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

Alexey Titov, Volodymyr Kindratenko
Innovative Systems Laboratory, NCSA/UIUC

Ivan Ufimtsev, Todd Martinez
Department of Chemistry, Stanford University
**General problem statement**

Numerically solve stationary Schrödinger equation (SE) for arbitrary large molecular systems ($1-10^3$ atoms, days-weeks of runtime) to obtain energy $E$ and other molecular properties (based on $\psi \Psi$)

$$H\Psi(r) = E\Psi(r)$$

- $3$ atoms $\rightarrow 10^e$, $30$ dimensions
- Antisymmetry of $\Psi(r_i)$ required:
  $$\Psi(\ldots, r_i, \ldots, r_j, \ldots) = -\Psi(\ldots, r_j, \ldots, r_i, \ldots)$$
  $\rightarrow 3,628,000$ permutations

Employ Hartree-Fock approximation for $\Psi$:

$$\Psi = A[\psi_1(r_1)\psi_2(r_2)\ldots\psi_N(r_N)]$$
Each 1e orbital $\psi$ is expanded over a predefined set of basis functions $\varphi$.

$$\psi_i (r) = \sum_{j=1}^{K} C_{ij} \varphi_j (r)$$

SE is then solved self-consistently until $C_{k+1} - C_k \approx 0$

$$F(C_k) = E_{SC_k}$$

$$F_{k+1}(C) = F(C_k)$$

$$F_{k+1} C_{k+1} = E_{SC_{k+1}}$$

Approximations and solution strategy

Matrices $N \times N$ ($N \sim 10^3 - 10^4$)

O($N^3$)

$$F_{ij} (C) = H_{ij}^{\text{core}} + J_{ij} (C) - \frac{1}{2} K_{ij} (C)$$

Coulomb ERIs

$$J_{ij} = \sum_{k,l} (ij \mid kl) P_{kl} (C)$$

$$K_{ij} = \sum_{k,l} (ik \mid jl) P_{kl} (C)$$

Exchange ERIs

O($N^4$)
where $\varphi$ is given as an atom-centered Cartesian Gaussian function (GTO)

$$\varphi(r_A) = x^n y^l z^m \cdot e^{-\alpha \cdot r_A^2}, \quad \vec{r}_A = \vec{r} - \vec{A}$$

functions $\varphi$ are referred as $s,p,d,..$ functions if $n+l+m$ equals to $0,1,2,..$

Such integrals could be evaluated analytically using Gaussian Product Theorem and Boys (1950):

An ERI $(ab|cd)$ over $s$ functions can be evaluated as

$$\frac{2\pi^{5/2}}{\gamma} \cdot e^{-\kappa} \cdot F_0(T)$$

$F_0(T)$ – 0-th order incomplete Gamma function (erf)
McMurchie-Davidson ERI scheme (1/3)

• Expand two-center Cartesian Gaussians \( \varphi_P(r) = \varphi_A(r) \times \varphi_B(r) \) over Hermite Gaussian basis using coefficients

\[
x_A^{n_A} x_B^{n_B} = \sum_{N=0}^{n_A+n_B} d_{N}^{n_A,n_B} \Lambda_N(x_P; \alpha_P), \quad \Lambda_j(x_P; \alpha_P) = \alpha_P^{j/2} H_j(\alpha_P^{1/2} x_P)
\]

similarly for \( y \) and \( z \) coordinates.

• Once integrals are evaluated – transform back to Cartesian Gaussians
• Coefficients \( d_{N}^{n_A,n_B} \) are evaluated through the known recurrence relations for Hermite polynomials now expressed via

\[
x_A \Lambda_N(x_P; \alpha_P) = N \Lambda_{N-1} + P A_x \Lambda_N + \frac{1}{2 \alpha_P} \Lambda_{N+1}
\]
McMurchie-Davidson ERI scheme (2/3)

- The basic integral \((ab|cd)\) takes the form

\[
[NLM | \theta | N'L'M'] = \int \int \theta(r_1, r_2) \Lambda_N(x_{1p}; \alpha_P) \Lambda_L(y_{1p}; \alpha_P) \Lambda_M(z_{1p}; \alpha_P) \\
\times \exp(-\alpha_P r_{1p}^2) \Lambda_{N'}(x_{1q}; \alpha_Q) \Lambda_{L'}(y_{1q}; \alpha_Q) \Lambda_{M'}(z_{1q}; \alpha_Q) \\
\times \exp(-\alpha_Q r_{1q}^2) d\tau_1 d\tau_2
\]

- We denote \([NLM|\theta|N’L’M’]\) as an auxiliary function \(R_{N+N'L+L'M+M'}^{0}\)
- Functions \(R_{000}^j(T)\) are evaluated via incomplete Gamma functions \(F_j(T)\)
McMurchie-Davidson ERI scheme (3/3)

**Batch (ss|sp): 1 element with 4 different terms**

\[
(D_{0,0} \cdot QD_x + D_{0,1} \cdot QD_y + D_{0,2} \cdot QD_z) \cdot [R^0_{000} + (2\alpha_Q)^{-1} \cdot D_{0,0} \cdot R^0_{100} + (2\alpha_Q)^{-1} \cdot D_{0,1} \cdot R^0_{010} + (2\alpha_Q)^{-1} \cdot D_{0,2} \cdot R^0_{001}]
\]

**Batch (ss|pp): 1 element with 10 different terms**

\[
P_0 \cdot R^0_{000} + P_1 \cdot R^0_{100} + P_2 \cdot R^0_{010} + P_3 \cdot R^0_{001} + P_4 \cdot R^0_{110} + P_5 \cdot R^0_{101} + P_6 \cdot R^0_{011} + P_7 \cdot R^0_{200} + P_8 \cdot R^0_{020} + P_9 \cdot R^0_{002}
\]

**Batch (sp|sp): 3 elements with 10 different terms each**

\[
\ldots \times 6
\]

**Batch (ss|sd): 1 element with 10 different terms**

\[
\ldots \times 23
\]

**Batch (dd|dd): 36 elements with 35 different terms each**
Why GPUs – 5 reasons

- Massively parallel – more than 10,000 GPU threads in parallel
- Simple kernels (min memory usage) – a lot of threads to hide memory latency
- High FLOP/MOP ratio
- Little or no communication between threads (communication within the block – efficient data sharing)
- Interpolation via polynomial functions is ubiquitous in computer graphics (e.g. splines – piecewise polynomial functions)

\[ Y_i(t) = a_i + b_i \cdot t + c_i \cdot t^2 + d_i \cdot t^3 \]

from wolfram.com
• Based on the earlier implementation*:
  \( J: \text{1PI} \rightarrow \text{1T} \quad K: \text{1ME} \rightarrow \text{1TB} \quad \text{OEI: on CPU} \)

2e integral grid

\[
[ij \mid kl] \leq \sqrt{|ij|} \cdot \sqrt{|kl|} \geq 10^{-11}
\]
leaves only \( N^2 \) out of \( N^4 \) integrals

\[
[kl]
\]

Original integral grid

Presorted integral grid

SIMD warp

Most negligibly small integrals will be calculated

Only significant integrals will be calculated

Courtesy of Todd Martinez and Ivan Ufimtsev

*Ufimtsev & Martinez, *JCTC* 2009, 5, p. 3138*
**J-matrix implementation**

\[ J_{ij} = \sum_{k,l} (ij \mid kl) P_{kl} \]

\[ [ij \mid kl] \leq \sqrt{[ij \mid ij]} \cdot \sqrt{[kl \mid kl]} \]

**Presorted ket-pairwise quantities \([q]\)**

**Presorted bra-pairwise quantities \([p]\)**

**[p|q] Hermite primitive integral grid**

Each row leads to one \(J_{[p]}\) matrix element

**Block 2, calculates \(J_{[2]}\) and \(J_{[4]}\)**

Final row-wise reduction

Thank you to Todd Martinez and Ivan Ufimtsev for their contributions.
Problem statement

Addition of d-functions leads to increase in the number of ERI batches (i.e. kernels):

- $J$: 9 → 36
- $K$: 10 → 45

A clear need to *automate* kernel generation and optimization.

Meta-code dealing with $J$, $K$ and one-electron matrices:

```
Maple → sed → C/CUDA C
```

- Symbolic evaluation of matrix elements and ERI classes
- Exported code clean-up and embedding into the kernel structure
Algebraic derivations for source code generation

1) Form systems of linear equations from recurrence relations and analytically solve for expansion coefficients $d_{N}^{n_{A},n_{B}}$ (using standard computer algebra tools)

One unique solution → 6 sets of coefficients

\[
\begin{align*}
    d_{0,0}^{1,0} &= PA_x \\
    d_{0,1}^{0,1} &= PB_x \\
    d_{1,0}^{1,0} &= (2\alpha_{P})^{-1} \\
    d_{3,2}^{2,2} &= (4\alpha_{P}^{3})^{-1} \cdot (PA_x + PB_x) \\
    d_{4,2}^{2,2} &= (2\alpha_{P})^{-4}
\end{align*}
\]

\{ \times 27 \) non-zero coefficients

2) Derive each ERI class within $R_{NLM}^{0}$ framework and split final expressions according the chosen computation strategy (i.e. CPU/GPU/CPU, GPU only, etc.)

\[
(D_{0,0} \cdot QD_{x} + D_{0,1} \cdot QD_{y} + D_{0,2} \cdot QD_{z}) \cdot R_{000}^{0} + (2\alpha_{Q})^{-1} \cdot D_{0,0} \cdot R_{100}^{0} \\
+ (2\alpha_{Q})^{-1} \cdot D_{0,1} \cdot R_{010}^{0} + (2\alpha_{Q})^{-1} \cdot D_{0,2} \cdot R_{001}^{0}
\]

$P_0, P_1, P_2, P_3 \times R0000, R0010, R0100, R1000$
Define Cartesian GTO level

Form Hermite expansion of Cartesian GTO products

Convert $\Lambda$-expressions into auxiliary functions $R_{\text{NL}Mj}$

Coulomb integrals

Exchange integrals

Split expressions into

Preprocessed (ket) part with density data

Expressions with preprocessed elements and $R_{\text{NL}Mj}$

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
A simple example passing through the generation pipeline

SSSP

\[ ssp := \text{subs}(n_A = 0, n_B = 0, l_A = 0, l_B = 0) \]

\[ \left( \sum_{N=0}^{0} \sum_{L=0}^{0} d_{0,1,N} A_{x,N}(Q) \right) \left( \sum_{L=0}^{0} e_{0,0,L} A_{y,L}(Q) \right) \]

\[ f_{0,0,M} A_{x,M}(Q) \cdot D_{0,1} + \left( \sum_{N=0}^{0} d_{0,1} \right) \]

\[ J_i \text{igen} := J_i \text{effCD}, e_{\text{coeffC}} \]

\[ \Lambda x_0(P) A y_0(P) A z_0(P) A x_0(Q) + \Lambda x_0(P) A y_0(P) A z_0(P) A x_0(Q) \]

\[ + \Lambda x_0(P) A y_0(P) A z_0(P) A x_0(Q) \]

\[ + \Lambda x_0(P) A y_0(P) A z_0(P) A x_0(Q) \]

\[ ssp := \text{expproc}(J_i \text{igen}, 0, 1) \]

\[ (D_{0,2} Q D_x + D_{0,1} Q D_y) \cdot \]

\[ \text{extractP}(sssp, 1); \]

\[ [D_{0,2} \ldots \ldots \ldots \alpha_Q \ldots \alpha_Q \ldots \alpha_Q] \]

\[ \text{subsP}(sssp, \%_0, 0, 1); \]

\[ P0 R0000 + P0 R1000 + P0 R0100 + P0 R0010 \]
for( [bra • ket] > ε) {
   // load data
   DP_Gamma8(…)
   // calculate a, b, c and auxiliary functions R000j
   float R0010 = c * R0001;
   ...
   float R3000 = a * R2001 + 2.0f * R1001;
   ...
   float R0080 = c * R0071 + 7.0f * R0061;
   ...
   float P0 = tex1Dfetch(tex_preproP, g_thidX);
   tmp0  += R0000 * P0;
   tmp1  += R1000 * P0;
   ...
   tmp34 += R0040 * P0;
   ...
   float P34 = tex1Dfetch(tex_preproP, g_thidX +
   __mul24(ne, 34));
   ...
   // accumulate tmps in DP
}
// collect integrals and upload to global memory
Preliminary results*

- **Kernel issues**
  - Kernel size (unopt K kernels are too big to compile)
  - nvcc bugs (related to size)
  - Accuracy (SP is not enough)

- **Generate correct kernels involving s, p and d functions**
  - Mixed precision used on GPU:
    - DP for evaluating Gamma functions and accumulation of integrals,
    - SP – everywhere else
  - Obtained accuracy for K, J matrices: 6-7 digits
  - Resulting accuracy for the SCF energy: 6 digits
  - Measured calc speed (ddddd batch): ~ 66.6 GFlops/sec
Debunking the 100X GPU vs. CPU Myth: An Evaluation of Throughput Computing on CPU and GPU

Victor W Lee†, Changkyu Kim†, Jatin Chhugani†, Michael Deisher†, Daehyun Kim†, Anthony D. Nguyen†, Nadathur Satish†, Mikhail Smelyanskiy†, Srinivas Chennupaty*, Per Hammarlund*, Ronak Singhal* and Pradeep Dubey†

victor.w.lee@intel.com

†Throughput Computing Lab, Intel Corporation
*Intel Architecture Group, Intel Corporation
... for( [bra • ket] > \varepsilon )
{
    // load data
    DP_Gamma8(…)
    // calculate a,b,c and auxiliary functions R000j

    volatile float R0010 = c * R0001;
    ...
    volatile float R3000 = a * R2001 + 2.0f * R1001;
    ...
    volatile float R0080 = c * R0071 + 7.0f * R0061;

    float P0 = tex1Dfetch(tex_preproP, g_thidX);
    tmp0 += R0000 * P0;
    tmp1 += R1000 * P0;
    ...
    tmp34 += R0040 * P0;

    float P34 = tex1Dfetch(tex_preproP, g_thidX + __mul24(ne, 34));
    ...

    // accumulate tmps in DP
}
// collect integrals and upload to global memory

Bytes per thread: 1880
(122 in regs)

Bytes per thread: 1322

\quad 1880 \times 35 = 1300

486 lines

35 lines

× 35 = 1300

35 lines

486 lines
Conclusions

• Extending existing computational schemes can be done with meta-codes: it might be advantageous to use developments in neighboring fields (computer algebra systems)

• Working with abstract algebraic expression provides a flexibility in the actual C code breakdown

• Mixed precision is here to stay, but the results should be always tested for correctness/stability

Work in progress/Future directions

• Wrap to actual SCF procedure

• Optimize kernels (algebraic and CUDA levels)
Acknowledgements

NSF for funding the research

Thank you for your attention!
Questions?