Accelerating Quantum Chemistry Research using GPUs – Two Electron Integrals in GAMESS

Andrey Asadchev
Jacob Felder
Veerendra Allada
Dr. Mark S Gordon
Dr. Theresa Windus
Dr. Brett Bode

GPU Technology Conference, NVIDIA, San Jose, 2009
Outline

- Computational Quantum Chemistry
- General Atomic Molecular Electronic Structure Systems - GAMESS
- Electron Repulsion Integral (ERI) Problem
- Our Approach
  - CUDA Implementation
  - Optimizations
  - Automatically generated code
- Performance Results
- Future Goals
- Questions & Discussion
Computational Quantum Chemistry

- Use computational methods to solve the electronic structure and properties of molecules.
- Finds utility in the design of new drugs and materials
- Underlying theory is based on Quantum Mechanics – Schrodinger wave equation
- Properties calculated
  - Energies
  - Electronic charge distribution
  - Dipole moments, vibrational frequencies.
- Methods employed
  - Ab initio Methods (Solve from first principles)
  - Density Functional Theory (DFT)
  - Semi-empirical methods
  - Molecular Mechanics (MM)
General Atomic Molecular Electronic Structure System (GAMESS)

- **Ab initio** molecular quantum chemistry software
- USDOE “SciDAC Basic Energy Sciences” (BES) application
- Serial and parallel versions for several methods
- In brief, GAMESS can compute
  - Self Consistent Field (SCF) wave functions - RHF, ROHF, UHF, GVB, and MCSCF using the Hartree-Fock method
  - Correlation corrections to SCF using configuration interaction (CI), second order perturbation theory, and coupled cluster theories (CC)
  - Density Functional Theory approximations

Background

- Molecules are made of atoms and atoms have electrons
- Electrons live in shells – s, p, d, f, g, h
- Shells are made of sub-shells – all have the same angular momentum (L)
- Shells are represented using the mathematical functions
  - Gaussian functions are taken as standard primitive functions (S.F. Boys)
  - \( \varphi(r) = x^{a_x} y^{a_y} z^{a_z} \exp(-\alpha r^2) \)
  - \( x, y, z \) – Cartesian center
  - \( a_x, a_y, a_z \) – Angular momenta components; \( L = a_x + a_y + a_z \)
  - \( \alpha \) is the exponent
  - Shells with low angular momentum are typically contracted
    - \( \phi_a(r) = \sum_k D_{ka} \varphi_k(r) \)
    - K is the contraction coefficient. \( D_k \)'s are the contraction coefficients
Molecule Specification
- List of Atoms (Atomic Numbers Z)
- List of Nuclear Coordinates (R)
- Number of electrons
- List of Primitive Functions, exponents
- Number of contractions

Form the basis functions (M)

\( H_{\text{core}} \)
(one-electron integrals)
Kinetic Energy Integrals (T)
Nuclear Attraction Integrals (V)

Initial guess of the wave function
Obtain the guess at the Density Matrix (P)
\( O(M^2) \)

Form the Fock Matrix
\( F = H_{\text{core}} + G \)
\( G = [(ij|kl) - \frac{1}{2}(ik|jl)]*P \)
\( O(M^2) \)

Transformations
\( F' = X'FX \)
\( C' \leftarrow \text{Diagonalize}(F') \)
\( C \leftarrow XC' \)

Convergence Checks
Repeat steps 3, 4, 5, 6, 7

Overview of Hartree-Fock Algorithm
Reference: Szabo and Ostlund
Modern Quantum Chemistry
Four-center two-electron repulsion integral

\[(ab|cd) = \int \int \varphi_a(1) \varphi_b(1) \frac{1}{r_{12}} \varphi_c(2) \varphi_d(2)\]

Major computational step in both \textit{Ab initio} and DFT methods

Complexity is \(O(M^3) - O(M^4)\), \(M\) is the number of basis functions (Gaussian functions are standard)

Rys Quadrature – proposed by Dupius, Rys, King (DRK)

- Numerical Gaussian quadrature based on a set of orthogonal Rys polynomials
- Numerically stable, low memory footprint
- Amenable for GPUs and architectures with smaller caches
Two electron integral is expressed as 

\[(ij|kl) = \sum_{m=0}^{L} C_m F_m(X)\]

where \( F_m(X) = \int_{0}^{\frac{1}{2}} t^{2m} \exp(-X t^2) dt \) and \( L = L_a + L_b + L_c + L_d \)

\( X \) depends on exponents, centers and is independent of angular momenta

\[
X = \rho (r_A - r_B)^2 \\
r_A = (\alpha_i r_i + \alpha_j r_j) / A \\
r_B = (\alpha_k r_k + \alpha_l r_l) / B
\]

\( \rho = AB / (A + B) \)

\( A = \alpha_i + \alpha_j \)

\( B = \alpha_k + \alpha_l \)

\[(ij|kl) = \int_{0}^{1} \exp(-X t^2) P_L(t) dt \]

, where \( P_L(t) \) is polynomial of degree \( L \) in \( t^2 \). Evaluated using \( N \)-point quadrature and hence

\[(ij|kl) = \sum_{\omega=1}^{N} W_{\omega} P_L(t_{\omega}) \]

where \( N = \frac{L}{2} + 1 \)

Using separation of variables, \( P_L(t) \) which is integral over \( dr_1 dr_2 \), can be written as a product of three (2-D) integrals over \( dx_1 dx_2, dr_1 dr_2, dz_1 dz_2 \)

\[(ij|kl) = 2(\rho / \pi)^{L/2} \sum_{\omega} I_x(t_{\omega}) I_y(t_{\omega}) I_z(t_{\omega}) W_{\omega} \quad \text{and} \quad I_{q(i=x,y,z)}(N,0:L_{a},0:L_{b},0:L_{c},0:L_{d}) \]

\( I_x, I_y, I_z \) are computed using recurrence and transfer relations
Rys Quadrature Algorithm

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

- Summation over the roots over all the intermediate 2-D integrals
- floating point operations = $3N(L_a + 1)(L_b + 1)(L_c + 1)(L_d + 1)$
- Recurrence, transfer and roots have predictable memory access patterns, fewer flops. Quadrature step is the main focus here.
Rys Quadrature Algorithm

- Example: (dd|dd) ERI block
  - $L_a = L_b = L_c = L_d = 2$
  - Number of roots, $N = 5$
  - ERI size = $6^4 = 1296$ elements
  - Intermediate 2-D integrals $I_x, I_y, I_z$ size: $3^4 \times 5 = 245$

Possible Optimizations
- ERI computations are memory bound, hence optimize memory accesses
- Intermediate 2-D integrals are reused multiple times to construct different ERI elements.
- Generate the different combinations automatically
CUDA – Compute Unified Device Architecture (Birds eye-view)

Symmetric Multiprocessor N

Symmetric Multiprocessor 2

Symmetric Multiprocessor 1

Multithreaded Instruction Unit

SP1
Registers

SP2
Registers

SP3
Registers

SP4
Registers

SFU

SP5
Registers

SP6
Registers

SP7
Registers

SP8
Registers

SFU

DP unit

Shared Memory

Constant Cache

Texture Cache

Device Memory

SM – Streaming Multiprocessor
SP – Scalar Processor Core
SFU – Special Functional Unit
DP – Double Precision Unit

Grid of Blocks

Block (0,0)  Block (1,0)  Block (2,0)  Block (0,1)  Block (1,1)  Block (2,1)

Thread (0,0)  Thread (1,0)  Thread (2,0)  Thread (3,0)  Thread (4,0)  Thread (0,1)  Thread (1,1)  Thread (2,1)  Thread (3,1)  Thread (4,1)  Thread (0,2)  Thread (1,2)  Thread (2,2)  Thread (3,2)  Thread (4,2)
Since 2-D integrals are reused multiple times, load them into shared memory

- However, shared memory access, synchronization limited to thread block boundaries
- ERI block should be mapped onto a single thread block
- Is it possible to map all the ERI elements to individual threads in a block?
- The answer depends on the ERI block under consideration

For a (dd|dd) ERI block, ERI size $= 6^4 = 1296$ elements

- Maximum of 512 or 768 threads per block
- Map $i, j, k$ indices corresponding to the three shells of the block to unique threads and iterate over the $l$ index
- Thread blocks are three dimensional, the mapping of $i, j, k$ is natural

For (ff|ff) ERI block, ERI size $= 10^4 = 1000$ elements

- Map $i, j$ indices corresponding to the first two shells of the block to unique threads and iterate over the $l$ index
CUDA Rys quadrature: i, j, k mapping

# map threads to ERI elements
I = threadIdx.x, j = threadIdx.y, k = threadIdx.z

# arrays LX, LY, LZ map functions to exponents
(ix, iy, iz) ← (LX[i], LY[i], LZ[i])
(jx, jy, jz) ← (LX[j], LY[j], LZ[j])
(kx, ky, kz) ← (LX[k], LY[k], LZ[k])

for all l do
    syncthreads
    ## load the 2-D integrals to shmem
    Ix, shmem ← lx(:,:,LX[l])
    Iy, shmem ← ly(:,:,LX[l])
    Iz, shmem ← lz(:,:,LX[l])
    syncthreads

    l(i, j, k, l) ← ∑N Ix,shmemIy,shmemIz,shmem
end for

Further optimizations

- (dd|dd) case
- l_{ix,y,z}^{shmem} = 5(3^3) = 135 elements per 2-D block
- Across iterations, some of the elements in shared memory can be reused

**d-shell**

dx^2, dy^2, dz^2, dxy, dxz, dyz → 18 loads
dx^2, dy^2, dz^2, dxy, dxz, dyz → 13 loads

<table>
<thead>
<tr>
<th>l_x</th>
<th>0*</th>
<th>0</th>
<th>0</th>
<th>1*</th>
<th>1</th>
<th>2*</th>
</tr>
</thead>
<tbody>
<tr>
<td>l_y</td>
<td>2*</td>
<td>0*</td>
<td>1*</td>
<td>1</td>
<td>0*</td>
<td>0</td>
</tr>
<tr>
<td>l_z</td>
<td>0*</td>
<td>2*</td>
<td>1*</td>
<td>0*</td>
<td>1*</td>
<td>0*</td>
</tr>
</tbody>
</table>
CUDA Rys quadrature: i, j mapping

# map threads to ERI elements
I = threadIdx.x, j = threadIdx.y

# arrays LX, LY, LZ map functions to exponents
(ix, iy, iz) \leftarrow (LX[i], LY[i], LZ[i])
(jx, jy, jz) \leftarrow (LX[j], LY[j], LZ[j])

for all \(kl_z\)-block do
  syncthreads
  \(I_{z,shmem} \leftarrow I_z(:,;LZ[k],LZ[l])\)
  ## load 2-D integrals to shmem

for all \(kl_{xy}\) \(kl_z\)-block do
  syncthreads
  \(I_{x,shmem} \leftarrow I_x(:,;LX[k],LX[l])\)
  \(I_{y,shmem} \leftarrow I_y(:,;LY[k],LX[l])\)
  syncthreads
  \(I(i, j, k, l) \leftarrow \sum_{N} I_{x,shmem} I_{y,shmem} I_{z,shmem}\)
end for
end for

Further optimizations

- (ff|ff) case
- \(I_{ix,y,z,shmem} = 7(4^2) = 112\) elements per 2-D block
- 10 functions in the f-shell
- Reorder them (next slide)
<table>
<thead>
<tr>
<th>X</th>
<th>Y</th>
<th>Z</th>
</tr>
</thead>
<tbody>
<tr>
<td>3 0 0 2 2 1 1 1 0 0</td>
<td>0 3 0 1 0 2 0 1 2 1</td>
<td>0 0 3 0 1 0 2 1 1 2</td>
</tr>
</tbody>
</table>

\[ f_x^3, f_y^3, f_z^3, f_{xy}, f_{xz}, f_{xy}, f_{x^2}, f_{y^2}, f_{xyz}, f_{yz}, f_{y^2}, f_{z^3} \]

<table>
<thead>
<tr>
<th>X</th>
<th>Y</th>
<th>Z</th>
</tr>
</thead>
<tbody>
<tr>
<td>3 2 1 0 0 1 2 1 0 0</td>
<td>0 1 2 3 2 1 0 0 1 0</td>
<td>0 0 0 0 1 1 1 2 2 3</td>
</tr>
</tbody>
</table>

\[ f_x^3, f_x^2, f_y^3, f_y^2, f_{y^2}, f_{xyz}, f_{y^3}, f_{xz}, f_{z^2}, f_{yz}, f_z^3 \]
Automatic Code Generation based on Templates
Python – Cheetah

Warning: Template engine but not C++ templates

- Number of registers per thread, shared memory per thread block limits the thread blocks that can be assigned per SM
- Loops implemented directly result in high register usage
- Explicitly unroll the loops. How? Manually it’s tedious and error-prone
- Use a common template and generate all the cases
- Python based Cheetah template engine is used- reuse existing Python utilities and program support modules easily.
### Performance Results – Evaluated using the GeForce GTX 275

<table>
<thead>
<tr>
<th>ERI</th>
<th>blocks</th>
<th>flop count</th>
<th>$\text{GFLOPS}_{\text{SP}}^3$</th>
<th>$\text{GFLOPS}_{\text{DP}}^4$</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td>$\text{map}^5ijk$</td>
<td>$\text{map}^5ij$</td>
</tr>
<tr>
<td>(gg</td>
<td>gg)</td>
<td>2000</td>
<td>2733750000</td>
<td>n/a</td>
</tr>
<tr>
<td>(gg</td>
<td>ff)</td>
<td>4000</td>
<td>2160000000</td>
<td>n/a</td>
</tr>
<tr>
<td>(ff</td>
<td>gg)</td>
<td>4000</td>
<td>2160000000</td>
<td>n/a</td>
</tr>
<tr>
<td>(gg</td>
<td>dd)</td>
<td>10000</td>
<td>1701000000</td>
<td>n/a</td>
</tr>
<tr>
<td>(gg</td>
<td>pp)</td>
<td>40000</td>
<td>1458000000</td>
<td>n/a</td>
</tr>
<tr>
<td>(pp</td>
<td>gg)</td>
<td>40000</td>
<td>1458000000</td>
<td>34.23</td>
</tr>
<tr>
<td>(ff</td>
<td>ff)</td>
<td>10000</td>
<td>2100000000</td>
<td>n/a</td>
</tr>
<tr>
<td>(ff</td>
<td>dd)</td>
<td>20000</td>
<td>1296000000</td>
<td>n/a</td>
</tr>
<tr>
<td>(dd</td>
<td>ff)</td>
<td>20000</td>
<td>1296000000</td>
<td>37.69</td>
</tr>
<tr>
<td>(ff</td>
<td>pp)</td>
<td>80000</td>
<td>1080000000</td>
<td>27.43</td>
</tr>
<tr>
<td>(pp</td>
<td>ff)</td>
<td>80000</td>
<td>1080000000</td>
<td>32.23</td>
</tr>
<tr>
<td>(dd</td>
<td>dd)</td>
<td>60000</td>
<td>1166400000</td>
<td>31.10</td>
</tr>
</tbody>
</table>

- ERIs with odd number of roots have maximum performance over the even roots
  - Odd roots - (gg|gg), (gg|dd), and (ff|ff) cases
  - Even roots – (ff|gg), (gg|ff), and (dd|gg)
- The difference is as high as 25%
- Difference in the single and double precision is roughly a factor of two
- Larger $ijk$ mapping perform better than the $ij$ mappings
## Inferences

- Performance depends on the ERI class under evaluation and hence also on the mapping \((i,j,k)\ vs. \(i,j)\).
- Difference between single and double precision performance is roughly a factor of two.
- Difference between the GTX and Tesla T is roughly 30% (consistent with the clock speeds).
- In terms of register and shared memory usage both are identical.

### GTX 275 and Tesla Performance Comparison

<table>
<thead>
<tr>
<th>ERI</th>
<th>blocks</th>
<th>flop count</th>
<th>(\text{GFLOPS}_{\text{SP}}^3)</th>
<th>(\text{GFLOPS}_{\text{DP}}^4)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td><strong>GTX 275</strong></td>
<td><strong>Tesla</strong></td>
</tr>
<tr>
<td>((gg</td>
<td>gg))</td>
<td>2000</td>
<td>2733750000</td>
<td>45.23</td>
</tr>
<tr>
<td>((gg</td>
<td>ff))</td>
<td>4000</td>
<td>2160000000</td>
<td>34.42</td>
</tr>
<tr>
<td>((ff</td>
<td>gg))</td>
<td>4000</td>
<td>2160000000</td>
<td>30.91</td>
</tr>
<tr>
<td>((gg</td>
<td>dd))</td>
<td>10000</td>
<td>1701000000</td>
<td>43.08</td>
</tr>
<tr>
<td>((dd</td>
<td>gg))</td>
<td>10000</td>
<td>1701000000</td>
<td>23.63</td>
</tr>
<tr>
<td>((gg</td>
<td>pp))</td>
<td>40000</td>
<td>1458000000</td>
<td>36.53</td>
</tr>
<tr>
<td>((pp</td>
<td>gg))</td>
<td>40000</td>
<td>1458000000</td>
<td>34.23</td>
</tr>
<tr>
<td>((ff</td>
<td>ff))</td>
<td>10000</td>
<td>2100000000</td>
<td>40.43</td>
</tr>
<tr>
<td>((ff</td>
<td>dd))</td>
<td>20000</td>
<td>1296000000</td>
<td>37.54</td>
</tr>
<tr>
<td>((dd</td>
<td>ff))</td>
<td>20000</td>
<td>1296000000</td>
<td>37.69</td>
</tr>
<tr>
<td>((ff</td>
<td>pp))</td>
<td>80000</td>
<td>1080000000</td>
<td>31.46</td>
</tr>
<tr>
<td>((fp</td>
<td>ff))</td>
<td>80000</td>
<td>1080000000</td>
<td>32.23</td>
</tr>
<tr>
<td>((dd</td>
<td>dd))</td>
<td>60000</td>
<td>1166400000</td>
<td>31.10</td>
</tr>
</tbody>
</table>
Conclusions

- Rysq quadrature implementation performance results are comparable or better than DGEMV BLAS routines.

- Some more improvements are possible by caching (texture, constant) and also by more aggressive memory reuse possibly at the expense of recomputation.

- Very easy to generate the possible ERI shell combinations using a single template.

- Explicit unrolling can be controlled at different levels such as shells, roots to test for performance improvements.

- Being developed as a standalone library and application agnostic.
On going work

- ERIs are 4-dimensional, hence it is very expensive to transfer them to the host memory after computation.
- Fock matrix is 2-dimensional. So, consume the ERI’s as they are formed to build the Fock matrix
- Handle the contracted ERI’s
- Mixed precision support
- A complete working SCF algorithm
Acknowledgements & Contacts

US Department of Energy
Department of Defense - DURIP Grant
Ames Laboratory, Iowa State University
Air Force Office of Scientific Research
National Science Foundation - Petascale Applications grant
NVIDIA Corporation
Professor Todd Martinez and his group

asadchev@gmail.com
jfelder@iastate.edu
allada.v@gmail.com
mark@si.msg.chem.iastate.edu
theresa@fi.ameslab.gov
brett@si.msg.chem.iastate.edu