Generation of Kernels for Calculating Electron Repulsion Integrals of High Angular Momentum Functions on GPUs – Preliminary Results

Alexey V. Titov, Volodymyr V. Kindratenko
National Center for Supercomputing Applications
University of Illinois at Urbana-Champaign
Urbana, IL, USA
Email: {atitov|kindrt}@ncsa.illinois.edu

Ivan S. Ufimtsev, Todd J. Martinez
Department of Chemistry
Stanford University
Palo Alto, CA, USA
Email: {ufimtsev|todd.martinez}@stanford.edu

***Abstract*—Evaluation of electron repulsion integrals (ERIs) takes considerable time in modern quantum chemistry applications and also presents a certain difficulty to be efficiently computed on GPUs. Here, we describe a novel methodology for generating high-arithmetic-density kernels for ERI evaluation of d and higher angular momentum functions, as well as highlight challenges associated with the efficient use of GPUs for ERIs. Employing this approach allows to perform the ERI evaluations substantially faster than just using traditional CPU codes.**

***Index Terms*—GPUs; quantum chemistry; gaussian integrals; d-functions**

I. Introduction

One of the most demanding components in ab initio quantum chemistry is the evaluation of electron repulsion integrals (ERIs). The development of quantum chemistry codes for graphics processing units (GPUs) has shown a remarkable potential for these problems [1]–[3]. In this work, we start from the existing code that implements the evaluation of integrals for s and p functions and extend it to the higher angular momentum functions [1]. Previously, all the GPU kernels have been hand-written and manually tuned for optimal performance. Stepping up to the integrals with d functions is not a trivial task with hand-written C code, because of involved mathematical formulations and a large number of integral batches. Here we discuss how the coding of ERI batches can be supplemented (or merely substituted) with a meta-code, which generates the GPU kernels for a specific ERI batch.

GPUs utilize stream processing architecture, where fine-grained parallelism and use of memory hierarchy is imperative in order to perform calculations efficiently. GPU threads are lightweight and exchange data through the shared or global memory. Data localization is required so that adjacent threads should access adjacent DRAM addresses to coalesce multithread memory instructions into one memory load or store operation. If these recommendations are not met, the overall memory bandwidth drops significantly, thus removing the benefit of using GPUs. The ratio of floating point operation (flop) and global memory operation (mop) execution times is around two orders of magnitude in the currently used GPUs (GF200 series) [4]. Therefore, it is also desirable to perform as many flops as possible using limited local memory storage (registers) and possibly avoiding global memory operations. It is also necessary to carefully orchestrate memory accesses to eliminate potential race conditions, that is, the relationships between threads and the memory addresses has to be carefully designed.

Currently, there are two major programming frameworks for GPU computing: CUDA and OpenCL. In our work, the code is written within CUDA C programming framework and executed on NVIDIA's GPUs.

II. Integrals Evaluation

A. Electron Repulsion Integrals

One of the most essential steps in ab initio Molecular Orbital theory is the calculation of 2-electron repulsion integrals over N atom-centered 1-electron basis functions ϕ

$$(ab|cd) = \iint \phi_a(\vec{r}_1)\phi_b(\vec{r}_1)\frac{1}{|\vec{r}_1 - \vec{r}_2|}\phi_c(\vec{r}_2)\phi_d(\vec{r}_2)d\vec{r}_1 d\vec{r}_2 \,, \tag{1}$$

where $\vec{r}$ describes electronic coordinates. Consequently, one electron functions ϕ could be approximated by a series of Gaussian-type functions, which have advantages for the integral evaluation. An Gaussian function on A-th atomic center is given by

$$\varphi(r_A) = x_A^n \, y_A^l \, z_A^m \, e^{-\alpha r_A^2} \,, \tag{2}$$

where (x_A, y_A, z_A) are the components of the vector $\vec{r}_A = \vec{r} - \vec{A}$ with norm r_A. The normalization of this function is given as a product

$$N_{nlm}(\alpha_A) = \prod_{k=n,l,m} N_k(\alpha_A)\,, \tag{3}$$

where

$$N_k(\alpha) = \frac{(2\alpha/\pi)^{1/4}\,(4\alpha)^{k/2}}{\sqrt{(2k-1)!!}}\,. \tag{4}$$

Functions in Eq. 2 are referred to as $s, p, d, \ldots$ functions when the sum of powers $L = n + l + m$ equals to $0, 1, 2, \ldots,$ respectively. Integrals over such Gaussian-type orbitals (GTO) can be evaluated analytically, as proposed by Boys [5]. The total number of integrals to be computed over GTOs is large: if the basis set of functions is K, then the number of integrals to be calculated scales as $\sim K^4/8$. Several integration schemes have been proposed to efficiently evaluate these quantities, including McMurchie-Davidson (MD) scheme [6], where Hermite GTOs and auxiliary functions are used for integral evaluation.

B. *McMurchie-Davidson Integration Scheme*

The core of MD methodology lies in the Hermite expansion of Cartesian GTOs and in the well-known Gaussian transformation allowing to convert four center integrals $(ab|cd)$ into the two center integrals $(P|Q)$. Omitting details, we will focus only on the most relevant steps for the discussion of our algorithm.

A prefactor of the x-component for transformed GTOs in Hermite expansion can be found as

$$x_A^{n_A} x_B^{n_B} = \sum_{N=0}^{n_A+n_B} d_N^{n_A,n_B} \Lambda_N(x_P, \alpha_P)\,, \tag{5}$$

where $\Lambda_j(x_P, \alpha_P) = \alpha_P^{j/2} H_j(\alpha_P^{1/2} x_P)$ is defined through the Hermite polynomial of j-th order. Similarly we can write the expressions for $y^{l_A} y^{l_B}$, cross-terms such as $x^{n_A} y^{l_B}$ and others. In Eq. 5, the coefficients $d_N^{n_A,n_B}$ are obtained via the well-known recursion relations for the Hermite polynomials [6]. The expansion in terms of Λ is useful, because it allows the transformed two-center Gaussian (of a vector P) to be written as a sum of derivatives with respect to coordinates of P and the derivatives can be taken outside of the integral.

The products of all possible Λ_N combinations can be obtained through the auxiliary functions R_{NLMj} based on the incomplete Gamma functions $F_j(T)$ [6].

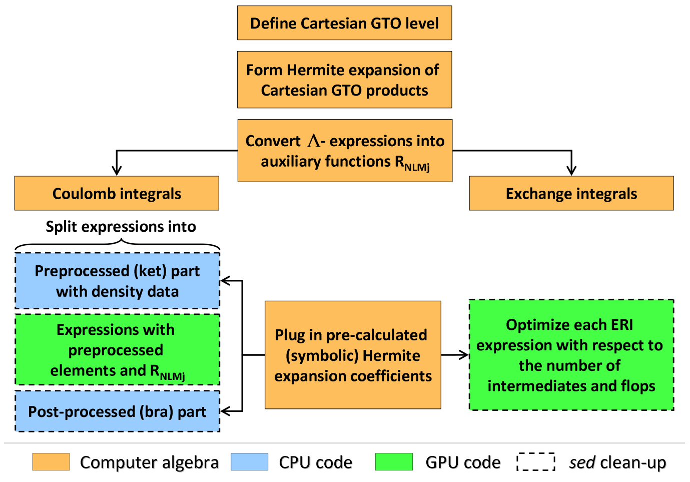
Define Cartesian GTO level
Form Hermite expansion of Cartesian GTO products
Convert Λ- expressions into auxiliary functions R_NLMj
Coulomb integrals
Exchange integrals
Split expressions into
Preprocessed (ket) part with density data
Expressions with preprocessed elements and R_NLMj
Post-processed (bra) 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

Fig. 1. Algorithm diagram for generating symbolic ERI expressions to use within MD 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. Coulomb integrals are calculated using J-engine method [8] which allows to offload some of the work to CPU (pre- and post-processing parts on the left).

III. Formation of ERI Batches

Generation of ERIs in the Hermite form requires a considerable number of series in terms of Eq. 5. It is possible to simplify the problem by using a computer algebra system with symbolic mathematics such as Maple [7]. We follow MD integration scheme, starting from the general form of Cartesian GTOs and converting all their possible combinations of products into the Hermite form defined in terms of Λ. For two d-orbitals, two-center Cartesian GTO product is expanded in 36 elements $5-9$ Λ-terms in each. Subsequently, all Λ-terms are expressed via auxiliary functions R_{NLMj} and the precalculated coefficients $d_N^{n_A,n_B}$ in a symbolic form are then substituted into these expressions. Finally, the obtained expressions are factored by the appropriate (precalculated) normalization coefficients from Eq. 3.

In Fig. 1 we illustrate the algorithm for generating ERI expressions. This figure depicts the flow of symbolic computations. In the final steps, algebraic expressions are exported from Maple to C using built-in porting procedure. The exported C code is then cleaned and wrapped with CUDA syntax using *sed* scripts. Exchange and coulomb integral analytical expressions are mapped according to the previously developed scheme [2]. The mapping “1 Thread – 1 Primitive Integral” and “1 Thread - 1 Matrix Element” has proven to be effective in the evaluation of s/p coulomb ERIs and exchange ERIs, respectively. This mapping works well in generating C code for GPU kernels with higher (d and other) angular momentum functions. Our preliminary estimates show that for coulomb integrals in the $(dd|dd)$ batch we can obtain approximately 20 GFlop/s. Here, the correct integrals are evaluted using mixed precision: single precision for separate integrals and double precision for their accumulation. Same-level ERIs are evaluated at ~ 0.79 GFlop/s (on a CPU) in the mature third-party quantum chemistry program GAMESS [9].

In a similar fashion, exchange integrals are treated. However, we encounter a problem with generated CUDA C code for integral batches involving four d-functions. Namely, the algebraic expressions converted into C code for evaluating integrals have too many instructions to be converted into GPU kernels and to be effectively break up into intermediates and effectively use available memory. In fact, the code produced with the direct generation of $(dd|dd)$ kernel can not be compiled with CUDA compiler. We evade this problem by splitting the $(dd|dd)$ batch into several kernels with the number of instructions and intermediates amenable to the CUDA compiler.

We observe an additional benefit for ERI analytic expressions while working with the computer algebra system. Namely, it allows a systematic division of expressions into arbitrary subexpressions of smaller size, thus providing a better control over the number of intermediates. Furthermore, each intermediate can be easily minimized with respect to the number of flops. Overall, it gives a precise control over the register usage and the arithmetic density of the GPU kernels, allowing much better balance on the rough GPU performance landscape.

The use of meta-coding for generating GPU kernels is justified by the considerable number and the large size of kernels. For example, coulomb $(pp|pp)$ kernel in the previous implementation takes 306 lines, while coulomb $(dd|dd)$ kernel is already 2094 lines of CUDA C code. The total number of kernels is 19 and 96 for s/p and $s/p/d$ implementation, respectively. Thus, symbolic computations allow us to rapidly generate a large number of arithmetically dense GPU kernels.

IV. Conclusion

We have developed a novel approach to efficiently deploy analytical ERI expressions to GPU kernels. Our initial results already demonstrate the flexibility with mapping to compute unit sizes and the benefits of this methodology.

Acknowledgment

Authors are grateful to Thom H. Dunning, Jr. for his support and guidance. This work was supported by the NFS grant CHE-0626354 and utilized the AC cluster [10] operated by the Innovative Systems Laboratory (ISL) at the National Center for Supercomputing Applications (NCSA) at the University of Illinois.

References

[1] I. S. Ufimtsev and T. J. Martinez, “Quantum chemistry on graphical processing units. 1. strategies for two-electron integral evaluation,” *J. Chem. Theory Comput.*, vol. 4, no. 2, pp. 222–231, Feb. 2008.

[2] I. S. Ufimtsev and T. J. Martinez, “Quantum chemistry on graphical processing units. 2. direct self-consistent-field implementation,” *J. Chem. Theory Comput.*, vol. 5, no. 4, pp. 1004–1015, Apr. 2009.

[3] A. Asadchev, V. Allada, J. Felder, B. M. Bode, M. S. Gordon, and T. L. Windus, “Uncontracted rys quadrature implementation of up to g functions on graphical processing units,” *J. Chem. Theory Comput.*, vol. 6, no. 3, pp. 696–704, Mar. 2010.

[4] NVIDIA CUDA. Compute Unified Device Architecture Programming Guide Version 3.0; NVIDIA Developer Web Site.

[5] S. F. Boys, “Electronic wave functions. i. a general method of calculation for the stationary states of any molecular system,” *Proc. R. Soc. London, Ser. A*, vol. 200, no. 1063, pp. 542–554, Feb. 22, 1950.

[6] L. E. Mcmurchie and E. R. Davidson, “One-electron and 2-electron integrals over cartesian gaussian functions,” *J. Comp. Phys.*, vol. 26, no. 2, pp. 218–231, 1978.

[7] Maple 13.02 by Waterloo Maple Inc. http://www.maplesoft.com/.

[8] G. R. Ahmadi and J. Almlof, “The coulomb operator in a gaussian product basis,” *Chem. Phys. Lett.*, vol. 246, no. 4-5, pp. 364–370, Dec. 1995.

[9] M. W. Schmidt, K. K. Baldridge, J. A. Boatz, S. T. Elbert, M. S. Gordon, J. H. Jensen, S. Koseki, N. Matsunaga, K. A. Nguyen, S. J. Su, T. L. Windus, M. Dupuis, and J. A. Montgomery, “General atomic and molecular electronic-structure system,” *J. Comput. Chem.*, vol. 14, no. 11, pp. 1347–1363, Nov. 1993.

[10] V. Kindratenko, J. Enos, G. Shi, M. Showerman, G. Arnold, J. Stone, J. Phillips, and W. mei Hwu, “Gpu clusters for high-performance computing,” in *Cluster Computing and Workshops, 2009. CLUSTER ’09. IEEE International Conference on*, Aug. 2009, pp. 1–8.