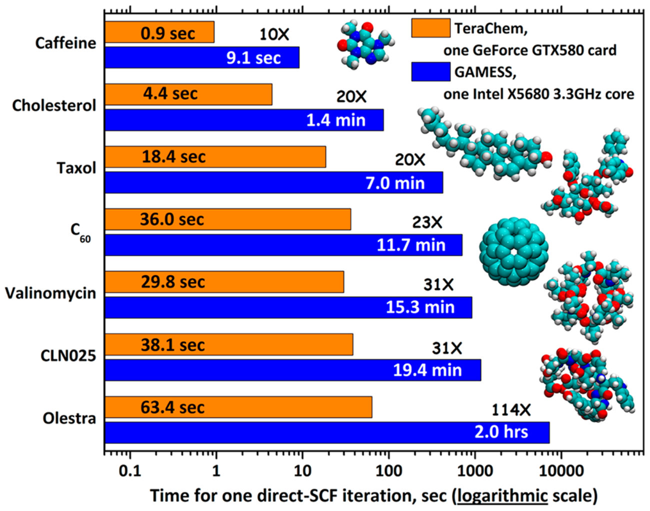


Figure 5. Comparison of times for a single (first) direct-SCF iteration using TeraChem (dynamic precision RHF/6-31G*, as in Table 2) on a single GeForce GTX580 GPU, as compared to GAMESS on the state-of-the-art Intel Westmere 3.33 GHz CPU core.

Table 3. Timing Data (in seconds) for Different Steps of the SCF Procedure (the First Iteration, RHF, 6-31G* Basis set)[a]

	Fock matrix				
molecule	pre-calculation data management	J	K	linear algebra	SCF time
caffeine	0.08	0.10	0.73	0.03	0.94
cholesterol	0.11	0.46	3.72	0.12	4.41
C_{60}	0.26	1.54	16.12	0.45	18.37
taxol	0.21	2.11	33.37	0.34	36.03
valinomycin	0.42	2.80	25.76	0.77	29.75
CLN025	0.55	3.31	33.27	1.01	38.14
olestra	3.61	9.30	43.54	6.93	63.38

[a]The Fock matrix formation in the SCF procedure consists of three steps: presorting and precalculation of pair-quantities (done on CPU), and the J- and K-matrix formation steps (done on GPU). The timing data were obtained on a dual 3.4 GHz Intel Xeon X5690 hexacore CPU and a GeForce GTX580 GPU.

However, we expect such limitations will be lifted as GPUs are available with increasing amounts of memory.

CONCLUSIONS

We have developed an automated code generation approach to integral evaluation on stream processing architectures, including generation of many code variants for the same tasks and selection between them through empirical testing. We have applied the approach to both GPU and CPU architectures. Using this approach, we were able to extend our previous implementations of electron repulsion integrals on GPUs from s and p angular momentum Gaussian basis functions to also include d angular momenta, which are critical for applications involving transition metal atoms, but also for accurate thermochemistry of molecules containing first-row elements. In this case, the use of a meta-programming strategy was mandated by the considerable number and high complexity of GPU kernels for integral evaluation. For example, while s and p functions require only 6 kernels, extension to d functions requires 36 kernels. Furthermore, these kernels are more complex: while the [pp|pp] Coulomb integral kernel takes only 306 lines of CUDA source code, the [dd|dd] Coulomb kernel requires more than 2000 lines.

The basic ideas of metaprogramming and generation of many code variants selected by empirical performance testing can be expected to become more important and more widespread in the future. This approach provides both the ability to separate the derivation of correct mathematical expressions from the details of the implementation and also a robustness of performance with respect to different hardware without requiring development for each new hardware variant. The results we report here lay the foundation for analytic derivatives involving d functions and the coding of higher angular momentum functions (such as f functions) in an efficient manner on the GPU and similar architectures.

ASSOCIATED CONTENT

Supporting Information

Example code fragments demonstrating different variants that can be autogenerated by the CAS. Representative syntax trees for the autogenerated kernels. Cartesian coordinates for the molecules used as benchmarks in Figure 5. This material is available free of charge via the Internet at http://pubs.acs.org.

AUTHOR INFORMATION

Corresponding Author

*E-mail: todd.martinez@stanford.edu.

Notes

The authors declare the following competing financial interest(s): Two of the authors (I.S.U. and T.J.M.) are founders of PetaChem, LLC.

ACKNOWLEDGMENTS

A.V.T. is grateful to Thom H. Dunning, Jr. and Guochun Shi for insightful discussions. We thank Volodymyr Kindratenko for insights into the optimization of kernels ported to the CPU. This work was supported by the NSF Grant CHE-0626354 and utilized the AC cluster[60] operated by the Innovative Systems Laboratory (ISL) at the National Center for Supercomputing Applications (NCSA) at the University of Illinois.

REFERENCES

(1) Barnett, M. P.; Capitani, J. F.; von zur Gathen, J.; Gerhard, J. *Int. J. Quantum Chem.* **2004**, *100*, 80.
(2) Jones, H. W. *Int. J. Quantum Chem.* **1980**, *18*, 709.
(3) Maple 16 by Maplesoft, www.maplesoft.com
(4) Wolfram Mathematica 8 by Wolfram Research, www.wolfram.com
(5) Gomez, C.; Scott, T. *Comput. Phys. Commun.* **1998**, *115*, 548.
(6) Scott, T. C.; Monagan, M. B.; Grant, I. P.; Saunders, V. R. *Nucl. Inst. Meth. Phys. A* **1997**, *389*, 117.
(7) Bracken, P.; Bartlett, R. J. *Int. J. Quantum Chem.* **1997**, *62*, 557.
(8) Fermann, J. T.; Valeev, E. F.; http://www.ccmst.gatech.edu/~evaleev/libint/: 2003.
(9) Strange, R.; Manby, F. R.; Knowles, P. J. *Comput. Phys. Commun.* **2001**, *136*, 310.
(10) Salek, P.; Hesselmann, A. *J. Comput. Chem.* **2007**, *28*, 2569.
(11) Hartono, A.; Lu, Q.; Henretty, T.; Krishnamoorthy, S.; Zhang, H.; Baumgartner, G.; Bernholdt, D. E.; Nooijen, M.; Pitzer, R. M. *J. Phys. Chem. A* **2009**, *113*, 12715.
(12) Auer, A.; Baumgartner, G.; Bernholdt, D. E.; Bibireata, A.; Choppella, V.; Cociorva, D.; Gao, X.; Harrison, R. J.; Hartono, A.; Krishnamoorthy, S.; Krishnan, S.; Lam, C.; Lu, Q.; Nooijen, M.; Pitzer, R. M.; Ramanujam, J.; Sadayappan, P.; Sibiryakov, A. *Mol. Phys.* **2006**, *104*, 211.
(13) Stone, J. E.; Hardy, D. J.; Ufimtsev, I. S.; Schulten, K. *J. Mol. Graphics Modell.* **2010**, *29*, 116.
(14) Ufimtsev, I. S.; Martinez, T. J. *J. Chem. Theory Comput.* **2008**, *4*, 222.
(15) Yasuda, K. *J. Comput. Chem.* **2008**, *29*, 334.
(16) Asadchev, A.; Allada, V.; Felder, J.; Bode, B. M.; Gordon, M. S.; Windus, T. L. *J. Chem. Theory Comput.* **2010**, *6*, 696.
(17) Titov, A. V.; Kindratenko, V. V.; Ufimtsev, I. S.; Martinez, T. J. In *Proceedings of the Symposium on Application Accelerators in High-Performance Computing (SAAHPC)*, University of Tennessee, Knoxville, TN, July 13−15, 2010; National Center for Supercomputing Applications, University of Illinois at Urbana-Champaign: Champaign, IL, 2010; http://saahpc.ncsa.illinois.edu/10/papers/paper_53.pdf.
(18) Ufimtsev, I. S.; Martinez, T. J. *J. Chem. Theory Comput.* **2009**, *5*, 1004.
(19) Yasuda, K. *J. Chem. Theory Comput.* **2008**, *4*, 1230.
(20) Genovese, L.; Ospici, M.; Deutsch, T.; Mehaut, J.-F.; Neelov, A.; Goedecker, S. *J. Chem. Phys.* **2009**, *131*, 034103.
(21) Brown, P.; Woods, C. J.; McIntosh-Smith, S.; Manby, F. R. *J. Comput. Chem.* **2010**, *31*, 2008.
(22) Vogt, L.; Olivares-Amaya, R.; Kermes, S.; Shao, Y.; Amador-Bedolla, C.; Aspuru-Guzik, A. *J. Phys. Chem. A* **2008**, *112*, 2049.
(23) Olivares-Amaya, R.; Watson, M. A.; Edgar, R. G.; Vogt, L.; Shao, Y.; Aspuru-Guzik, A. *J. Chem. Theory Comput.* **2010**, *6*, 135.
(24) Anderson, A. G.; Goddard, W. A., III; Schroeder, P. *Comput. Phys. Commun.* **2007**, *177*, 298.
(25) Esler, K.; Kim, J.; Shulenburger, L.; Ceperley, D. *Comp. Sci. Eng.* **2010**, *14*, 40.
(26) Vahtras, O.; Almlof, J.; Feyereisen, M. W. *Chem. Phys. Lett.* **1993**, *213*, 514.
(27) Feyereisen, M. W.; Fitzgerald, G.; Komornicki, A. *Chem. Phys. Lett.* **1993**, *208*, 359.
(28) Werner, H. J.; Manby, F. R.; Knowles, P. J. *J. Chem. Phys.* **2003**, *118*, 8149.
(29) Friesner, R. A. *Annu. Rev. Phys. Chem.* **1991**, *42*, 341.
(30) Martinez, T. J.; Carter, E. A. In *Modern Electronic Structure Theory*; Yarkony, D. R., Ed.; World Scientific: Singapore, 1995.
(31) Beebe, N. H. F.; Linderberg, J. *Int. J. Quantum Chem.* **1977**, *12*, 683.
(32) Chwee, T. S.; Carter, E. A. *J. Chem. Phys.* **2010**, *132*, 074104.
(33) Aquilante, F.; Gagliardi, L.; Pedersen, T. B.; Lindh, R. *J. Chem. Phys.* **2009**, *130*, 154107.
(34) Ufimtsev, I. S.; Martinez, T. J. *J. Chem. Theory Comput.* **2009**, *5*, 2619.
(35) Rys, J.; Dupuis, M.; King, H. F. *J. Comput. Chem.* **1983**, *4*, 154.

(36) Kapasi, U.; Rixner, S.; Dally, W.; Khailany, B.; Ahn, J. H.; Mattson, P.; Owens, J. D. *Computer* **2003**, *36*, 54.
(37) Rul, S.; Vandierendonck, H.; D'Haene, J.; De Bosschere, K. In *Proceedings of the Symposium on Application Accelerators in High-Performance Computing (SAAHPC)*, University of Tennessee, Knoxville, TN, July 13−15, 2010; National Center for Supercomputing Applications, University of Illinois at Urbana-Champaign: Champaign, IL, 2010.
(38) Schmidt, M. W.; Baldridge, K. K.; Boatz, J. A.; Elbert, S. T.; Gordon, M. S.; Jensen, J. H.; Koseki, S.; Matsunaga, N.; Nguyen, K. A.; Su, S. J.; Windus, T. L.; Dupuis, M.; Montgomery, J. A. *J. Comput. Chem.* **1993**, *14*, 1347.
(39) McMurchie, L. E.; Davidson, E. R. *J. Comput. Phys.* **1978**, *26*, 218.
(40) Ahmadi, G. R.; Almlof, J. *Chem. Phys. Lett.* **1995**, *246*, 364.
(41) White, C. A.; Head-Gordon, M. *J. Chem. Phys.* **1996**, *104*, 2620.
(42) Ryu, U.; Lee, Y. S.; Lindh, R. *Chem. Phys. Lett.* **1991**, *185*, 562.
(43) Head-Gordon, M.; Pople, J. A. *J. Chem. Phys.* **1988**, *89*, 5777.
(44) Gill, P. M. W.; Pople, J. A. *Int. J. Quantum Chem.* **1991**, *40*, 753.
(45) Whaley, R. C.; Petitet, A.; Dongarra, J. J. *Parallel Comput.* **2001**, *27*, 3.
(46) Lu, Q.; Gao, X.; Krishnamoorthy, S.; Baumgartner, G.; Ramanujam, J.; Sadayappan, P. *J. Parallel Dist. Comput.* **2012**, *72*, 338.
(47) Volkov, V. In *Proceedings of the GPU Technology Conference*, San Jose, CA, Sept 23, 2010; NVIDIA Corporation: Santa Clara, CA, 2010.
(48) Luehr, N.; Ufimtsev, I. S.; Martinez, T. J. *J. Chem. Theory Comput.* **2011**, *7*, 949.
(49) Knizia, G.; Li, W. B.; Simon, S.; Werner, H. J. *J. Chem. Theory Comput.* **2011**, *7*, 2387.
(50) Baer, R.; Livshits, E.; Salzner, U. *Annu. Rev. Phys. Chem.* **2010**, *61*, 85.
(51) Rohrdanz, M. A.; Martins, K. M.; M., H. J. *J. Chem. Phys.* **2009**, *130*, 054112.
(52) Tawada, Y.; Tsuneda, T.; Yanagisawa, S.; Yanai, T.; Hirao, K. *J. Chem. Phys.* **2004**, *120*, 8425.
(53) Toulouse, J.; Colonna, F.; Savin, A. *Phys. Rev. A* **2004**, *70*, 062505.
(54) Heyd, J.; Scuseria, G. E.; Ernzerhof, M. *J. Chem. Phys.* **2003**, *118*, 8207.
(55) Adamson, R. D.; Dombroski, J. P.; Gill, P. M. W. *J. Comput. Chem.* **1999**, *20*, 921.
(56) Ufimtsev, I. S.; Martinez, T. J. *Comp. in Sci. Eng.* **2008**, *10*, 26.
(57) *NVIDIA CUDA. Compute Unified Device Architecture Programming Guide Version 3.1; NVIDIA Developer Web Site.*
(58) Ye, D.; Titov, A. V.; Kindratenko, V. V.; Ufimtsev, I. S.; Martinez, T. J. In *Proceedings of the Symposium on Application Accelerators in High-Performance Computing (SAAHPC)*, University of Tennessee, Knoxville, TN, July 19−21, 2011; National Center for Supercomputing Applications, University of Illinois at Urbana-Champaign: Champaign, IL, 2011; p 72.
(59) In the release of CUDA we use here, this is possible with the volatile keyword which has the side effect of forcing in-place evaluation of intermediates with no persistent storage.
(60) Kindratenko, V. V.; Enos, J. J.; Shi, G.; Showerman, M. T.; Arnold, G. W.; Stone, J. E.; Phillips, J. C.; Hwu, W. In *Proceedings of the IEEE International Conference on Cluster Computing and Workshops*; IEEE Computer Society Press, John Wiley & Sons: New York, 2009; DOI: 10.1109/CLUSTR.2009.5289128.