Automated Code Engine for Graphical Processing Units: Application to the Effective Core Potential Integrals and Gradients

Chenchen Song[1,2], Lee-Ping Wang[1,2], and Todd J. Martínez[1,2]

[1]*Department of Chemistry and the PULSE Institute,*

Stanford University, Stanford, CA 94305

[2]*SLAC National Accelerator Laboratory, Menlo Park, CA 94025*

Abstract

We present an automated code engine (ACE) that automatically generates optimized kernels for computing integrals in electronic structure theory on a given graphical processing unit (GPU) computing platform. The code generator in ACE creates multiple code variants with different memory and floating point operation trade-offs. A graph representation is created as the foundation of the code generation, which allows the code generator to be extended to various types of integrals. The code optimizer in ACE determines the optimal code variant and GPU configurations for a given GPU computing platform by scanning over all possible code candidates, and then choosing the best-performing code candidate for each kernel. We apply ACE to the optimization of effective core potential integrals and gradients. It is observed that the best code candidate varies with differing angular momentum, floating point precision, and type of GPU being used, which shows that the ACE may be a powerful tool in adapting to fast evolving GPU architectures.

1. Introduction

Many well-established approaches in modern quantum chemistry rely on introducing a set of one-electron orbitals to describe the many-body wavefunction. The *ab initio* calculations of the electronic energy, atomic forces, and other observables naturally involve integrals of the atomic basis functions with operators such as electron-nuclear attraction and electron-electron repulsion,[1] dipole moment, and spin-orbit coupling,[2] along with their corresponding derivatives.[3] Consequently, the computational cost of many quantum chemistry methods is dominated by large numbers of integral and gradient evaluations. More recently, graphical processing units (GPUs) have become a powerful resource in accelerating integral construction.[4,5,6,7] However, due to significant differences in the architecture between GPUs and traditional central processing units (CPU), several important programming challenges must be addressed in order to efficiently utilize the computational power of the GPU.

In terms of evaluating integrals in electronic structure theory, there are at least three important challenges. First of all, the streaming multiprocessor (SM) in GPUs derives its computational power from executing kernels on thousands of threads in parallel, but each thread only has rapid access to a relatively small amount of data compared to programs running on the CPU. GPUs have a complex memory hierarchy; in order of increasing latency (the time needed to access a variable), memory is composed of 1) thread-specific registers, 2) L1 cache / shared memory (shared among a group of threads called a *block*), and 3) L2 cache / global memory (accessible by all threads). In contrast with a typical CPU, where the cache size per thread is usually several megabytes, the cache size per warp (a group of 32 threads comprising the minimum execution unit) on the GPUs is only several kilobytes. The properties of small cache, high latency, limited bandwidth and potential bank conflicts all make global memory (DRAM) usage expensive. Broadly speaking, performance is improved by writing GPU code that efficiently uses the registers and shared memory while minimizing the use of global memory, such that cores stay "busy" with computations rather than "waiting" to read a variable from higher-latency memory.

Second, the architecture of graphical processing units evolves quickly. NVIDIA has announced its Fermi architecture, Kepler architecture and Maxwell architecture within only six years since 2009. These architectures all have different features from each other. For example, a typical graphics card with Fermi architecture is composed of 16 SMs, each with 32 cores. The

Kepler architecture reduces the number of SMs of each card, but increases the cores per SM to 192 cores. The Maxwell architecture has 128 cores on each SM that are partitioned into 4 processing blocks, each with its own resources for scheduling and instruction buffering. More detailed comparisons between architectures are given in Section 1 of the Supporting Information. Due to the diversity of GPU architectures, computing kernels optimized for one platform are not guaranteed to perform well on another.

Third, the complexity of kernels in both operations and memory requirements increases with respect to the angular momentum of the basis set.[8-9] As a result, optimizing the kernels becomes increasingly difficult with higher angular momentum. This is also a challenge in carrying out integrals and gradients on the CPU, but more onerous in the case of the GPU due to the rapid introduction of new architectures and the limited memory constraints.

The problem could be solved if there existed a fully optimizing compiler capable of transforming each program *P* into the optimum program *Opt*(*P*) with the same input/output behavior as *P*. However, the full employment theorem in computer science has proved that such an ideal compiler cannot exist.[10] In addition, the range of transformations that a compiler can explore is usually limited such that the program can be compiled in a reasonable amount of time; in a qualitative sense, the compiler performs only a very local optimization in the full space of programs. As a result, we cannot completely rely on a general purpose compiler to search for better program transformations. An alternative approach is to develop a code generator which can use knowledge of the mathematics behind integral methods (which is not available to a compiler) to create a wide variety of code variants. We expect that the resulting code variants will more broadly explore the space of possible program implementations and thus be more likely to expose the best performance on the hardware.

The complexity inherent in *ab initio* quantum chemistry and many-body theories has long motivated attempts to use automated code generation. The earliest attempt dates back to the pioneering work of Jones,[11-12] automating the generation of overlap integrals between Slater-type orbitals by using a computer algebra system to transform mathematical expressions into computer code. Subsequent studies have applied this method to generate code for Gaussian integrals,[13] density functional theory[14] and many-body theories.[15-16] Very recently, MacLeod et.al. developed an automatic code generator which enables analytical nuclear gradients implementations for fully internally contracted active space second-order perturbation theory

(CASPT2).[17] These studies focused on the use of code generators to guarantee code correctness while avoiding intensive (and often error-prone) manual derivations.

There has also been previous work aimed at using code generators to improve implementation performance in addition to ensuring code correctness. One approach to automated code generation with performance optimization is *model-driven optimization*. An example of this approach is the tensor contraction engine (TCE),[18] which has achieved great success in CPU-based computational many-body theory. TCE includes an operator contraction engine (OCE) which transforms Feynman-like diagrams into tensor expressions expressed in a domain-specific language. The TCE, which is the "compiler" of the domain-specific language, then translates the high-level symbolic math language into low-level languages like FORTRAN and C. The model-driven search-based optimization approach adopted by TCE relies on cost models to estimate and minimize the computational cost.[19] Very recently, researchers have been working on generalizing the TCE optimization approach such that tensor computation optimization can be extended to other architectures like GPUs.[20] Another example is LIBINT,[21-22] which uses an optimizing compiler to automatically generate two-body integrals over Gaussian functions. The optimizing compiler approach enables easy implementation of new recurrence relationships, and yields high performance for the generated code on superscalar CPU architectures.

Empirical performance-driven optimization is another approach for automatic generation of optimized code. Examples include the automatically tuned linear algebra library[23] (ATLAS) and the Fastest Fourier Transform in the West (FFTW).[24] Both numerical libraries perform program optimization by empirically testing the performance of several versions of generated code on the target architecture. The optimization approach is based on the idea that the accurate prediction of code performance requires good knowledge of the hardware layout and parameters, which can be highly complex and vary greatly across different types of target architectures. In addition, different code variants will trigger different (and often unpredictable) optimization paths in the chosen compiler, which decreases the accuracy of model-based approaches. ATLAS and FFTW address the problem by generating a set of code variants that cover a wide range of possibilities, and selecting the best code variant based on timing tests. ATLAS is reported to match or exceed the performance of the vendor-supplied version of matrix multiplication on almost every tested platform.[25] A similar empirical performance-driven approach has been developed and applied to problems in tensor algebra.[26]

The performance-driven optimization strategy is especially appealing for quantum chemistry computations on GPUs, because the high complexity of both the algorithms and the hardware architecture make it difficult to accurately model performance. Titov *et. al.* took first steps towards applying this approach to optimize Fock matrix construction on the GPU with *d*-orbitals,[27] and were able to achieve similar performance as hand-tuned GPU kernels limited to *s* and *p* orbitals. However, this work was only partially automated, hampering extensions to higher angular momentum functions or different integral types.

In this work we describe the development of a fully automated code engine (ACE) that generates optimized kernels of integral calculations for a given CUDA computing platform. In order to highlight the fundamental issues, we have first focused on the most complex one-electron integral (integrals over effective core potentials or ECPs), although generalization to other integrals is also possible. In order for the code generator to be easily generalizable, we introduce a graph representation of the procedure for evaluating the integral, which enables easy searching over different code variants.

The structure of this paper is as follows. First, we describe the design of the three modules in ACE, namely the code generator, the code tester and the code optimizer. We then describe in detail the graph representation, which is the foundation of our automated code generation. Next, we apply ACE to ECP integrals and gradients. We show that the angular momentum, floating point precision, and the type of GPU architecture all affect the optimal choices. We observe significant differences in performance across different compiler-optimized code variants. This indicates that compiler optimization is far from complete and the generation and testing of codes using ACE is a way to achieve high performance for quantum chemistry calculation kernels on the GPU. Finally, we discuss improvements to be carried out in future work.

2. Methods

The most fundamental concept in ACE is the representation of the program that evaluates an integral as a graph that connects input and output variables through intermediate values and mathematical operations. Given this graph representation, ACE automatically generates multiple computational kernels by transforming the graph structure and designating variables as being stored or recomputed on-the-fly. In the current manuscript, we provide an initial graph representation of the integral as the starting point; future work will describe the capabilities of ACE for automatically generating graph representations starting from high-level working equations. In the next section, we introduce the equations for computing the integrals that provide the basis of the graph representation.

2.1 ECP integrals and Gradients

Here we briefly summarize the method for computing ECP integrals and gradients, following McMurchie and Davidson.[28] The form of the effective core potential operator for an ECP center[29] located at the origin is

$$U_{ECP}(r) = U_{L+1}(r) + \sum_{l=0}^{L} \sum_{m=-l}^{l} |S_{lm}\rangle \left(U_l(r) - U_{L+1}(r)\right) \langle S_{lm}| \tag{1}$$

where L is the maximum angular momentum orbital in the core, and the angular functions $S_{lm}(\theta,\varphi)$ are the normalized real spherical harmonics. The following two types of integrals appear in ECP integral evaluations $<\phi_a \,|\, U(r) \,|\, \phi_b>$

$$\chi_{ab} = \int_0^\infty r^2 U(r) \int_\Omega \phi_a(\mathbf{r}) \phi_b(\mathbf{r}) d\Omega \tag{2}$$

$$\gamma_{ab}^{l} = \int_0^\infty r^2 U(r) \int_\Omega \phi_a(\mathbf{r}) S_{lm}\, d\Omega \int_{\Omega'} \phi_b(\mathbf{r}) S_{lm}\, d\Omega' dr \tag{3}$$

where

$$U(r; d_u, n, \zeta) = d_u r^n e^{-\zeta_u r^2} \tag{4}$$

is the primitive radial Gaussian functions for the ECP potential, and

$$\phi_a(\mathbf{r}) = d_a (x - A_x)^{a_x} (y - A_y)^{a_y} (z - A_z)^{a_z} e^{-\alpha(\mathbf{r}-\mathbf{A})^2} \tag{5}$$

$$\phi_b(\mathbf{r}) = d_b (x - B_x)^{b_x} (y - B_y)^{b_y} (z - B_z)^{b_z} e^{-\beta(\mathbf{r}-\mathbf{B})^2} \tag{6}$$

are the primitive Gaussian basis functions. The contraction coefficients for the ECP potential and basis functions are given by d_u, d_a, d_b, and the exponents are given by ζ_u, η_a, η_b. Here we have employed a local coordinate system centered at the position of the ECP center for each integral. The centers of the basis functions are given by $\mathbf{A} = (A_x, A_y, A_z)$; $\mathbf{B} = (B_x, B_y, B_z)$, and the angular momenta are a_x, a_y, a_z; b_x, b_y, b_z. The integral in Eq. (3) can be evaluated as

$$\gamma^{l}_{a_xa_ya_z,b_xb_yb_z} = 16\pi^2 d_u d_a d_b \sum_{\alpha_x=0}^{a_x}\sum_{\alpha_y=0}^{a_y}\sum_{\alpha_z=0}^{a_z}\Theta^{0}_{a_xa_ya_z,\alpha_x\alpha_y\alpha_z}\left(A_x,A_y,A_z\right)$$
$$\times\sum_{\beta_x=0}^{b_x}\sum_{\beta_y=0}^{b_y}\sum_{\beta_z=0}^{b_z}\Theta^{0}_{b_xb_yb_z,\beta_x\beta_y\beta_z}\left(B_x,B_y,B_z\right) \tag{7}$$
$$\times\sum_{\lambda_1=0}^{l+\alpha}\sum_{\lambda_2=0}^{l+\beta}R_{\lambda_1,\lambda_2}\left(2+n+\alpha+\beta,\zeta,\eta_A,R_A,\eta_B,R_B\right)\Omega^{\alpha_x\alpha_y\alpha_z,\beta_x\beta_y\beta_z}_{l,\lambda_1\lambda_2}\left(\mathbf{r_A},\mathbf{r_B}\right)$$

where $a=a_x+a_y+a_z$, $b=b_x+b_y+b_z$, $\alpha=\alpha_x+\alpha_y+\alpha_z$, $\beta=\beta_x+\beta_y+\beta_z$, $R_A=\|\mathbf{A}\|$, and $R_B=\|\mathbf{B}\|$. We have defined the angular factors as

$$\Theta^{0}_{a_xa_ya_z,\alpha_x\alpha_y\alpha_z}\left(A_x,A_y,A_z\right)=\left(-1\right)^{L_a-\alpha}\binom{a_x}{\alpha_x}\binom{a_y}{\alpha_y}\binom{a_z}{\alpha_z}A_x^{a_x-\alpha_x}A_y^{a_y-\alpha_y}A_z^{a_z-\alpha_z} \tag{8}$$

$$\Omega^{\alpha_x\alpha_y\alpha_z,\beta_x\beta_y\beta_z}_{l,\lambda_1\lambda_2}\left(\mathbf{r_A},\mathbf{r_B}\right)=\sum_{m=-l}^{l}\sum_{\mu_1=-\lambda_1}^{\lambda_1}S_{\lambda_1\mu_1}\left(\theta_A,\varphi_A\right)\int S_{lm}S_{\lambda_1\mu_1}x_n^{\alpha_x}y_n^{\alpha_y}z_n^{\alpha_z}\,d\Omega$$
$$\times\sum_{\mu_2=-\lambda_2}^{\lambda_2}S_{\lambda_2\mu_2}\left(\theta_B,\varphi_B\right)\int S_{lm}S_{\lambda_2\mu_2}x_n^{\beta_x}y_n^{\beta_y}z_n^{\beta_z}\,d\Omega \tag{9}$$

and the radial function is defined as

$$R_{\lambda_1,\lambda_2}\left(N,\zeta,\alpha,R_A,\beta,R_B\right)=\int_0^\infty r^{2+N}e^{-\zeta r^2}e^{-\alpha\left(r-R_A\right)^2}e^{-\beta\left(r-R_B\right)^2}K_{\lambda_1}\left(2\alpha R_A r\right)K_{\lambda_2}\left(2\beta R_B r\right)dr \tag{10}$$

where K is the modified spherical Bessel function of the first kind weighted by an exponential factor as $K_\lambda\left(z\right)=M_\lambda\left(z\right)e^{-z}$.

The corresponding analytic gradient of the integral in Eq. (3) can be derived by making use of the property that

$$\frac{d}{dA}\left[\left(x-A\right)^a e^{-\eta_a\left(x-A\right)^2}\right]=a\left(x-A\right)^{a-1}e^{-\eta_a\left(x-A\right)^2}-2\eta_a\left(x-A\right)^{a+1}e^{-\eta_a\left(x-A\right)^2} \tag{11}$$

As a result, the analytic gradient of the integral in Eq. (3) with respect to A_x can be computed as

$$\frac{d}{dA_x}\gamma^{l}_{a_xa_ya_z,b_xb_yb_z} = 16\pi^2 d_u d_a d_b \sum_{\alpha_x=0}^{a_x+1}\sum_{\alpha_y=0}^{a_y}\sum_{\alpha_z=0}^{a_z}\Theta^{x}_{a_xa_ya_z,\alpha_x\alpha_y\alpha_z}\left(A_x,A_y,A_z\right)$$
$$\times\sum_{\beta_x=0}^{b_x}\sum_{\beta_y=0}^{b_y}\sum_{\beta_z=0}^{b_z}\Theta^{0}_{b_xb_yb_z,\beta_x\beta_y\beta_z}\left(B_x,B_y,B_z\right) \tag{12}$$
$$\times\sum_{\lambda_1=0}^{l+\alpha}\sum_{\lambda_2=0}^{l+\beta}R_{\lambda_1,\lambda_2}\left(2+n+\alpha+\beta,\zeta,\eta_A,R_A,\eta_B,R_B\right)\Omega^{\alpha_x\alpha_y\alpha_z,\beta_x\beta_y\beta_z}_{l,\lambda_1\lambda_2}\left(\mathbf{r_A},\mathbf{r_B}\right)$$

where

$$\Theta^{x}_{a_x a_y a_z,\alpha_x\alpha_y\alpha_z}\left(A_x,A_y,A_z\right)=$$
$$\left(-1\right)^{L_a+1-\alpha}\left[a_x\begin{pmatrix}a_y\\ \alpha_y\end{pmatrix}A_x^{a_x-1-\alpha_x}-2\eta_A\begin{pmatrix}a_y\\ \alpha_y\end{pmatrix}A_x^{a_x+1-\alpha_x}\right]\begin{pmatrix}a_y\\ \alpha_y\end{pmatrix}\begin{pmatrix}a_z\\ \alpha_z\end{pmatrix}A_y^{a_y-\alpha_y}A_z^{a_z-\alpha_z} \tag{13}$$

The gradients with respect to A_y, A_z, B_x, B_y, B_z are similar to Eq. (12).

In order to compute the analytic gradient of the total energy with respect to the nuclear displacements, the analytic gradient of the integrals need to be contracted with some given density matrix as

$$\Gamma^{l(\xi)}_{ab}=\sum_{\substack{a_x,a_y,a_z:\\ a_x+a_y+a_z=a}}\ \sum_{\substack{b_x,b_y,b_z:\\ b_x+b_y+b_z=b}}P_{a_xa_ya_z,b_xb_yb_z}\frac{d}{d\xi}\gamma^{l}_{a_xa_ya_z,b_xb_yb_z} \tag{14}$$

where $\Gamma^{l(\xi)}_{ab}$ is the gradient of the integrals contracted with the corresponding block of the density matrix P, and ξ represents the nuclear degree of freedom from either atom A or atom B. Throughout this paper, $\Gamma^{l(\xi)}_{ab}$ is used to illustrate computational strategies.

In practice, the first step toward calculating the ECP integrals and gradients is to compute the radial integrals R in Eq. (10), which requires the parameters of the three Gaussian functions involved (those of the basis functions and the ECP) as well as their relative displacement vectors. We previously showed how an adaptive quadrature strategy with screening could accelerate radial integral evaluations.[30] The radial integral is then contracted with several other factors in a multi-level summation to compute the integral in Eq. (7) and the gradient in Eq. (12). Therefore, we now introduce a series of intermediate variables to represent the contraction process, which will be used for later analysis.

Here we use the gradient $\Gamma^{l(\xi)}_{ab}$ in Eq. (14) as an illustration. The array of radial integrals R is indexed by λ_1, λ_2 and N, where the number of variables is determined by several conditions on the indices forced by the summation rules in Eq. (12), i.e. $\lambda_1 \le l+L_a+1$, $\lambda_2 \le l+L_b+1$, $\lambda_1+\lambda_2 \le L_a+L_b+1$ and $\max(0,\lambda_1+\lambda_2-2l) \le N \le L_a+L_b+1$. First, $R(\lambda_1,\lambda_2,N)$ is contracted with an angular factor $\Omega^{\alpha_x\alpha_y\alpha_z,\beta_x\beta_y\beta_z}_{l,\lambda_1\lambda_2}\left(\mathbf{r_A},\mathbf{r_B}\right)$ that again depends on the parameters of basis functions. This contraction eliminates the λ_1 and λ_2 indices, resulting in an intermediate variable T:

$$T_{\alpha_x\alpha_y\alpha_z,\beta_x\beta_y\beta_z} = \sum_{\lambda_1=0}^{l+\alpha}\sum_{\lambda_2=0}^{l+\beta} R_{\lambda_1,\lambda_2}\left(2+n+\alpha+\beta,\zeta,\eta_A,R_A,\eta_B,R_B\right)\Omega_{l,\lambda_1\lambda_2}^{\alpha_x\alpha_y\alpha_z,\beta_x\beta_y\beta_z}\left(\mathbf{r_A},\mathbf{r_B}\right) \quad (15)$$

where the six summation indices obey the conditions $\alpha_x+\alpha_y+\alpha_z \le L_a+1$, $\beta_x+\beta_y+\beta_z \le L_b+1$, and $\alpha_x+\alpha_y+\alpha_z+\beta_x+\beta_y+\beta_z \le L_a+L_b+1$, and we have omitted the index l for clarity because it is not a summation index. Next T is contracted with a different angular factor $\Theta^0_{b_xb_yb_z,\beta_x\beta_y\beta_z}\left(B_x,B_y,B_z\right)$ where the indices β_x,β_y,β_z are eliminated, resulting in another intermediate variable G:

$$G_{\alpha_x\alpha_y\alpha_z,b_xb_yb_z} = \sum_{\beta_x=0}^{b_x}\sum_{\beta_y=0}^{b_y}\sum_{\beta_z=0}^{b_z}\Theta^0_{b_xb_yb_z,\beta_x\beta_y\beta_z}\left(B_x,B_y,B_z\right)T_{\alpha_x\alpha_y\alpha_z,\beta_x\beta_y\beta_z} \quad (16)$$

By analogy, we define $\bar{G}$ by contracting over the $\alpha_x,\alpha_y,\alpha_z$ indices:

$$\bar{G}_{a_xa_ya_z,\beta_x\beta_y\beta_z} = \sum_{\alpha_x=0}^{a_x}\sum_{\alpha_y=0}^{a_y}\sum_{\alpha_z=0}^{a_z}\Theta^0_{a_xa_ya_z,\alpha_x\alpha_y\alpha_z}\left(A_x,A_y,A_z\right)T_{\alpha_x\alpha_y\alpha_z,\beta_x\beta_y\beta_z} \quad (17)$$

Contracting over the last angular factor $\Theta^x_{a_xa_ya_z,\alpha_x\alpha_y\alpha_z}\left(A_x,A_y,A_z\right)$ and multiplying by some parameters gives the integral gradient:

$$\frac{d}{dA_x}\gamma^l_{a_xa_ya_z,b_xb_yb_z} = 16\pi^2 d_u d_a d_b \sum_{\alpha_x=0}^{a_x+1}\sum_{\alpha_y=0}^{a_y}\sum_{\alpha_z=0}^{a_z}\Theta^x_{a_xa_ya_z,\alpha_x\alpha_y\alpha_z}\left(A_x,A_y,A_z\right)G_{\alpha_x\alpha_y\alpha_z,b_xb_yb_z} \quad (18)$$

Finally, the integral gradient is contracted with the density matrix, representing a fourteen-fold loop in total (as the six-fold summation in Eq. (14) has two constraints). The calculation of the ECP integral is a ten-fold loop because it does not involve contracting with the density matrix.

The calculations that transform the radial integral into the ECP integrals and gradients present a large number of possibilities in terms of which intermediate variables to store, which intermediate variables are recomputed, and the possible orderings of the computations. Here, we introduce a graph representation to formalize these choices and apply ACE to optimize the computations. Although $\Gamma^{l(\xi)}_{ab}$ is used as an example for this paper, other integrals or integral gradients are generated and optimized in similar ways. The next section describes how the equations are represented using a *dependence graph* structure and a set of decisions, which together with the dependence graph is referred to as a *decision graph*.

2.2 Design of Code Generator-Generating Different Code Variants

The code generator is the most important module in the ECP-ACE, as it generates the code candidates to be profiled. The code generator has two major functions: 1) generating multiple code variants, and 2) performing loop unrolling. As the registers are limited resources on the GPUs, we design a code generator that can tune the register usage for different code variants in a controllable way. Therefore, we adopt ideas from *liveness analysis*[31] in compiler theory and the *interference graph*[32,33] which is closely related with register allocation optimization.

A few important concepts about the interference graph are: 1) a variable is *live* if it holds a value that may be needed in the future, 2) redefinition of a variable changes its value, therefore is equivalent to destroying the old variable and creating a new variable, 3) variables a and b *interfere* if they cannot be allocated to the same register at the same time, which happens when a and b have *overlapping live range*. 4) An *interference graph* is an undirected graph where nodes represent variables and edges connect variables that interfere. The register allocation problem can then be reinterpreted as coloring the nodes in the interference graph with the minimum number of colors, under the constraint that nodes connected by edges cannot have the same color. If the minimum number of colors K is less than the available number of registers N_{reg}, then the compiler (e.g. nvcc) will allocate K registers for the function. However if K is greater than N_{reg}, the compiler will select nodes to push to the stack (register spilling), which we discuss more below.

Concepts 3) and 4) suggest that the number of registers consumed is closely related to the maximum number of interfering variables. Concept 2) suggests that redefinition can be used as a tool to reduce the number of interfering live variables by recomputing some variables rather than keeping them alive through the entire function. However, recomputing intermediates increases the number of floating point operations (flops). Therefore, *the best code variant will be a tradeoff between consumed flops and register usage*.

The above analysis suggests that we can generate code variants with different flops-storage tradeoffs by deciding which intermediates to store and which to recompute on the fly. To help generalize the decision procedure, we introduce two types of graphs: 1) the *dependence graph*, which represents the procedure for evaluating the integral expression and how intermediates are related, and 2) the *decision graph*, which represents the decisions regarding intermediate storage and uniquely defines each code variant.[34]

2.2.1 Dependence Graph

Dependence analysis with its graph representation is crucial in the development of compiler technology. The data dependence,[35,36] which represents whether the computing of data *a* requires the knowledge of another data *b*, i.e. *a* is dependent on *b*, is one of the fundamental types of dependence relationships. Data dependence has played an important role in the instruction scheduling optimization in combination with control dependence analysis[37,38] and program dependence analysis.[39,40] Therefore, we borrow the basic idea from the formal data dependence graph used in the compiler theory, and set up a simplified dependence graph to facilitate the design of the code generator.

The dependence graph we set up has two basic components:

- *Nodes*. A node stores a list of mutually independent values, and can be used to represent a collection of variables identified by one or more summation indices. Each node has the following associated properties: 1) node index, 2) node name (which is a string to be used for declaring variables in the generated code), 3) color, which could be *input, output,* or *intermediate,* 4) an ordered-list of the variables, and 5) a list of the summation indices for each variable. For example, a node representing G from Eq. 16 contains a list of variables, as well as their corresponding summation indices α_x, α_y, α_z, b_x, b_y, and b_z.
- *Vertices*. A vertex represents the mathematical relationships between the variables in the three nodes connected to it. There are two *parent nodes* above the vertex, whose variables are inputs for computing those of the *child node* below. The I-th variable in the child node c_I can be computed from the variables in the two parent nodes p and q as:

$$c_I = \sum_{j=0}^{n_p} \sum_{k=0}^{n_q} \hat{V}_{I;j,k}\left(p_j, q_k\right) \tag{19}$$

where p_j and q_k denote the jth/kth variable of node p/q, respectively. Therefore each vertex is associated with the following properties: 1) pointers to the left parent node, right parent node, and child node, and 2) a map of mathematical operations $\hat{V}_{I;j,k}\left(p_j, q_k\right)$ that maps the triplet (I,j,k) to a function that takes p_j and q_k as arguments. The operation function V can take arbitrary form, and determines how p_j and q_k together contribute to c_I. The vertex keeps track of all the triplets (I,j,k) where the contribution is non zero. For example, a vertex that connects the parent nodes $R(\lambda_1,\lambda_2,N)$ and $\Omega_{l,\lambda_1\lambda_2}^{\alpha_x\alpha_y\alpha_z,\beta_x\beta_y\beta_z}\left(\mathbf{r}_\mathbf{A},\mathbf{r}_\mathbf{B}\right)$ to the child node $T_{\alpha_x\alpha_y\alpha_z,\beta_x\beta_y\beta_z}$ requires that the λ_1 and λ_2

indices match in the parent variables, and that the α_x, etc. indices match between the variables in Ω and T.

One possible dependence graph for the ECP gradient is shown in Figure 1. The ECP integrals have a very special property. For all vertices, one of the parent nodes depends directly on the input nodes, and is much easier to compute than the other parent node (i.e. the angular factors mentioned above). As a result, we can always compute the simpler parent on the fly without much penalty. This procedure can be interpreted as defining a new operator

$$c_I = \sum_{k=0}^{n_q} \hat{A}_{I;k}\left(q_k\right) = \sum_{k=0}^{n_q} \left\{ \sum_{j=0}^{n_p} \hat{V}_{I;j,k}\left(p_j, q_k\right) \right\} \qquad (20)$$

The new operator $\hat{A}_{I,k}$ can be represented as a *dressed arrow*. By removing one of the parents for each vertex, we get a *reduced dependence graph*. The root of the reduced dependence graph is the input node, and the leaves are the output nodes. Each intermediate node has only one parent node, but can have multiple child nodes. Nodes with multiple child nodes are called *branching nodes*. The reduced dependence graph for $\Gamma_{ab}^{l(\xi)}$ is shown in Figure 2a. It has the following components:

- *Nodes.* The definitions of nodes are same as before.
- *Arrows.* Each arrow defines the relationship between the parent node and child node it connects. It has the following associated properties: 1) pointer to the parent node, 2) pointer to the child node, and 3) operation matrix elements $\hat{A}_{I,k}$. The matrix elements are represented as strings, which can be printed when generating code. If $\hat{A}_{I,k} = 0$, then the *I*-th variable in the child node is independent of the *k*-th variable in the parent node.

The distances between nodes within the same graph are defined as the shortest path connecting them. Therefore, for a given reference node, it is possible to compare whether one node is closer to the reference node than another.

In the code generator, each type of integral is represented by its own *initial dependence graph,* which defines variables in the nodes and operations in the arrows. Given the initial dependence graph, the code generator first performs graph transformations and creates several equivalent dependence graphs with differing topology. One possible way to transform one dependence graph into another is by removing the branches. For example, the node $T_{\alpha\beta}$ in Figure

2a is a branching node with two child branches. According to the liveness analysis, if we compute the left branch first, then the node $T_{\alpha\beta}$ interferes with the entire left branch because the right branch is dependent on $T_{\alpha\beta}$. In order to remove this interference, we can add another node *T'* to the top of the right branch, corresponding to recomputing the node $T_{\alpha\beta}$ at the beginning of the right branch. This node duplication (which can also be thought of as branch removal) decreases the maximum number of interfering variables but will increase the flops - another example of the flops-storage tradeoff. This basic idea is illustrated in detail in Section 2 of the Supporting Information, where we have included several other dependence graphs of $\Gamma_{ab}^{l(\xi)}$ transformed from Figure 2a by removing branches.

In this paper, we only discuss code generation from reduced dependence graphs. We will talk about code generation from general dependence graphs in a forthcoming paper.

2.2.2 Decision Graph Representation of Different Code Variants

A decision graph has the same topology as the dependence graph from which it is created, except that each node colored as *intermediate* in the dependence graph is further designated as *stored* or *transient* in the decision graph. The color *stored* implies that variables in the nodes are computed only once, and stored in registers/memory for all future usage. The color *transient* denotes that variables in the nodes are recomputed on the fly whenever needed. Figure 2b shows one possible decision graph corresponding to the dependence graph in Figure 2a.

In order to generate codes from a decision graph, we use the following strategy.

1) If a node is designated as stored, then all subsequent calculations depending on this node will read from its stored values. Thus, we disconnect the graph at the stored nodes into independent subunits with well-defined structures. Each subunit represents a *job* to be processed by the code generator. Processing a job refers to generating the code that calculates the stored variables for the graph subunit. Each job has a directionality, starting from the stored nodes where the variables are known, called the *source*, and moving forward in the direction of arrows toward the stored nodes whose values are to be computed, called the *sink.*

2) The jobs need to be ordered such that dependencies are satisfied and calculations are not repeated. To determine this ordering, we create a job stack for each output node. Starting with the first output node, the first job pushed onto the stack is the subunit containing the output node itself. We then follow up the dependencies and search for the next job that takes the source node

of the first job as its sink node. The detected job is pushed onto the stack. This procedure is repeated recursively until the input node is reached, which indicates that the whole stack of the first output node has been built. The code to compute the first output node is generated by popping each job off the stack and processing it. For the remaining output nodes, jobs are detected and pushed onto the stack in the same recursive way until either it reaches the input, where a complete job stack has been built; or it encounters a job that has already been processed, which indicates that codes corresponding to the job and all jobs prior to it have been generated by a previous output node.

Now that the decision graph has been broken down into an ordered list of individual jobs, we consider how to generate the code that computes the variables in the sink nodes from the source node within a job. By examining any transient variable in a graph subunit, we can identify two types of fundamental operations:

(1) Backward Propagation computes a variable by applying a chain of operations starting from the closest source node. Figure 3a shows the graph representation of a backward propagation. As each node has only one parent node, the direction of backward propagation is unique and defined by the intermediate node *T* and the variable *t* which is being computed. It can be constructed recursively, as shown in Table S3 of the supporting information.

(2) Forward Propagation applies a chain of operations to the variable to compute its contribution to stored variables in the sink node. Figure 3b shows the graph representation of the simplest forward propagation, which has a single sink node. As each node can have multiple child nodes, a more general type of forward propagation can have multiple sinks, as shown in Figure 3(c). As the sink nodes can potentially compete to be computed first, we assign the left branches with higher priority to resolve the ambiguity. Similar to backward propagation, both single sink and multi-sink forward propagation can be constructed recursively.

By using these two types of basic propagations, we can construct types of basic sub-unit structures.

(1) Closed connection. Figure 4a shows the graph representation of a closed connection. A closed connection is a series of non-branching nodes connecting a single source with a sink. For each closed connection, one transient node needs to be designated as *primary* representing the outermost loop. Each variable in the primary node is computed using backward propagation, and contributes to variables in the sink using forward propagation. Therefore, the closed connection is uniquely defined by the chosen primary node, and can be divided at the primary node into a backward propagation followed by a single sink forward propagation as shown in Figure 4a.

From liveness analysis, the source node and the sink node are interfering throughout the closed connection computation. As a result, the size of the memory request for a closed connection is equal to the number of variables in the source node plus the number of variables in the sink node.

As we have discussed before, variables in the primary node are computed only once while variables in other transient nodes are computed multiple times as required by the propagation steps. The choice of primary node in closed connections does not affect storage but could have an impact on flops. For example, it is always preferable to choose a transient node instead of the source node as primary; since retrieving variables from the source node has no flop cost, designating it as primary and accessing it the least number of times is not beneficial to the performance. In contrast, choosing a transient node as primary may potentially reduce the total number of flops, as it minimizes the number of requests to the variables belonging to the primary node, which are only accessible at the cost of floating point operations.

(2) Transient Branching. Figure 4b shows the graph representation of a transient branching structure, which contains one source node, one or multiple transient branching nodes, and multiple sink nodes. If there are multiple transient branching nodes, the one closest to the source node will be chosen as the *primary branching node*, which defines the transient branching structure. This is the only branching node that can be chosen as primary because it is connected to all sinks by forward propagation. By splitting the graph at the primary branching node, a transient branching subunit can be divided into a backward propagation from the primary branching node, followed by a multi-sink forward propagation, as shown in Figure 4b.

By liveness analysis, the source node and all sink nodes interfere with each other. Therefore, the memory request for a transient branching structure is the number of variables in the source nodes, plus the sum of number of variables in the sink nodes.

(3) Stored Branching. Figure 4c shows the structure of a stored branching structure containing one source node, one stored branching node, and multiple sink nodes. As shown in Figure 4c, by disconnecting the graph at the stored branching node, the stored branching subunit can be divided into closed connections and transient branching subunits. Therefore, the stored branching structure does not need to be included as a basic subunit structure – however, the stored branching node affects the liveness analysis as it interferes with all nodes on its branches except the rightmost branch.

The above analysis shows that the entire decision graph can be split into either closed connections (identified by the primary node) or transient branching structures (identified by the primary branching node), corresponding to different types of jobs (i.e. kinds of generated code). Table S3 provides the pseudo-code for the processing of different job types. Step IV in Figure 5 illustrates the job stack and processing order for the decision graph in Figure 2b.

2.2.3 Summary of ACE-code generator

Figure 5 shows the workflow of the code generator that uses the reduced dependence/decision graph to generate all possible code variants known to the code generator. A code variant is uniquely defined by the structure of its decision graph, the decisions regarding stored or transient colorings of intermediate nodes, and the designation of primary nodes in closed connections with multiple transient nodes.

For each code variant generated, the code generator reports the floating point operations (flops) and the maximum number of interfering variables of any individual subunit.[41] If there were an infinite number of registers, then the maximum interfering variables reported provides an estimate to the number of registers required. However, when register spilling takes place due to limited physical registers, analysis becomes much more complicated. First of all, it is usually difficult to predict which variable the compiler will choose to push to the stack (which is also called local memory for CUDA-GPUs). Second, the compiler has to transform the code as pushing a variable to stack. Loading a variable from stack is at least accompanied by allocating an additional variable in the registers, and performing a *load* instruction from the stack to the register. The number of load/store instructions issued due to register spilling is not easy to predict; the additional variable may also cause spilling on other variables and trigger further code transformations. This is especially notable in kernels with larger angular momentum, as register spilling becomes severe. As will be shown in the results section, when register spilling happens, the stack used may not be closely correlated with the maximum number of interfering live variables, and the large number of load/store instructions issued may sometimes become the bottleneck.

The code generator performs loop unrolling as it generates each code variant. The loop unrolling has three benefits:

1) Reducing loop overhead.

2) Reducing stack usage: compared with CPUs, load/store instructions with stacks on GPUs are much slower. Allocating arrays on the stack is required if values need to be loaded from/written to an address depending on the iterator. As registers are not indexable, unrolling the loops can avoid the usage of arrays as all indices are known at compilation time. A series of instructions associated with array manipulations may also be avoided.

3) Using compile time constants. Coefficients like $\int S_{lm} S_{\lambda\mu} x_n^i y_n^j z_n^k \, d\Omega$ and $\binom{a_x}{\alpha_x}$ do not depend on parameters of the primitive Gaussian functions except the angular momentum; the values of these functions are evaluated by the code generator and written to the generated code, so they need not be evaluated at run time.

However, the downside of loop unrolling is that it increases the size of the codes – this increases the search space for the compiler and could make compiler optimization more difficult.

The generated code depends on the ordering of variables within a node, which could potentially influence the performance. In this work, we stick with one predefined ordering for all the illustrated results. However, users can provide other orderings to the code generator, which allows the exploration of many more code variants. In principle, this could be automated and the code generator could scan over many possibilities. However, the number of code variants will grow quickly, so some form of model-based optimization approach would be well-advised if this strategy were to be pursued.

2.3 GPU Configuration Parameters

In addition to the code variants, the hardware configuration settings defined at compile-time can also affect performance. We consider the following GPU configuration parameters: 1) the maximum number of registers per thread allowed, 2) number of threads per block, and 3) L1 cache/shared memory configuration (applicable to architectures prior to Maxwell). All three parameters can change how many active warps/blocks can be executed simultaneously on a streaming multiprocessor, which can be evaluated as achieved occupancy (i.e. the average fraction of warps that are active on a multiprocessor), as discussed below.

The first parameter changes the number of registers required per thread, and is passed to the CUDA compiler using the *–maxrregcount* flag. Setting a large number of registers per thread may increase performance by avoiding register spilling within each warp, but may also decrease performance as the number of warps that can be executed simultaneously decreases due to the limited size of register files.

The second parameter changes the size of shared memory required per block. In ECP gradient computations, the shared memory requested is linearly dependent on the block size, and is used for contracting gradients on the same ECP centers. The block size can be changed when launching kernels. Requiring more shared memory per block can increase performance by enabling more communication and contraction between threads within a block, but the downside is the reduced occupancy due to hardware limitations on the amount of physical shared memory.

Unlike the above two parameters, which change the requests for each warp/block, the last parameter influences the occupancy and the performance by changing how the same physical resources are partitioned between different usages. For architectures prior to Maxwell, the L1 cache and shared memory use the same physical hardware resources and the split between these can be set in software. The Fermi architecture supports 16KB/48KB or 48KB/16KB partitions between shared/L1 cache memory, and the Kepler architecture supports an additional 32KB/32KB partition. The preference can be set separately in each kernel by calling *cudaFuncSetCacheConfig()*. According to the discussion for block size, larger shared memory configuration can increase occupancy, but the consequence of reducing L1 cache size is an increase in load/store misses from L1, and consequently more load/store from L2 cache or global memory which have much higher latency. Therefore, there is a tradeoff between shared memory and L1 cache that may be worth tuning.

There are many combinations between code variants and GPU configurations. We use the term *code candidate* to represent a specific combination of a code variant and GPU configuration. Although one could test all the possible combinations of code variants and GPU configurations, the number of code candidates would rapidly become very large, making the optimization prohibitively time consuming. Therefore, during the optimization, we first find the optimal code variants under default GPU configurations: block size is set to 64; maximum number of registers per thread is 63 for CC2.0 and CC3.0, 225 for CC3.5 and CC5.0; and

L1/shared configuration is 16KB/48KB partition. We then optimize the GPU configuration for the optimal code variant.

2.4 **Code Tester and Code Optimizer**

The code tester and code optimizer are the other two components in ACE. The code tester is responsible for compiling the code candidates and verifying whether each compiled code candidate can correctly reproduce the Fock matrix and gradients. The code generator emits formally correct code by construction, but it is nevertheless prudent to test for correctness explicitly. This ensures that any potential roundoff errors from reordering floating point operations will be flagged. The ACE framework will only consider code candidates that pass the correctness check in the code tester. Currently, the code tester simply works by comparing the result of running the generated code on a test system with a reference result stored on the disk.

The code optimizer collects timing data and analyzes the results. Suppose there are N_{test} test systems, N_{run} timings are collected for each test system, and the time of code candidate I on test system s in the nth run is $t_{s,n}^{I}$. We compute the mean time of each code candidate as the average time over all runs:

$$MeanTime(I) = \frac{1}{N_{run}} \sum_{n=1}^{N_{run}} \sum_{s=1}^{N_{test}} t_{s,n}^{I} \tag{21}$$

The code candidate with the smallest mean time is then selected as the optimal one.

In order to get accurate and valid timing data, two questions need to be addressed. The first question is how to do deviation detection. To prevent including timing data with large deviation, here we compute the standard deviation of the set $\left\{ t_{s,n}^{I} \middle| 1 \le n \le N_{run} \right\}$ before computing the mean time. If the standard deviation exceeds some user defined threshold, the entire set is discarded and the corresponding calculations will be run and timed again. The second question is what requirements a proper test system should satisfy; we simply require that the system should be large to provide enough primitive Gaussian basis functions and ECP primitive functions. From a data analysis point of view, large testing systems tend to have smaller coefficients of variation; from the performance point of view, whether the card is saturated or not may have a large impact on the performance evaluation. We describe how the test results depend on system size in the Supporting Information.

The entire workflow of ACE is described in Figure 6. The main driver collects information about the user's platform and analyzes the user's requests, then performs two optimization cycles as shown in Figure 6. The first cycle optimizes the code variants under the default GPU configurations, and the second cycle optimizes the GPU configurations for the optimal code variants. After the two optimization cycles are completed, the driver signals the user and hands back the final optimal programs.

3. Results and Discussion

In this section, we compare and discuss the performance data collected by ACE. We tested ACE for ECP integrals and gradients on four different platforms: NVIDIA Tesla S2050 (Fermi architecture, CC2.0), NVIDIA GeForce 680 (Kepler architecture, CC3.0), NVIDIA GeForce Titan (Kepler architecture, CC3.5), and NVIDIA GeForce 970 (Maxwell architecture, CC5.0).[42] For simplicity, we use the shorthand notation Precision-l-L_aL_b to denote the type of gradients computed. For example, *Double-0-PP* represents the kernel that computes $\Gamma_{ab}^{l(\xi)}$ in double precision with (l=0, a=1, b=1). Detailed information about the test systems can be found in the supporting information. Timings on each system are run three times.

3.1 Code Variant Optimization

We start by comparing the performance between 36 different code variants with the kernel and GPU architecture held constant. The 36 code variants are all generated based on the four dependence graphs in Figure S1, under the constraint that nodes with the same depth receive the same coloration. Figure 7 shows the performance comparison for Double-0-FF on a CC 3.5 GPU, demonstrating large performance differences between different code variants. The performance difference between the slowest and the fastest code variants is more than a factor of 6, and the difference for Float-0-FF is even greater (see Figure S3). In addition, there are no obvious correlations between the performance and the theoretical flops or maximum interfering variables. Therefore, it is difficult to predict the optimal code variant for the target kernel without performance testing.

We also found that the optimal code variant depends significantly on the target kernel and architecture. In order to better understand this dependence, we carried out a detailed analysis on ten chosen code variants with relatively good performance and a clear flop-memory tradeoff (these are labeled in Figure 7). The decision graphs corresponding to the ten chosen variants are given in Figure S4. The ten code variants are sorted in decreasing order of flops to memory ratio, such that variant #1 has the largest number of interfering live variables, and variant #10 requires the largest number of flops (see Figure S5).

We summarize the relative performance among the ten selected code variants for double precision kernels with *l*=0 on different architectures in Figure 8. A similar comparison for double precision kernels with *l*=1 and *l*=2 is given in Figure S6. In general, different code variants are optimal for different architectures and some of the observed differences can be rationalized on the basis of architectural features:

The CC5.0 architecture supports up to 255 registers per thread, and has the largest register file size and cache size. However, its double precision peak performance is low -- only 1/32 of its single precision peak performance as shown in Table S2. Therefore, the optimal code variants on CC5.0 usually have a lower flops-to-mem ratio. The most flop intensive code variants like #8 through #10 do not perform well on CC5.0.

The CC2.0 architecture is quite the opposite from CC5.0. Its double precision peak performance is half of its single precision peak performance. However, it only supports 63 registers per thread, and its register file size is the smallest; therefore, register spilling has a more significant performance impact for CC2.0. For kernels with small register spilling, like Double-0-SS to Double-0-PP, flop intensive code variants are usually the optimal ones. As the angular momentum of the basis functions increases (e.g. Double-0-DD), flop-intensive code variants become less favorable since load/store instructions caused by register spilling start to dominate the computations (discussed in detail below), and the optimal code variants gradually shift to the ones with more balanced flop-mem tradeoff. CC3.0 also supports only 63 registers per thread, but nearly every other feature is improved over CC2.0. The best-performing code variants in CC3.0 also favor the flop intensive side, although the performance differences are not as large as CC2.0.

For CC3.5, the optimal code variants have a more diffuse distribution than the others. CC3.5 combines the advantages of CC2.0 and CC5.0; it also supports 255 registers per thread, and its double precision peak performance is about one third of its single precision peak performance. Therefore, code variants with higher memory requests and code variants with higher flops may achieve the same good performance, which makes predicting the optimal code variant much more difficult.

Although Figure 8 (and Figure S6) appear to show some general trends for different architectures, it also indicates that the optimal code variant for each individual kernel does not follow a regular pattern. When comparing Double-0-DD (row 7), Double-0-PD (row 8), and Double-0-PP (row 9) on the CC3.5 architecture, the performance ranking across variants is quite different depending on the particular kernel. Variant #10 is the fastest for DD, but is almost the slowest for PP and PD; on the other hand, variants #1 and #7 are among the fastest for PP and PD but perform poorly for DD. The optimal code variants on CC2.0 also depend on the angular momentum as shown in Figure 8, but not in the same way as CC3.5. In fact, the optimal code variant for one architecture often has bad performance on another – for example, variant #4 which is optimal for PD on CC2.0 (row 20) is the worst on CC3.5, and variant #1 which is among the best for PP and PD on CC3.5 becomes the worst on CC2.0 (rows 20 and 21). These comparisons are also illustrated as a bar chart in Figure S8.

To understand the irregularities in performance rankings, we examined several profiling metrics measuring the number of instructions and local memory transactions, which are given in Table S6. We found that the factors limiting performance may vary strongly with the choice of kernel and architecture. Our results from examining PD on CC3.5 indicate that the performance of the slowest variant is limited by the large number of floating point operations. On the other hand, the performance of DD on CC3.5 is limited by register spilling and local memory transactions, since DD involves more intermediate and output variables than PP or PD. When comparing architectures, CC2.0 has far fewer registers than CC3.5 (63 vs. 255), which means register spilling becomes more severe. We observed that for DD on CC2.0, variant #8 has a larger number of load/store operations than #4, despite variant #8 being explicitly designed to have fewer interfering variables and more flops. This provides an example for the complexity that arises when register spilling happens, as it becomes difficult to predict the code that will be produced by the compiler.

To summarize, the irregular dependence of our observed performance rankings on the kernel and GPU architecture can be explained by examining the profiling metrics, but it also presents a significant challenge for model-based optimization approaches to predict the performance accurately. The model should be sensitive to the details of the kernel and architecture in order to determine which factors limit performance, and when register spilling could make the performance much more unpredictable. In-depth analyses of the profiling metrics that support these conclusions are provided in the Supporting Information.

3.2 GPU configuration optimization

We show how the GPU configuration can influence the performance for the optimal code variants selected in the previous sections. We use CC3.5 as an example, and choose three different maximum numbers of registers per thread (64, 128, 256), three different block sizes (64, 128, 256), and two different L1/Shared partitions (L1-preferred, i.e. 48Kb/16Kb, and Shared-preferred, i.e. 16KB/48KB). These give us 18 different combinations of configuration parameters. It turns out that the optimal GPU configuration, especially the maximum number of registers per thread, is not the same for different kernels, as shown in Figure 9. For the Double-0-DD kernel (Figure 9b), the default GPU configuration parameters give near-optimal performance, and tuning the parameters yields a negligible 2-3% improvement. For Double-0-SD (Figure 9a), tuning the configuration parameters yields a performance increase of ~15% over the default. In either case, the impact of tuning GPU configuration parameters has a smaller impact compared to the choice of code variant.

For kernels with high angular momentum like Double-0-DD, it is expected that 255 registers per thread gives the best performance, as it reduces the local memory transactions as much as possible. In contrast, smaller number of registers per thread performs better for kernels with smaller angular momentum like Double-0-SD. Take block size 64 with L1-preferred partition as an example. When compiled with *maxrregcount=255*, Double-0-SD uses 202 registers. When compiled with *maxrregcount=127*, Double-0-SD uses all 127 registers plus 216 bytes of local memory. Despite the register spilling caused by smaller number of registers allowed, the achieved occupancy between *maxregcount*=255 and 127 is 0.1190 versus 0.2356, which indicates that more warps can be executed simultaneously. This gives an example to show that register spilling is not always harmful, and the tradeoff between register spilling and achieved occupancy can be tuned to achieve better performance.

For ECP calculations, the effects from adjusting block sizes and L1/Shared partitions are smaller than changing the register configuration. This could be because the shared memory is only accessed at the end of each loop over ECP centers, and doesn't participate in the intermediate calculations. Therefore, configurations related to the shared memory should not make a large difference. Despite this, we observed that smaller block sizes and larger L1 partitions tend to lead to better performance. The former may result in better achieved occupancy, and the latter may result in a higher L1 cache hit rate.

It is worth noting that the optimal code variant may in fact depend on the GPU configuration parameters, a possibility that we did not fully investigate. More complete explorations of the space of code candidates – either by means of a full search over the space of code variants and GPU configurations, or iterating back and forth between code variants and GPU configurations until convergence – may be requested by the user in order to investigate these possibilities.

4. Conclusions

We presented the automated code engine for graphical processing units that automatically generates optimized integral kernels for a given GPU computing platform. The use of graph representations for the basic equations allows the ACE-code generator to be generalized to other types of integrals. The application to ECP integrals demonstrates the complex factors that influence performance and the challenges associated with correct prediction of optimal code variants. The performance driven optimization strategy adopted by the ACE-code optimizer, which scans over the space of code candidates and chooses the best one from empirical testing, greatly simplifies the optimization procedure and is easily adaptable to the fast evolution of GPU computing architecture.

There are several future directions for further improvement and application of ACE. One improvement is to generate codes based on the general dependence graph where multiple parent nodes are allowed for each node. Although the single parent node restriction embedded in the reduced dependence graphs works well for the ECP integral and gradient calculations, the restriction is too strong for other types of integrals. More complex integrals where this restriction should be lifted include the two-electron-three-center integrals and gradients (the foundation of density-fitting methods), and two-electron-four-center integrals and gradients (needed for Fock matrix element evaluation).

Another crucial requirement for applying ACE to more complex integrals is automatic exploration of more types of dependence graph transformations. Here, we have already described graph transformations regarding the branching structures, but other types of transformations are possible that become important for different types of integrals. One topic of current study is the formulation of recursion relationships in terms of dependence graph transformations, enabling automatic discovery and testing of different recursion paths. Another possible transformation is systematic splitting of output nodes. For example, the FF kernel has high register and memory pressure because each thread needs to compute and store the entire 10 by 10 matrix of final integrals. Alternatively, a group of several kernels can be generated to work independently where each only computes and stores a part of the full matrix. This strategy repeats computations in order to reduce the number of stored variables, and is appealing for higher angular momentum as it reduces the register and memory pressure for each individual kernel.

Combining the possible dependence graphs and their associated decision graphs leads to an exponential increase in the space of potential code variants. This can make it challenging to generate, compile, and test all of the resulting code variants. One possible solution is to allow ACE to score each of the code variants before entering the optimization step. As the maximum interfering variables and the number of floating point operations are directly available from the code generator, cost models can be applied to estimate the performance as a function of the interfering variables and the flops. This could allow ACE to screen out code variants with unreasonably high storage or flop requirements, reducing the number of code variants that need to be compiled and tested – thus combining the advantages of model-driven and performance-driven approaches.

Currently, the initial dependence graphs representing different integrals are set up by the programmer manually from analyzing the equations. It will be appealing to study the systematic creation of initial dependence graphs based on just the mathematical expression for the integral, which can make ACE applicable to a much broader range of problems in electronic structure theory. Work along these lines is in progress.

Acknowledgements

This work was supported by the National Science Foundation (ACI-1450179). TJM is grateful to the Department of Defense (Office of the Assistant Secretary of Defense for Research and Engineering) for a National Security Science and Engineering Faculty Fellowship (NSSEFF). CS is grateful for a Stanford Graduate Fellowship.

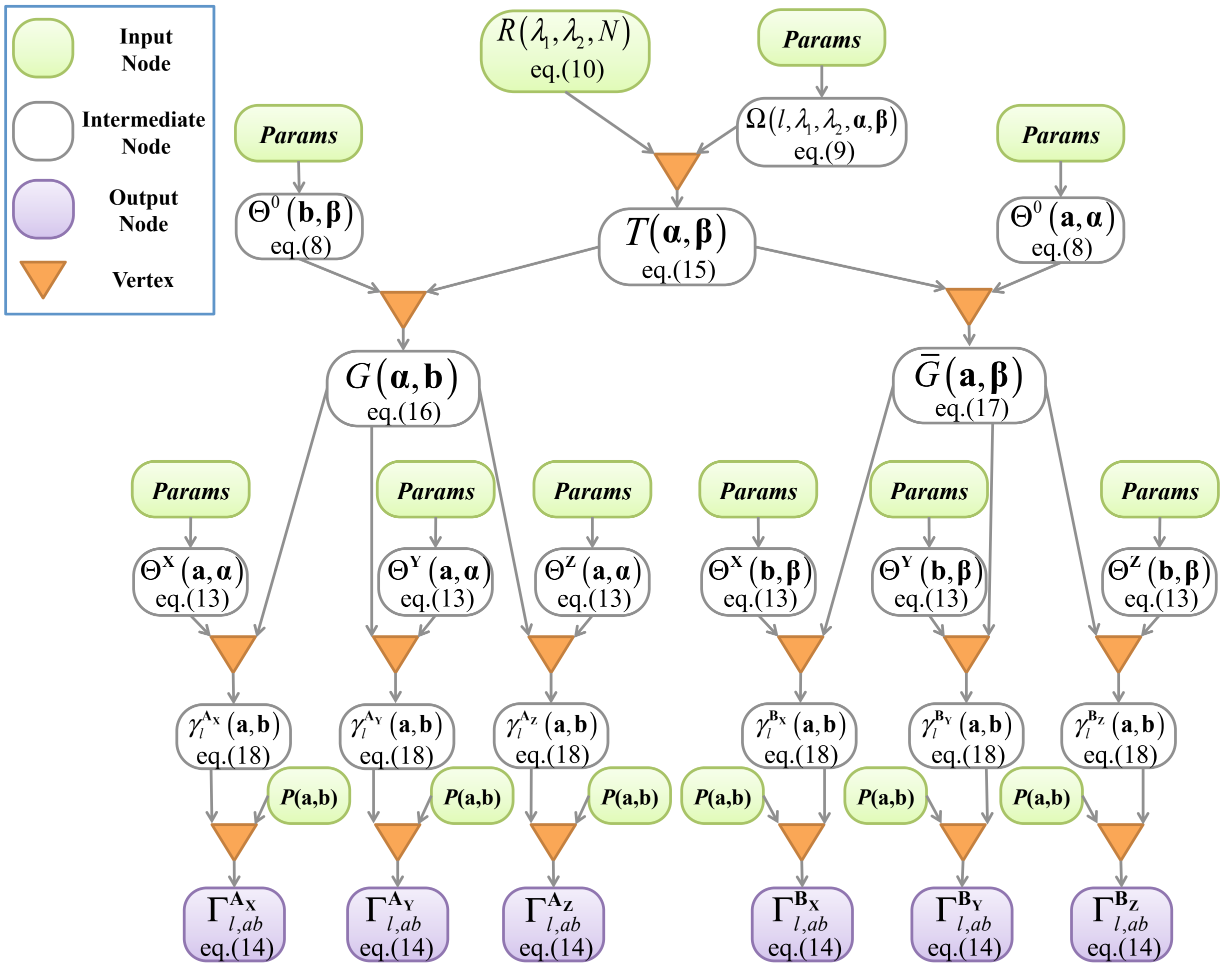
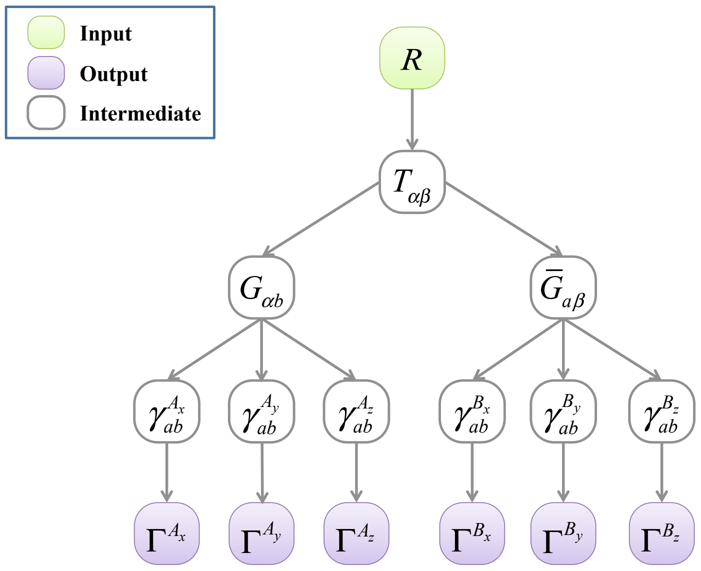

Input Node
Intermediate Node
Output Node
Vertex
$R(\lambda_1,\lambda_2,N)$ eq.(10)
Params
$\Omega(l,\lambda_1,\lambda_2,\boldsymbol{\alpha},\boldsymbol{\beta})$ eq.(9)
$T(\boldsymbol{\alpha},\boldsymbol{\beta})$ eq.(15)
$\Theta^0(\mathbf{b},\boldsymbol{\beta})$ eq.(8)
$\Theta^0(\mathbf{a},\boldsymbol{\alpha})$ eq.(8)
$G(\boldsymbol{\alpha},\mathbf{b})$ eq.(16)
$\bar{G}(\mathbf{a},\boldsymbol{\beta})$ eq.(17)
$\Theta^X(\mathbf{a},\boldsymbol{\alpha})$ eq.(13)
$\Theta^Y(\mathbf{a},\boldsymbol{\alpha})$ eq.(13)
$\Theta^Z(\mathbf{a},\boldsymbol{\alpha})$ eq.(13)
$\Theta^X(\mathbf{b},\boldsymbol{\beta})$ eq.(13)
$\Theta^Y(\mathbf{b},\boldsymbol{\beta})$ eq.(13)
$\Theta^Z(\mathbf{b},\boldsymbol{\beta})$ eq.(13)
$\gamma_l^{\mathbf{A}_X}(\mathbf{a},\mathbf{b})$ eq.(18)
$\gamma_l^{\mathbf{A}_Y}(\mathbf{a},\mathbf{b})$ eq.(18)
$\gamma_l^{\mathbf{A}_Z}(\mathbf{a},\mathbf{b})$ eq.(18)
$\gamma_l^{\mathbf{B}_X}(\mathbf{a},\mathbf{b})$ eq.(18)
$\gamma_l^{\mathbf{B}_Y}(\mathbf{a},\mathbf{b})$ eq.(18)
$\gamma_l^{\mathbf{B}_Z}(\mathbf{a},\mathbf{b})$ eq.(18)
P(a,b)
$\Gamma_{l,ab}^{\mathbf{A}_X}$ eq.(14)
$\Gamma_{l,ab}^{\mathbf{A}_Y}$ eq.(14)
$\Gamma_{l,ab}^{\mathbf{A}_Z}$ eq.(14)
$\Gamma_{l,ab}^{\mathbf{B}_X}$ eq.(14)
$\Gamma_{l,ab}^{\mathbf{B}_Y}$ eq.(14)
$\Gamma_{l,ab}^{\mathbf{B}_Z}$ eq.(14)

Figure 1. General dependence graph describing the ECP gradient calculation in a GPU kernel. The nuclear gradients of the ECP integrals for a primitive pair are computed as $\gamma_l^{\mathbf{A}_x}(\mathbf{a},\mathbf{b})$ and then contracted with the corresponding block of the density matrix $\mathbf{P}(\mathbf{a},\mathbf{b})$ to produce the final output, $\Gamma_{l,ab}^{\mathbf{A}_x}$. *Nodes* representing collections of variables and *vertices* representing mathematical relationships between nodes are described in Section 2.2.1. Equation references describe how the variables in each node are computed. Indices of intermediate variables are represented as: $\boldsymbol{\alpha} = (\alpha_x,\alpha_y,\alpha_z)$, $\boldsymbol{\beta} = (\beta_x,\beta_y,\beta_z)$, $\mathbf{a} = (a_x,a_y,a_z)$, $\mathbf{b} = (b_x,b_y,b_z)$. "***Params***" refer to parameters of basis functions, i.e. coordinates of atoms, exponential coefficients and contraction coefficients.

(a)



(b)

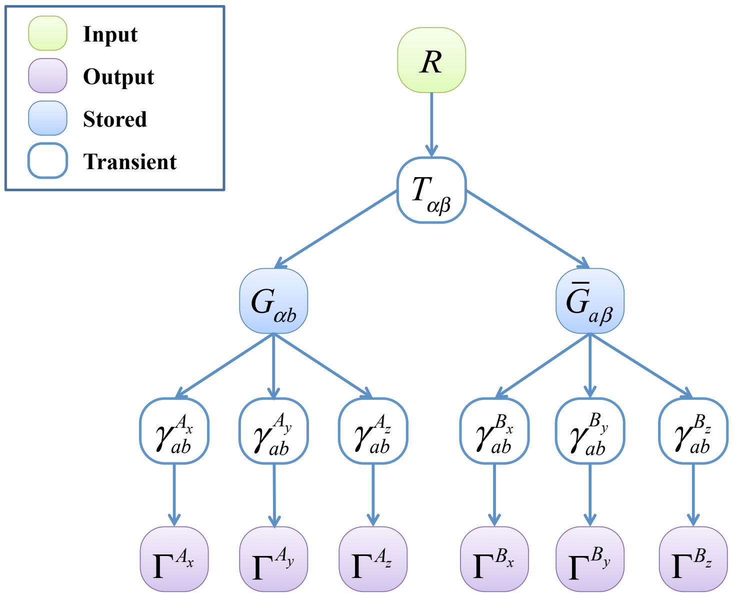


Figure 2. (a) Reduced dependence graph of $\Gamma_{ab}^{l(\xi)}$ corresponding to the full dependence graph in Figure 1. (b) One possible decision graph corresponding to the reduced dependence graph of (a), which represents code variant #6 in the main text. The notation for indices is simplified by omitting the total angular momenta and abbreviating variables with triples of indices as follows: G($\boldsymbol{\alpha}$,**b**) → $G_{\alpha b}$. The nuclear degree of freedom is denoted as ξ.

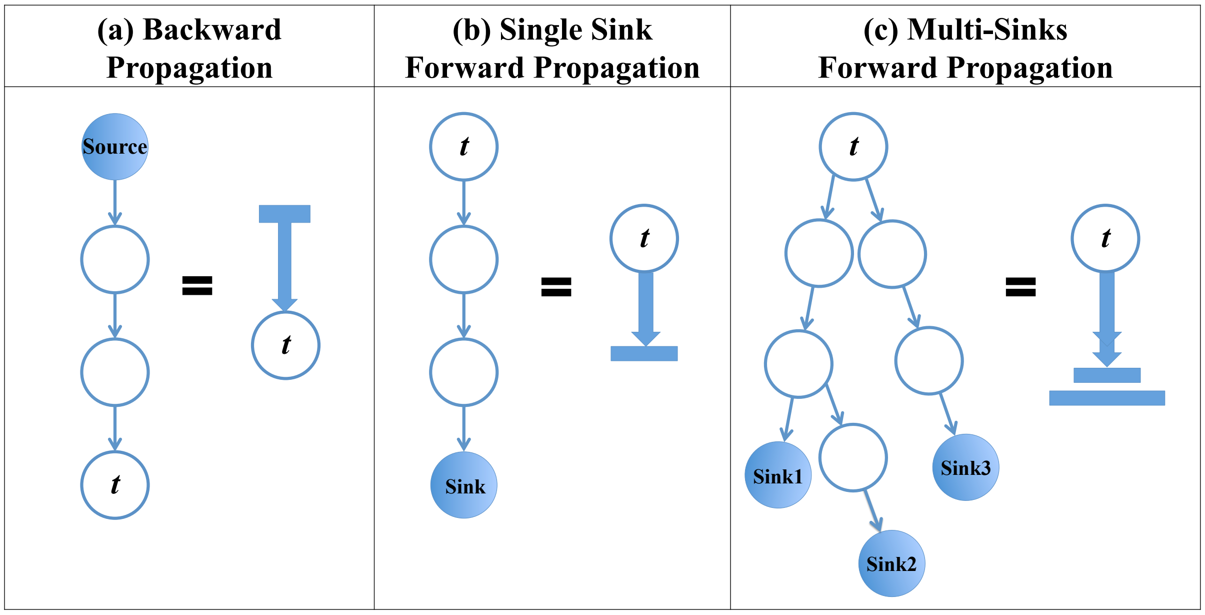

(a) Backward Propagation
Source
t
(b) Single Sink Forward Propagation
t
Sink
t
(c) Multi-Sinks Forward Propagation
t
Sink1
Sink2
Sink3
t

Figure 3. Graph representations of the basic propagations originating from any transient node in the graph (*t*) and ending at a stored node (*Source*/*Sink*). Within each column, the graph on the left gives an illustration corresponding to the type of the propagation, and the graph on the right shows the symbol that will be used to represent that propagation.

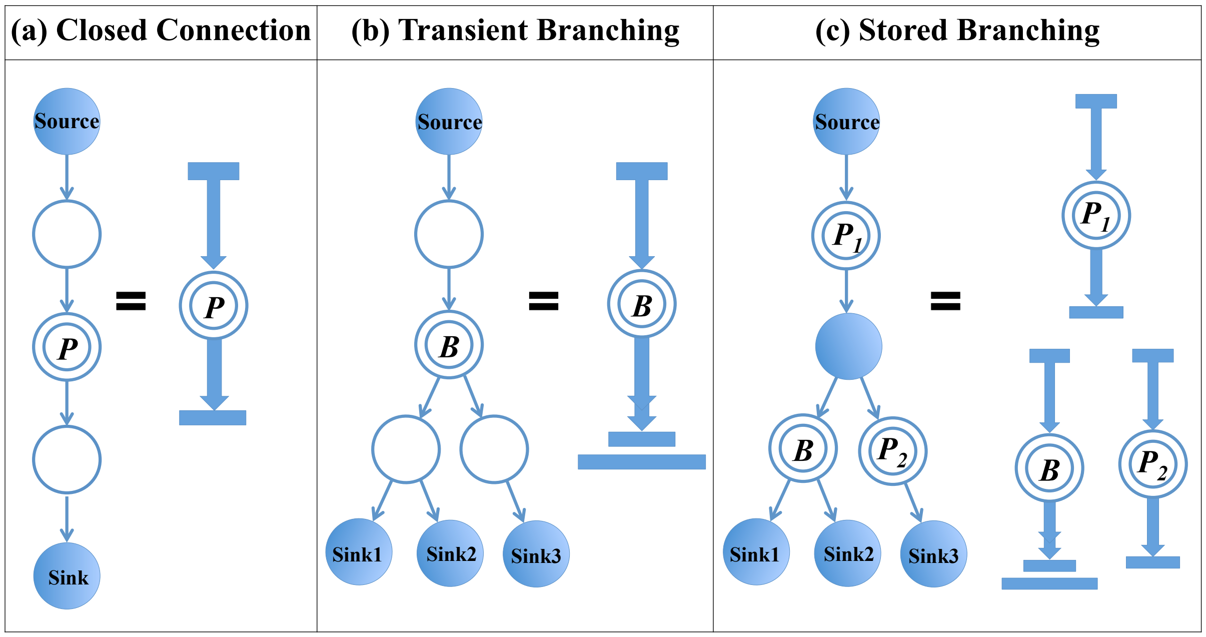


Figure 4. Graph representations of the three basic subunit structures. Within each column, the graph on the left gives an illustration corresponding to the type of the basic subunit structure, and the graph on the right shows how the subunits can be divided into basic propagations. The node labeled *P* represents the chosen *primary* node – note that in the closed connection, any transient node may be chosen as primary. The node labeled *B* represents the *primary branching* node.

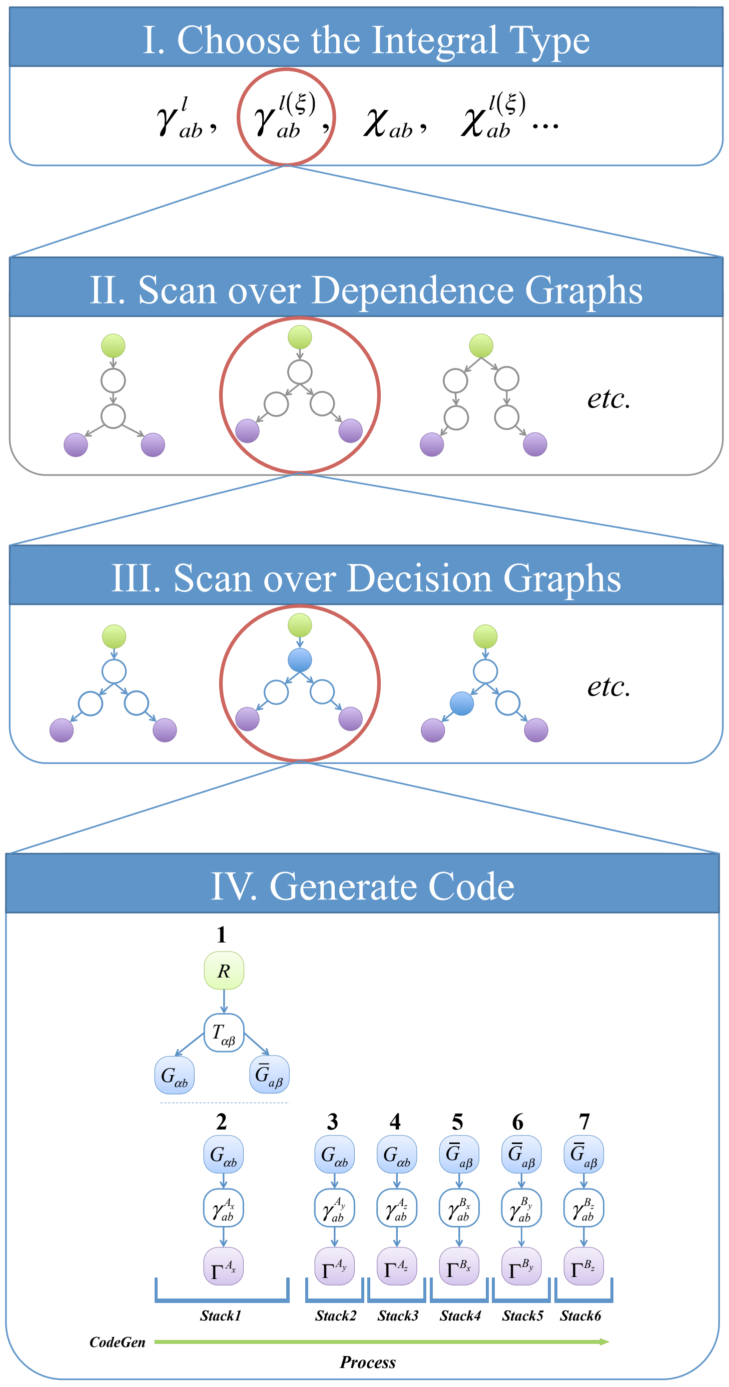

I. Choose the Integral Type
γ^{l}_{ab}, $\gamma^{l(\xi)}_{ab}$, χ_{ab}, $\chi^{l(\xi)}_{ab}$ …
II. Scan over Dependence Graphs
etc.
III. Scan over Decision Graphs
etc.
IV. Generate Code
1
R
$T_{\alpha\beta}$
$G_{\alpha b}$
$\bar{G}_{a\beta}$
2
3
4
5
6
7
Stack1
Stack2
Stack3
Stack4
Stack5
Stack6
CodeGen
Process

Figure 5. Workflow of code generator for producing all possible code variants for a chosen integral type. Notation follows Figure 2. Graph representations in steps II and III are toy models. The decision graph in Figure 2b is used as an illustration for step IV; the disconnected subunits are pushed into the job-stack for each output node, and the code generator produces code for each subunit in the indicated order. Note that since each closed connection (2-7) has only one transient node, it automatically becomes the primary node.

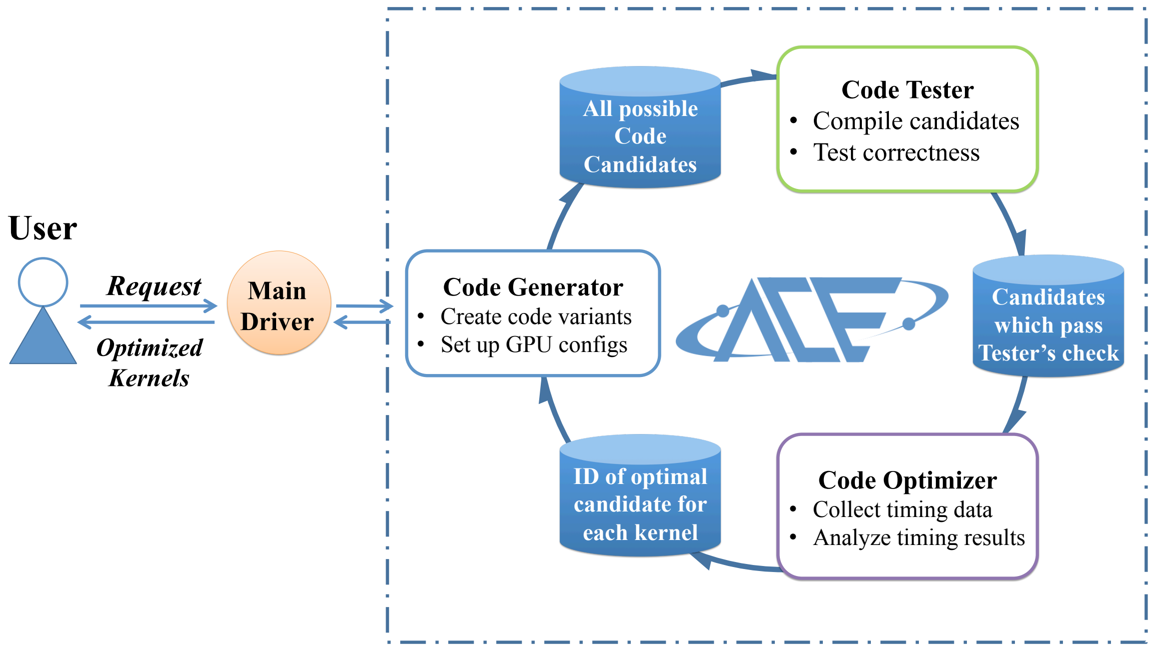


Figure 6. User interface and workflow of ACE. The user requests which type of integral to be optimized, the highest angular momentum, numerical precision and GPU compute capability. The main driver validates the user input and runs two optimization cycles - first optimizing over the code variants under the default GPU configuration, then optimizing over the GPU configurations for the chosen variant. The driver then returns the optimized kernels to the user.

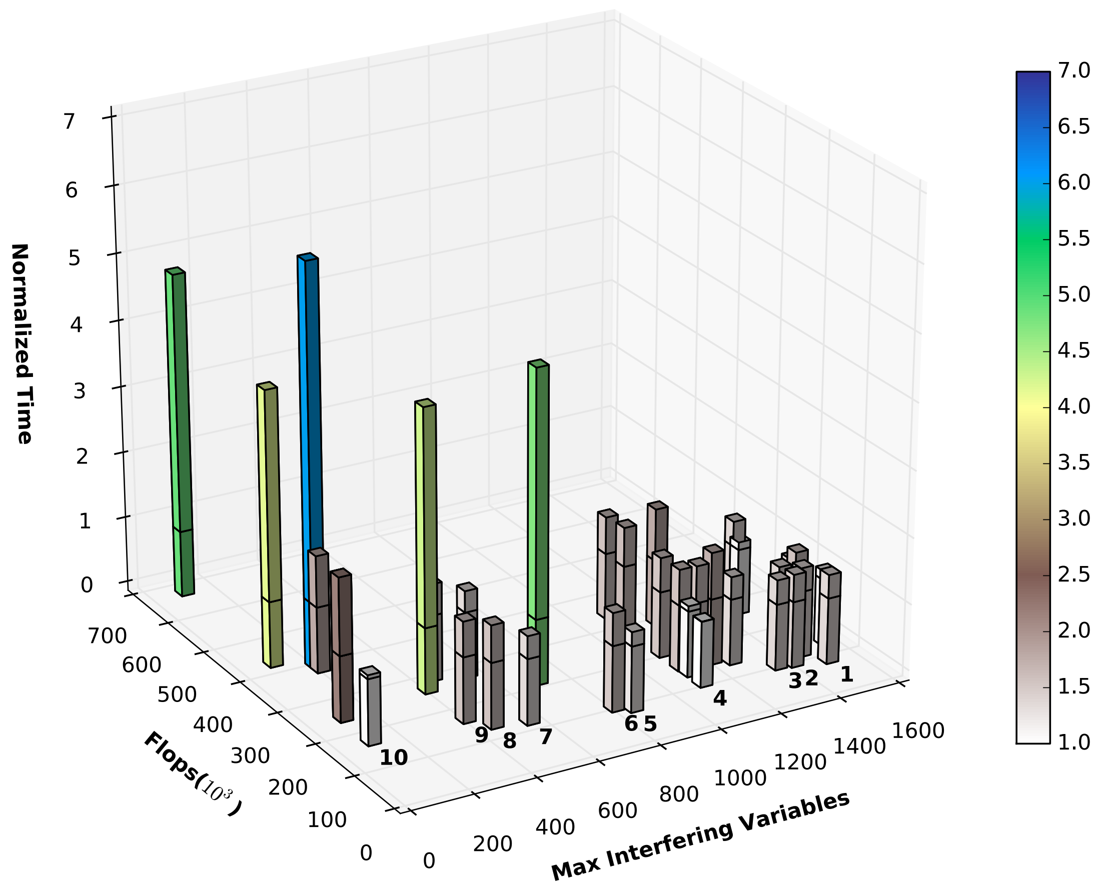


Figure 7. Performance comparisons between 36 code variants for Double-0-FF (double precision, *l*=0, *a*=3, *b*=3) on CC3.5 architecture. Normalized time is computed as the mean time of each code variant, as defined in Eq. (19), divided by the mean time of the optimal code variant. To show the differences between code variants clearly, the position of 1.0 is marked on each bar. The ten code variants selected for in-depth analysis in the main text are labeled using numbers 1-10.

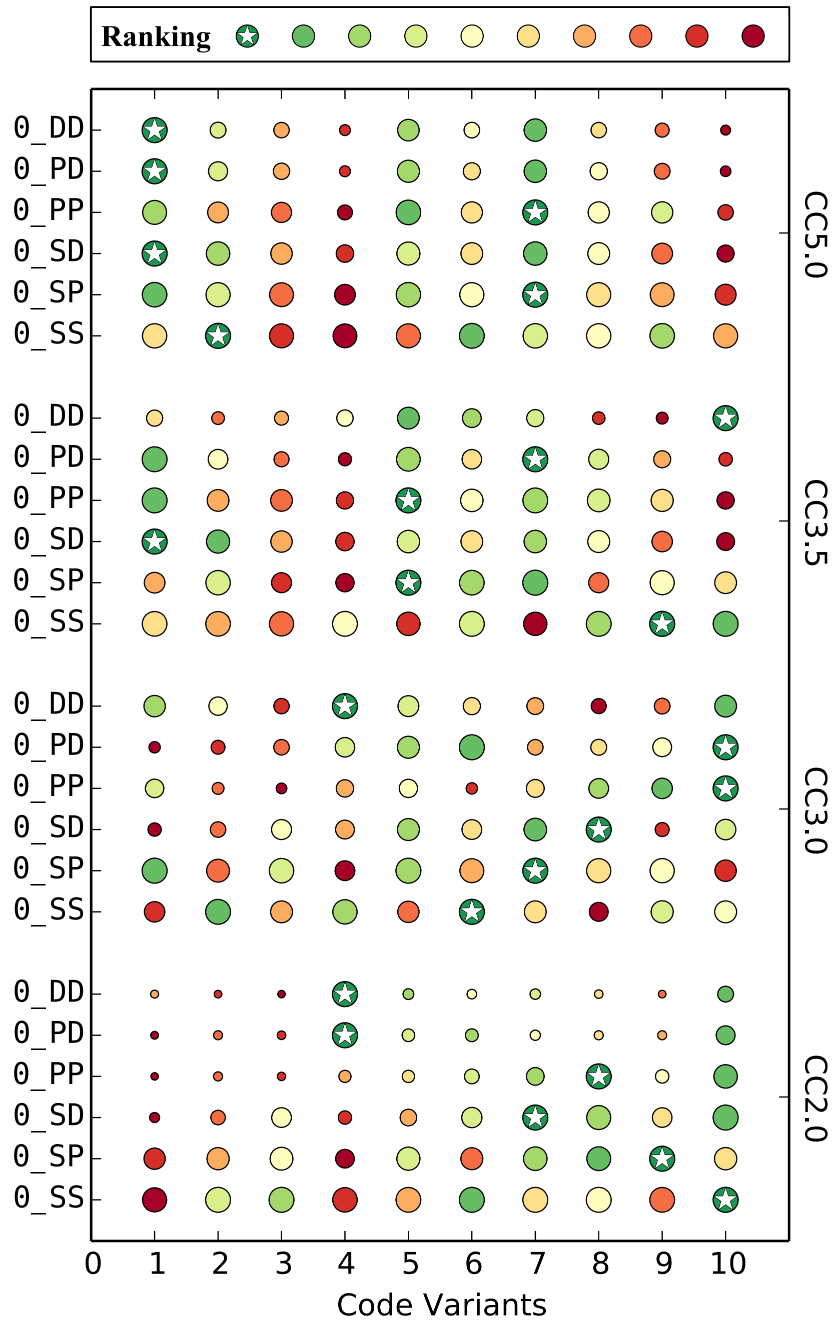
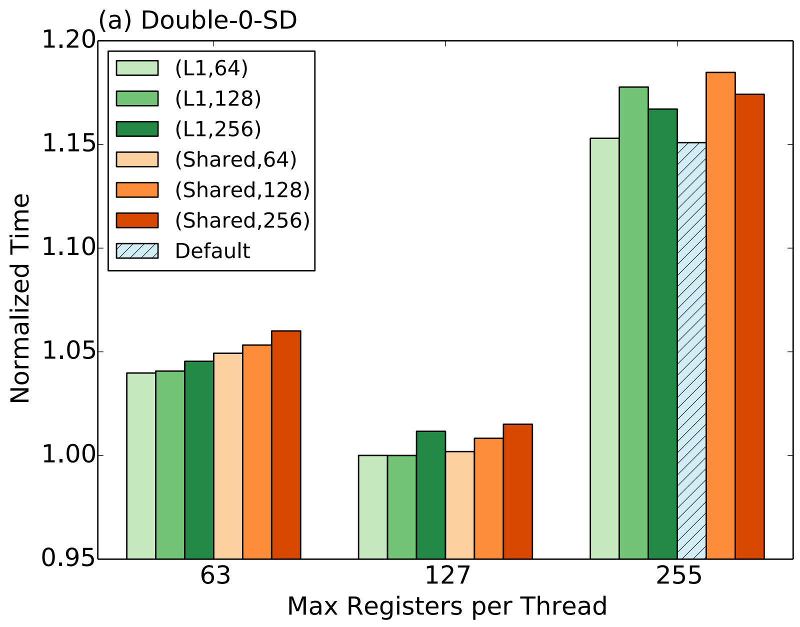


Figure 8. Performance ranking of code variants. Each row shows a different kernel where the primitive angular momentum and GPU architecture are varied. All kernels shown use *l*=0, double precision and the default GPU configuration. Within each row, the performance of the code variants are compared; the star represents the best code variant, the color of the circles represent the ranking of the variants as shown in the legend, and the radius of the circles indicate the performance of each variant relative to the optimal one – the smaller the radius, the lower the performance.



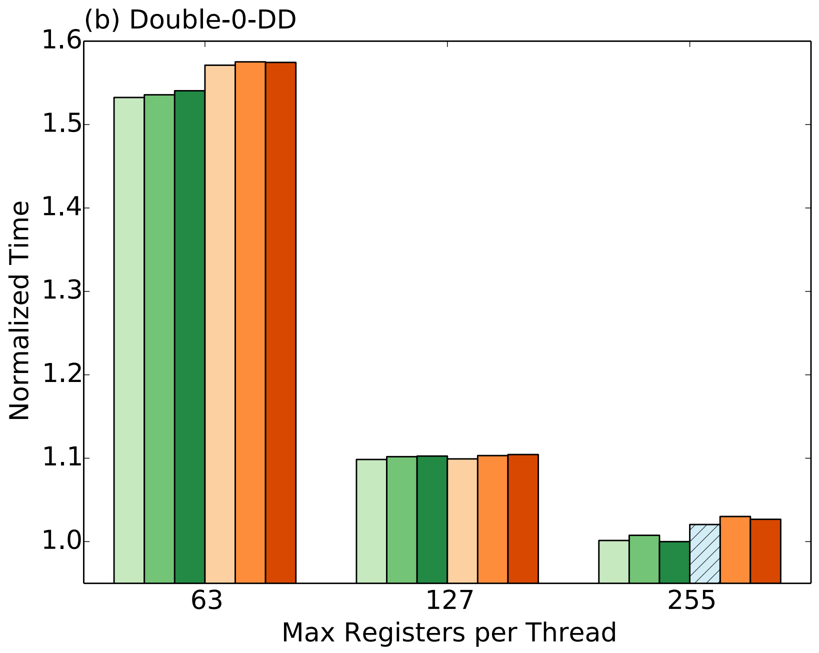


Figure 9. Performance comparison of different GPU configurations for two different kernels: (a) Double-0-SD and (b) Double-0-DD, both on CC3.5. Normalized time is computed with respect to the fastest configuration. Each comparison uses the code variant optimized under the default GPU configuration (blue), corresponding to: block size equal to 64, 16KB/48KB partition of L1/shared memory, and 225 registers per thread. In the legend, "L1" (resp. "Shared") denotes a 48/16 KB (resp. 16/48 KB) partitioning between L1 cache and shared memory. The second integer in the legend represents the block size.

References

1. McMurchie, L. E.; Davidson, E. R. One-electron and 2-electron integrals over Cartesian Gaussian functions. *J. Comp. Phys.* **1978**, *26*, 218.
2. Chiodo, S.; Russo, N. Determination of spin-orbit coupling contributions in the framework of density functional theory. *J. Comp. Chem.* **2008**, *29*, 912.
3. Helgaker, T.; Taylor, P. R. On the evaluation of derivatives of Gaussian integrals. *Theo. Chim. Acta* **1992**, *83*, 177.
4. Yasuda, K. Two-electron integral evaluation on the graphics processor unit. *J. Comp. Chem.* **2008**, *29*, 334.
5. Ufimtsev, I. S.; Martinez, T. J. Quantum chemistry on graphical processing units. 1. Strategies for two-electron integral evaluation. *J. Chem. Theo. Comp.* **2008**, *4*, 222.
6. Ufimtsev, I. S.; Martinez, T. J. Quantum Chemistry on Graphical Processing Units. 2. Direct Self-Consistent-Field Implementation. *J. Chem. Theo. Comp.* **2009**, *5*, 1004.
7. Ufimtsev, I. S.; Martinez, T. J. Quantum Chemistry on Graphical Processing Units. 3. Analytical Energy Gradients, Geometry Optimization, and First Principles Molecular Dynamics. *J. Chem. Theo. Comp.* **2009**, *5*, 2619.
8. Asadchev, A.; Allada, V.; Felder, J.; Bode, B. M.; Gordon, M. S.; Windus, T. L. Uncontracted Rys Quadrature Implementation of up to G Functions on Graphical Processing Units. *J. Chem. Theo. Comp.* **2010**, *6*, 696.
9. Miao, Y.; Merz, K. M., Jr. Acceleration of High Angular Momentum Electron Repulsion Integrals and Integral Derivatives on Graphics Processing Units. *J. Chem. Theo. Comp.* **2015**, *11*, 1449.
10. Appel, A. W.; Ginsburg, M. *Modern Compiler Implementation in C*. Cambridge University Press: Cambridge, UK, 2004.
11. Jones, H. W. Computer-generated formulas for overlap integrals of Slater-type orbitals. *Int. J. Quantum Chem.* **1980**, *18*, 709.
12. Jones, H. W. Computer-geenrated formulas for some 3-center molecular integrals over Slater-type orbitals. *Int. J. Quantum Chem.* **1983**, *23*, 953.
13. Bracken, P.; Bartlett, R. J. Calculation of Gaussian integrals using symbolic manipulation. *Int. J. Quantum Chem.* **1997**, *62*, 557.
14. Strange, R.; Manby, F. R.; Knowles, P. J. Automatic code generation in density functional theory. *Comp. Phys. Comm.* **2001**, *136*, 310.
15. Seidler, P.; Christiansen, O. Automatic derivation and evaluation of vibrational coupled cluster theory equations. *J. Chem. Phys.* **2009**, *131*, 15.
16. Janssen, C. L.; Schaefer, H. F. The automated solution of 2nd quantization equations with applications to the coupled cluster approach. *Theo. Chim. Acta* **1991**, *79*, 1.
17. MacLeod, M. K.; Shiozaki, T. Communication: Automatic code generation enables nuclear gradient computations for fully internally contracted multireference theory. *J. Chem. Phys.* **2015**, *142*.
18. Baumgartner, G.; Cociorva, D.; Bibireata, A.; Gao, X. Y.; Krishnamoorthy, S.; Krishnan, S.; Lam, C. C.; Lu, Q. D.; Sibiryakov, A.; Pitzer, R. M.; Sadayappan, P.; Bernholdt, D. E.; Choppella, V.; Hirata, S.; Ramanujam, J.; Nooijen, M.; Auer, A. Computer aided implementation of many-body methods: The tensor contraction engine. *Abstr. Pap. Amer. Chem. Soc.* **2003**, *226*, U303.

19. Hartono, A.; Lu, Q.; Henretty, T.; Krishnamoorthy, S.; Zhang, H.; Baumgartner, G.; Bernholdt, D. E.; Nooijen, M.; Pitzer, R.; Ramanujam, J.; Sadayappan, P. Performance Optimization of Tensor Contraction Expressions for Many-Body Methods in Quantum Chemistry. *J. Phys. Chem. A* **2009**, *113*, 12715.
20. Panyala, A. B., P.; Baumgartner, G.; Ramanujam, J. *Model-Driven Search-Based Loop Fusion Optimization for Handwritten Code*, Proceedings of the 17th Workshop on Compilers for Parallel Computers (CPC '13), Lyon, France, 2013.
21. Fermann, J. T.; Valeev, E. F. Libint: machine-generated library for efficient evaluation of molecular integrals over Gaussians. https://github.com/evaleev/libint **2003**.
22. Valeev, E. F.; Janssen, C. L. Second-order Moller-Plesset theory with linear R12 terms (MP2-R12) revisited: Auxiliary basis set method and massively parallel implementation. *J. Chem. Phys.* **2004**, *121*, 1214.
23. Whaley, R. C.; Dongarra, J. J. Automatically tuned linear algebra software. *Proceedings of ACM/IEEE SC98: 10th Anniversary. High Performance Networking and Computing Conference (Cat. No. RS00192)* **1998**, 33 pp.
24. Frigo, M.; Johnson, S. G. The design and implementation of FFTW3. *Proc. IEEE* **2005**, *93*, 216.
25. Whaley, R. C.; Petitet, A.; Dongarra, J. J. Automated empirical optimizations of software and the ATLAS project. *Par. Comp.* **2001**, *27*, 3.
26. Lu, Q.; Gao, X.; Krishnamoorthy, S.; Baumgartner, G.; Ramanujam, J.; Sadayappan, P. Empirical performance model-driven data layout optimization selection for tensor contraction expressions. *J. Par. Dist. Comp.* **2012**, *72*, 338.
27. Titov, A. V.; Ufimtsev, I. S.; Luehr, N.; Martinez, T. J. Generating Efficient Quantum Chemistry Codes for Novel Architectures. *J. Chem. Theo. Comp.* **2013**, *9*, 213.
28. McMurchie, L. E.; Davidson, E. R. Calculation of integrals over ab initio pseudopotentials. *J. Comp. Phys.* **1981**, *44*, 289.
29. Kahn, L. R.; Goddard, W. A. Ab-Initio effective potentials for use in molecular calculations. *J. Chem. Phys.* **1972**, *56*, 2685.
30. Song, C.; Wang, L.-P.; Sachse, T.; Preiss, J.; Presselt, M.; Martinez, T. J. Efficient Implementation of Effective Core Potential Integrals and Gradients on Graphical Processing Units (GPUs). *J. Chem. Phys.* **2015**, *143*, 014114.
31. Allen, F. E.; Cocke, J. Program data flow analysis procedure. *Comm. ACM* **1976**, *19*, 137.
32. Chaitin, G. J.; Auslander, M. A.; Chandra, A. K.; Cocke, J.; Hopkins, M. E.; Markstein, P. W. Register allocation via coloring. *Comp. Lang.* **1981**, *6*, 47.
33. Chaitin, G. J. Register allocation & spilling via graph coloring. *Acm Sigplan Notices* **2004**, *39*, 67.
34. As there are several types of graphs that we mention in this paper, we clarify that the figures and the graph structures in ACE consist only of dependence graphs and decision graphs; the interference graph is a guiding concept that helps to explain register spilling and estimate memory usage.
35. Dennis, J. B. First Version of a data flow procedure language. *Programming Symposium* **1974**, 362.
36. Dennis, J. B. Data flow supercomputers. *Computer* **1980**, *13*, 48.
37. Allen, F. E. Interprocedural analysis and the information derived by it. *Prog. Meth.* **1975**, 291.

38. Treleaven, P. C.; Hopkins, R. P.; Rautenbach, P. W. Combining data flow and control flow computing. *Computer Journal* **1982**, *25*, 207.
39. Ferrante, J.; Ottenstein, K. J.; Warren, J. D. The program dependence graph and its use in optimization. *ACM Trans. Prog. Lang.* **1987**, *9*, 319.
40. Ottenstein, K. J.; Ottenstein, L. M. The program dependence graph in a software-development environment. *Sigplan Notices* **1984**, *19*, 177.
41. The actual number of allocated registers reported by the compiler may be even higher than the number of interfering variables that we calculate here, since we do not explicitly reuse variable names in the generated code. This aspect will be addressed in future work.
42. CC stands for "compute capability." This is NVIDIA's designation of different GPU architectures.
43. Luehr, N.; Ufimtsev, I. S.; Martinez, T. J. Dynamic Precision for Electron Repulsion Integral Evaluation on Graphical Processing Units (GPUs). *J. Chem. Theo. Comp.* **2011**, *7*, 949.