To write or rewrite

Theresa L. Windus
Department of Chemistry & Ames Laboratory
Iowa State University
(with a cast of characters – Mark Gordon, Andrey Asadchev, George Schoendorff and Sarom Sok Leang)
QM CHALLENGE

• High cost of electronic structure calculations
  – HF, hybrid DFT scale \( \sim N^4 \) (\( N \) = \# basis functions)
    • Up to 1,000 atoms
    • Analytic gradients, Hessians available
  – MP2 scales \( \sim N^5 \)
    • \( \sim 100 \) atoms
    • Analytic gradients, Hessians available
  – CCSD(T) & related methods scale \( \sim N^7 \)
    • \( 2^7 = 128 \)
    • 10-20 atoms
    • Analytic gradients available

• There are many possible solutions
To write or rewrite

- Write from scratch
  - Use code generation, templating and meta-programming with C++ to enable flexibility and adaptability
- Rewrite with existing code (GAMESS)
  - Try to take advantage of acceleration for ~2 million lines of code
**Choice for write**

- Make use of C++ templates & automatically generated code
  - Python-based Cheetah code generation engine
  - BOOST libraries
  - OpenMP with CUDA

- Human hands-on code small: ~ 2,000 lines of code for the integral code (rest is generated)

- Final library size small: ~ 5 MB fully optimized

- Code kept small due to objects & generic templates

- GPU implementation for integrals driven by angular momentum & contraction order
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
$F = H_{\text{core}} + G$

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

G – Matrix
$O(M^2)$
$G = [(ij|kl) - \frac{1}{2}(ik|jl)]*P$

Convergence Checks
- Yes
  - Stop
- No
  - Repeat steps 4, 5, 6, 7

Summary of Hartree-Fock Procedure

ERI
Two Electron Repulsion Integral
$(\mu\nu|\lambda\sigma)$
$O(M^3)$ to $O(M^4)$

Cheap one-time operation
- Required in every iteration
- Very expensive operation
- Stored procedures not scalable
- Re-compute in every iteration
- Good target for GPU

Update the density matrix from C
Repeat steps 4, 5, 6, 7
### INTEGRAL TIMINGS

blocks\(^1\) – number of ERI blocks evaluated  
flop count\(^2\) – total floating point operations  
GFLOPS\(_{SP}\)^{3} – single precision performance  
GFLOPS\(_{DP}\)^{4} – double precision performance  
GFLOPS\(^{5}\) – GAMESS Rys Quadrature, Intel Xeon, 8 GFLOPS peak performance

<table>
<thead>
<tr>
<th>ERI</th>
<th>GFLOPS(_{SP})^{3}</th>
<th>GFLOPS(_{DP})^{4}</th>
<th>GFLOPS(^{5}) GAMEESS</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>GTX 275</td>
<td>Tesla</td>
<td>GTX 275</td>
</tr>
<tr>
<td>(gg</td>
<td>gg)</td>
<td>45.23</td>
<td>55.97</td>
</tr>
<tr>
<td>(gg</td>
<td>ff)</td>
<td>34.42</td>
<td>42.07</td>
</tr>
<tr>
<td>(ff</td>
<td>gg)</td>
<td>30.91</td>
<td>37.70</td>
</tr>
<tr>
<td>(gg</td>
<td>dd)</td>
<td>43.08</td>
<td>53.39</td>
</tr>
<tr>
<td>(dd</td>
<td>gg)</td>
<td>23.63</td>
<td>24.03</td>
</tr>
<tr>
<td>(gg</td>
<td>pp)</td>
<td>36.53</td>
<td>45.15</td>
</tr>
<tr>
<td>(pp</td>
<td>gg)</td>
<td>34.23</td>
<td>42.42</td>
</tr>
<tr>
<td>(ff</td>
<td>ff)</td>
<td>40.43</td>
<td>50.19</td>
</tr>
<tr>
<td>(ff</td>
<td>dd)</td>
<td>37.54</td>
<td>46.15</td>
</tr>
<tr>
<td>(dd</td>
<td>ff)</td>
<td>37.69</td>
<td>45.71</td>
</tr>
<tr>
<td>(ff</td>
<td>pp)</td>
<td>31.46</td>
<td>41.38</td>
</tr>
</tbody>
</table>
HARTREE-FOCK

- Organize atomic basis into blocks
  - shells in block are the same, except they are on different atomic centers
- Corresponding matrixes and tensors inherit block structure
  - allows same integral kernel to compute multiple quartets
- Favorable for both CPU and GPU
- HF algorithm uses single collapsed synchronized loop
  - No nested loops: Makes parallelization easier
- CPU/GPU thread processes work queue independently
  - Locks Fock matrix when block is ready for update
## Xeon E5405 2.00 GHz CPUs

<table>
<thead>
<tr>
<th>Basis Set</th>
<th>GAMESS</th>
<th>GAMESS</th>
<th>C++</th>
<th>C++ 8 threads</th>
<th># bf</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>Rys only</td>
<td>1 core</td>
<td>CPU</td>
<td></td>
</tr>
<tr>
<td>Cocaine 6-311G++(2df,2p)</td>
<td>16895.3</td>
<td>17682.3</td>
<td>10986.2</td>
<td>1447.3</td>
<td>1049</td>
</tr>
<tr>
<td>6-31G++(d,p)</td>
<td>526.4</td>
<td>735.6</td>
<td>463.5</td>
<td>63.4</td>
<td>537</td>
</tr>
<tr>
<td>6-31G</td>
<td>95.9</td>
<td>233.1</td>
<td>157.9</td>
<td>23.0</td>
<td>235</td>
</tr>
</tbody>
</table>

![Cocaine molecule structure](image)
Xeon E5405 2.00 GHz CPUs
Fermi

<table>
<thead>
<tr>
<th>Basis Set</th>
<th>GAMESS</th>
<th>GAMESS</th>
<th>C++</th>
<th>C++/CUDA</th>
<th>C++, 8 threads</th>
<th># bf</th>
</tr>
</thead>
<tbody>
<tr>
<td>Cocaine</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>6-311G++(2df,2p)</td>
<td>16895.3</td>
<td>17682.3</td>
<td>10986.2</td>
<td>1911.6</td>
<td>1447.3</td>
<td>1049</td>
</tr>
<tr>
<td>6-31G++(d,p)</td>
<td>526.4</td>
<td>735.6</td>
<td>463.5</td>
<td>317.1</td>
<td>63.4</td>
<td>537</td>
</tr>
<tr>
<td>6-31G</td>
<td>95.9</td>
<td>233.1</td>
<td>157.9</td>
<td>37.8</td>
<td>23.0</td>
<td>235</td>
</tr>
</tbody>
</table>

Good software is as important as good hardware
HARTREE-FOCK TIMINGS

- CPU = Intel Xeon E5405 2.0 GHz w/ 4 cores
- GPU = Tesla C2050
- Test case = Taxol, first SCF iteration

Taxol: Anticancer natural product with activity against several leukemia varieties and cancers
## HARTREE-FOCK TIMINGS

### 6-31G(d): 1032 basis functions

<table>
<thead>
<tr>
<th>#CPU</th>
<th>Time (min)</th>
<th>#CPU/#GPU</th>
<th>Total time/GPU time (min)</th>
</tr>
</thead>
<tbody>
<tr>
<td>4</td>
<td>04:04</td>
<td>4/1</td>
<td>02:00/01:53</td>
</tr>
<tr>
<td>2</td>
<td>07:59</td>
<td>2/1</td>
<td>02:34/02:27</td>
</tr>
<tr>
<td>1</td>
<td>15:54</td>
<td>1/1</td>
<td>02:56/02:51</td>
</tr>
</tbody>
</table>

### CPU only (no GPU)                         With 1 GPU thread

### 6-31G(df,p): 1805 basis functions

<table>
<thead>
<tr>
<th>#CPU</th>
<th>Time (min)</th>
<th>#CPU/#GPU</th>
<th>Total time/GPU time (min)</th>
</tr>
</thead>
<tbody>
<tr>
<td>4</td>
<td>13:30</td>
<td>4/1</td>
<td>07:02/06:35</td>
</tr>
<tr>
<td>2</td>
<td>26:40</td>
<td>2/1</td>
<td>10:41/09:34</td>
</tr>
<tr>
<td>1</td>
<td>53:11</td>
<td>1/1</td>
<td>14:52/11:43</td>
</tr>
</tbody>
</table>

### CPU only (no GPU)                         With 1 GPU thread
MP2 ALGORITHM

• Key step is transformation from AO basis to MO basis
  – \([\mu, \nu, \lambda, \sigma] \rightarrow [i, j, a, b]\) in two-electron integrals (index transformation)
  – Called “four label” transformation (integral transformation)
  – Required for all correlated methods (MP2, CC, also Hessians)

• \(i, j = \) occupied MO indices; \(a, b = \) virtual MO indices
  – For large basis sets (usual) many more virtual indices than occupied indices

• Most computations are in the 1st 3 index transformations
  – Can do transformations in any order
  – Choose to first transform to occupied indices: \((\mu \nu | \lambda \sigma) \rightarrow (\mu i | \lambda j)\)
    • Takes advantage of smaller occupied space and \(ij\) symmetry
  – Then transform to first virtual index \((a),\) stored in memory as \(T3 (a, ij, \mu)\)
  – Final transformation is easy
  – Implemented on both CPU and GPU
Taxol: 1032 orbitals
62 core, 226 occupied
2 quad core Xeon 2.66 GHz
~17x speedup on Fermi
(relative to GAMESS)
Rewrite choices

• cuBLAS (MAGMA)

• PGI Accelerator Model

• OpenAcc

• Explicit GPU programming model
cuBLAS 4.0

• Can be used with any FORTRAN or C compiler
• Need to compile and link to fortran.c and cublas.so
• Allows asynchronous execution
• cuBLAS 4.0 includes all standard BLAS routines
cuBLAS

- Can be used with any Fortran or C compiler
- Two ways to compile
  1. Thunking
     - Replace blas routine with cublas routine
       e.g. dgemm() becomes cublasDgemm()
     - Uses cublasCreate() and cublasDestroy() for each instance
     - Intended for testing purposes only
  2. Non-thunking
     - Programmer handles cublasCreate(), cublasDestroy(), and all memory copies
cuBLAS with PGI

DGEMM Example:

module cublas
  interface cuda_gemm
    subroutine cuda_dgemm(cta, ctb, m, n, k,&
                          alpha, A, lda, B, ldb, beta, c, ldc)
      bind(C,name='cublasDgemm')
    use iso_c_binding
    character(1,c_char),value :: cta, ctb
    integer(c_int),value :: m,n,k,lda,ldb,ldc
    real(c_double),value :: alpha,beta
    real(c_double), device, dimension(lda,*) :: A
    real(c_double), device, dimension(ldb,*) :: B
    real(c_double), device, dimension(ldc,*) :: C
    end subroutine cuda_dgemm
  end interface
end module cublas

call cuda_gemm (‘N’, ‘N’, m, n, k, alpha, a_d, m, b_d, k, beta, c_d, m)

- No cublas_init needed with PGI compiler
- Implicit synchronous memory transfers with PGI compiler or explicit asynchronous memory transfers with cublas helper functions
PGI Accelerator Model

• Simple method to parallelize loops, especially nested loops
• Data regions can be defined to manage memory transfers
• Method is available for PGI Fortran (77 and 90) as well as PGI C/C++
PGI Accelerator Model

- GPU code is generated by the compiler

- Loop(s) to be parallelized enclosed by the following directives:

  !$acc region clause, clause, ...
  DO ...
  !$acc end region

- The clauses are optional arguments to specify how to implement the GPU parallelization

  - Examples:
    - !$acc region if(condition) Compile for GPU if condition is true
    - !$acc region deviceptr(list) Specify which GPUs to use
    - !$acc region async Run CPU and GPU code asynchronously

PGI Accelerator Model

• Memory transfers can be handled by the compiler

• Memory transfers can be handled explicitly using the following directives:

  !$acc data region clause, clause, etc.

  ....

  !$acc end data region

• The clauses are optional arguments that determine how the data is handled

  - Examples:

    !$acc data region copyout(list)  Copies a list of variables from the device memory back to the host

    !$acc data region mirror(list)  Mirrors host and device data

    !$acc region update host (list)  Copies a list of variables to the host after every kernel call

PGI Accelerator Model - Restrictions

• A variable can occur only once in a compute/data region
  - Additional directives may be needed if a variable is reused.

• The upper bound for the last dimension of an assumed-size array must be specified

• Assumed-size arrays are not valid in data clauses

• Accelerator regions cannot be nested

• cuBLAS routines cannot be called within an accelerator region

## GAMESS DFT Profile

(\% of total wall time)

<table>
<thead>
<tr>
<th>Routine</th>
<th>B3LYP/6-31G(d,p)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>taxol</td>
</tr>
<tr>
<td></td>
<td>(110 atoms/1160</td>
</tr>
<tr>
<td></td>
<td>bfns)</td>
</tr>
<tr>
<td>Two electron integrals</td>
<td>38.74</td>
</tr>
<tr>
<td>DFT</td>
<td>58.33</td>
</tr>
<tr>
<td>DFT: Becke grid weights</td>
<td>7.51</td>
</tr>
<tr>
<td>DFT: Fock build</td>
<td>24.17</td>
</tr>
<tr>
<td>DFT: gradient transformation</td>
<td>16.21</td>
</tr>
</tbody>
</table>
## GAMESS DFT Profile (% of total wall time)

<table>
<thead>
<tr>
<th>Routine</th>
<th>taxol B3LYP/Basis Set (110 atoms)</th>
<th>6-31G(d,p) (1160 bfns)</th>
<th>6-31+G(d,p) (1404 bfns)</th>
<th>6-311G(d,p) (1453 bfns)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Two electron integrals</td>
<td></td>
<td>38.74</td>
<td>65.71</td>
<td>51.25</td>
</tr>
<tr>
<td>DFT</td>
<td></td>
<td>58.33</td>
<td>28.82</td>
<td>45.49</td>
</tr>
<tr>
<td>DFT: Becke grid weights</td>
<td></td>
<td>7.51</td>
<td>1.02</td>
<td>4.42</td>
</tr>
<tr>
<td>DFT: Fock build</td>
<td></td>
<td>24.17</td>
<td>9.86</td>
<td>19.83</td>
</tr>
<tr>
<td>DFT: gradient transformation</td>
<td></td>
<td>16.21</td>
<td>6.03</td>
<td>13.12</td>
</tr>
</tbody>
</table>
Becke Grid Weights

**Pseudocode:**

Radial grid loop
  Angular grid loop
    Atom i loop
      Atom j loop
    Calculate grid weights

- Typical radial grid size (Euler-MacLaurin)
  - 48, 96 can be any value
- Typical angular grid size (Lebedev)
  - 302 (default), 1202 (army grade)
  - possible values \{86, 110, 146, 170, 194, 302, 350, 434, 590, 770, 974, 1202, 1454, 1730, 2030, …\}
- Simple nested loop structure
Becke Grid Weights
Accelerator Directives

**Pseudocode:**

Radial grid loop

```
!$acc region

Angular grid loop
  Atom i loop
  Atom j loop
  Calculate grid weights

!$acc end region
```

- First step
  - Add accelerator directives around desired loop
  - Compile with ‘–ta=nvidia  -Minfo’
  - Read compiler output
Becke Grid Weights
Compiler Information

Compiler output:
40, Generating copyin(rij(1:nat,:))
   Generating copy(ri(:))
   Generating copyout(totwt(ncntr,:))
41, Loop carried dependence due to exposed use of 'ri(:)' prevents parallelization
50, Loop is parallelizable
58, Loop carried dependence of 'wtintr' prevents parallelization

- Compiler output offers information on:
  - Data movement (we wish to minimize)
  - Data dependencies (we wish to remove)
Becke Grid Weights
Minimize Data Movement

**Pseudocode:**

```fortran
!$acc data region copyin(...) 
    Radial grid loop 
!$acc data region copyout(...) 
!$acc region 
    Angular grid loop 
        Atom i loop 
            Atom j loop 
                Calculate grid weights 
!$acc end region 
!$acc end data region 
!$acc end data region 
```

- Data region
  - Copy in frequently access arrays only **once**
  - Copy out contiguous arrays only **once**
Pseudocode:

\[
\begin{align*}
!$acc\ data\ region\ copyin(\ldots) \\
 & \quad \text{Radial grid loop} \\
!$acc\ data\ region\ copyout(\ldots) \\
!$acc\ region \\
 & \quad \text{Angular grid loop} \\
 & \quad \quad \text{Atom i loop} \\
 & \quad \quad \quad \text{Atom j loop} \\
 & \quad \quad \quad \text{Calculate grid weights} \\
!$acc\ end\ region \\
!$acc\ end\ data\ region \\
!$acc\ end\ data\ region \quad \text{Additional loop structure to utilize arrays that were copied out.}
\end{align*}
\]
Becke Grid Weights
Final Result

Pseudocode:

 !$acc data region copyin(...)
  Radial grid loop
 !$acc data region copyout(...)  
 !$acc region

 Angular grid loop
    Atom i loop
       Atom j loop
       Calculate grid weights
 !$acc end region
 !$acc end data region
 !$acc end data region

 Additional loop structure to utilize arrays that were copied out.
## Speed-Up

### Becke Grid Weights

<table>
<thead>
<tr>
<th>Time spent on Becke grid weights (48, 302)</th>
<th>B3LYP/3-21G</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>taxol (110 atoms)</td>
</tr>
<tr>
<td>CPU Code (sec)</td>
<td>386</td>
</tr>
<tr>
<td>GPU Code (sec)</td>
<td>330</td>
</tr>
<tr>
<td>Speed-Up (x)</td>
<td>1.17</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Time spent on Becke grid weights (96, 1202)</th>
<th>B3LYP/3-21G</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>taxol (110 atoms)</td>
</tr>
<tr>
<td>CPU Code (sec)</td>
<td>3064</td>
</tr>
<tr>
<td>GPU Code (sec)</td>
<td>705</td>
</tr>
<tr>
<td>Speed-Up (x)</td>
<td>4.35</td>
</tr>
</tbody>
</table>

Intel 2.67 GHz hex core X5650 (Gulftown), Nvidia Fermi C2070
## Speed-Up
### Total Wall Time

<table>
<thead>
<tr>
<th>Total wall time</th>
<th>B3LYP/3-21G</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>taxol (110 atoms)</td>
</tr>
<tr>
<td>CPU Code (sec)</td>
<td>25593.2</td>
</tr>
<tr>
<td>GPU Code (sec)</td>
<td>23636.8</td>
</tr>
<tr>
<td>Speed-Up (x)</td>
<td>1.08</td>
</tr>
</tbody>
</table>
Beyond 472 atoms?

Pseudocode:
...

!$acc region

Angular grid loop
  Atom i loop
    Atom j loop
      Calculate grid weights

threadblock

thread

!$acc end region
...

- The kernel contains a double precision array of size [Natoms x Natoms x Nangular].
- PGI uses 32-bit integers for array addressing.
- An index overflow occurs for systems with > 472 atoms.
- Work around: break up kernel until PGI offers a solution.
Becke Grid Weights
Work Chunks in Highest Power of 2

Pseudocode:
chunk loop: DO IANG_=1, Nangular-Nchunk, Nchunk
    Radial grid loop
    !$acc region
        Angular grid loop: DO IANG=IANG_, IANG+Nchunk-1
            Atom i loop
                Atom j loop
                    Calculate grid weights
        !$acc end region
    ENDDO chunk loop
IANG_=Nangular-MOD(Nangular,Nchunk)+1
    Radial grid loop
    !$acc region
        Angular grid loop: DO IANG=IANG_, Nangular
            Atom i loop
                Atom j loop
                    Calculate grid weights
    !$acc end region
### Speed-Up

Becke Grid Weights, Highest Power of 2

<table>
<thead>
<tr>
<th>Time spent on Becke grid weights (48, 302)</th>
<th>B3LYP/3-21G</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>taxol (110 atoms)</td>
</tr>
<tr>
<td>CPU Code (sec)</td>
<td>386</td>
</tr>
<tr>
<td>GPU Code (sec)</td>
<td>48</td>
</tr>
<tr>
<td>Speed-Up (x)</td>
<td>8.04</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Time spent on Becke grid weights (96, 1202)</th>
<th>B3LYP/3-21G</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>taxol (110 atoms)</td>
</tr>
<tr>
<td>CPU Code (sec)</td>
<td>3064</td>
</tr>
<tr>
<td>GPU Code (sec)</td>
<td>339</td>
</tr>
<tr>
<td>Speed-Up (x)</td>
<td>9.03</td>
</tr>
</tbody>
</table>
# Speed-Up

## Total Wall Time, Highest Power of 2

<table>
<thead>
<tr>
<th>Total wall time</th>
<th>B3LYP/3-21G</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>taxol (110 atoms)</td>
</tr>
<tr>
<td>CPU Code (sec)</td>
<td>25593.2</td>
</tr>
<tr>
<td>GPU Code (sec)</td>
<td>19212.1</td>
</tr>
<tr>
<td>Speed-Up (x)</td>
<td>1.33</td>
</tr>
</tbody>
</table>
Beyond 472 atoms?

• Calculations pending
  – Carbon 540 (540 atoms)
  – BPTI (900 atoms)
Multi-GPU: OpenMP & OpenACC

**Pseudocode:**

```plaintext
Ngpu=ACC_GET_NUM_DEVICES(ACC_DEVICE_NVIDIA)
CALL OMP_SET_NUM_THREADS(Ngpu)
NOMPCHUNK=Nradial/NGPU
!omp parallel private(…)
OMP_ME=OMP_GET_THREAD_NUM()
IF (Ngpu > 1)
   IOMP_START=NOMPCHUNK*OMP_ME+1
   IOMP_END=NOMPCHUNK*OMP_ME+NOMPCHUNK
ELSE
   IOMP_START=1
   IOMP_END=Nradial
ENDIF
CALL ACC_SET_DEVICE_NUM(OMP_ME,ACC_DEVICE_NVIDIA)
...
```
Multi-GPU: OpenMP & OpenACC

**Pseudocode:**
chunk loop: DO IANG_=1, Nangular-Nchunk, Nchunk
    Radial grid loop: DO IRAD=IOMP_START, IOMP_END
    !$acc region
    Angular grid loop: DO IANG=IANG_, IANG_+Nchunk-1
        Atom i loop
            Atom j loop
                Calculate grid weights
    !$acc end region
ENDDO chunk loop
IANG_=Nangular-MOD(Nangular,Nchunk)+1
    Radial grid loop: DO IRAD=IOMP_START, IOMP_END
    !$acc region
    Angular grid loop: DO IANG=IANG_, Nangular
        Atom i loop
            Atom j loop
                Calculate grid weights
    !$acc end region
!$omp end parallel
# Speed-Up

## Total Wall Time, Highest Power of 2, Multi-GPU

<table>
<thead>
<tr>
<th>Total wall time (NGPU = 2)</th>
<th>B3LYP/3-21G</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>taxol (110 atoms)</td>
</tr>
<tr>
<td>CPU Code (sec)</td>
<td>25593.2</td>
</tr>
<tr>
<td>GPU Code (sec)</td>
<td>19106.0</td>
</tr>
<tr>
<td>Speed-Up (x)</td>
<td>1.34</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Total wall time (NGPU = 4)</th>
<th>B3LYP/3-21G</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>taxol (110 atoms)</td>
</tr>
<tr>
<td>CPU Code (sec)</td>
<td>25593.2</td>
</tr>
<tr>
<td>GPU Code (sec)</td>
<td>19727.6</td>
</tr>
<tr>
<td>Speed-Up (x)</td>
<td>1.30</td>
</tr>
</tbody>
</table>
Explicit Model

- Programmer explicitly manages data, kernel definition, and kernel calls
- Fortran code should be written for Fortran 90
  - Allows use of modules and dynamic allocation for device data
- Device data managed in a similar manner to host data
  - Example:
    
    Given:
    
    Host          Device
    real, dimension(*) :: vec   real, device dimension(*) :: vec_d
    integer A    integer, device A_d

- Simple copying of data to/from device memory
  - Example:
    
    A_d=A    Copy A from host to device
    A=A_d    Copy A_d from device to host

Restrictions

• Cannot call cuBLAS routines from within a GPU kernel
• If statements within the kernel lead to thread divergence
• Memory copies cannot execute asynchronously
• Fortran 90 is required (allocatable arrays)
CONCLUSIONS & FUTURE

• GPUs are great accelerators
• Significant development time - years!
  • Not a bed of roses - big payoff
• GPU/CUDA forces better coding practices
• In progress
  • Hehre-Pople, ERIC integral packages
  • Full DFT
  • Open shells
  • Gradients
  • Coupled cluster
  • FMO, FMO gradients
  • EFP
THANKS!

US Air Force Office of Scientific Research
DoD DURIP (GPU Cluster)
ISU (GPU Cluster)
NSF
ENABLING ACCURATE CALCULATIONS FOR LARGE MOLECULAR SYSTEMS

• Maintain accuracy/reduce cost
  • Highly scalable code
  • Novel algorithms
  • Take advantage of GPUs

• Avoid the gong
CATALYSIS

777 atoms 3625 basis functions

Mesoporous Silica Nanoparticles

<table>
<thead>
<tr>
<th></th>
<th>RHF/STO-3G</th>
<th>FM02/STO-3G</th>
</tr>
</thead>
<tbody>
<tr>
<td>Absolute energy (h)</td>
<td>-73537.7728</td>
<td>-73537.7627</td>
</tr>
<tr>
<td>Dipole Moment (debye)</td>
<td>108.6</td>
<td>105.8</td>
</tr>
</tbody>
</table>
Helper Functions

• `cublasCreate()` initializes the GPU

• `cublasDestroy()` releases GPU resources

• `cublasSetStream()`
  - up to 16 concurrent streams per GPU can be executed if there are sufficient resources

• `cublasGetStream()` returns the streamid
Memory Transfers

- `cublasSetVector` and `cublasGetVector`
  - Copies a vector to/from GPU memory
  - `cublasSetVector(N, elemSize, X, incX, Y, incY)`
    - $N =$ number of elements
    - $\text{elemSize} =$ size of each elements in bytes
    - $X =$ vector in host memory
    - $\text{incX} =$ size of an element of $X$ in bytes
    - $Y =$ vector in device memory
    - $\text{incY} =$ size of an element of $Y$ in bytes

- `cublasSetMatrix` and `cublasGetMatrix` are analogous to the vector functions
Memory Transfers

- PGI Fortran does not need helper functions
  
  Example – copy vector A on the host to A_d on the device:
  
  \[
  A_d = A
  \]

- Only works when compiled with the cudafor module

- Memory copies are synchronous

- Helper functions should not be mixed with implicit data transfers.
Asynchronous Memory Transfers

• Memory can be transferred asynchronously using cublasSetVectorAsync(), cublasSetMatrixAsync(), cublasGetVectorAsync(), and cublasGetMatrixAsync()

• Transfer is asynchronous with respect to the host

• Usage is same as for synchronous transfers
Cublas Summary

- Provides a way to use GPUs with only minor changes to the CPU code
- Cublas routines can be executed asynchronously and on multiple GPUs
- Cublas 4.0 includes all blas routines
Additional Information

- www.pgroup.com/lit/articles/insiter/v1n1a1.htm
- www.pgroup.com/resources/accel.htm
- www.openacc-standard.org/home
- www.pgroup.com/resources/cudafortran.htm
OpenAcc

• Allows for a single code base for CPU and accelerated architecture

• In principle, not limited to Nvidia GPUs

• Based on the PGI Accelerator model with minor changes
## OpenAcc vs. PGI

<table>
<thead>
<tr>
<th>OpenAcc</th>
<th>PGI</th>
</tr>
</thead>
<tbody>
<tr>
<td>!$acc parallel</td>
<td>!$acc region</td>
</tr>
<tr>
<td>!$acc loop</td>
<td>!$acc do</td>
</tr>
</tbody>
</table>

- Directives can be accompanied by clauses that specify how the data and execution is handled.
- Many clauses are the same, but minor differences exists.
ELECTRON REPULSION INTEGRALS

- Major computational step in both HF and DFT methods
- Complexity is $O(N^3)$-$O(N^4)$, $N = \text{number of Gaussian basis functions}$
- Method choice depends on basis function angular momenta:
  - $s,p$: Hehre-Pople
  - $s,p,d$: McMurchie-Davidson
  - $s,p,d,f,g$: Rys polynomials, ERIC, Obara-Saika
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
Variable Qualifiers

• Device variables can be declared with qualifiers *constant, shared, or pinned*
  
  - Example:
    real, device attributes(constant) var_d

• Constant
  - Data is stored in device constant memory
  - Cannot be modified from device code, but can be changed in host code
  - Fast memory access

• Shared
  - Data is stored in device shared memory
  - Visible to all threads in a block

• Pinned
  - Page-locked memory, but reverts to unpinned memory if page-lock fails
  - Fast data transfer when it works

Kernel Implementation

- Syntax for GPU kernels is similar to that used with C/C++
- Subroutines can become kernels
- Subroutines are prefixed with “attributes("type")” to specify the execution
  - attributes(host) is a subroutine executed on the host (default)
  - attributes(global) is a subroutine/kernel executed on the device that is called from the host
  - attributes(device) is a subroutine/kernel executed on the device that is executed on the device and called from a “global” or “device” subroutine

Kernel Calls

- Chevron notation, \texttt{<<< … >>>}, used to specify how the kernel is executed on the device
  - Example:
    \begin{verbatim}
    call kernel\texttt{<<<grid,block,bytes,streamid>>>}(arg1,arg2,...)
    \end{verbatim}

- \textit{grid} is a type \texttt{dim3} variable that specifies the grid dimensions

- \textit{block} is a type \texttt{dim3} variable that specifies the block dimensions

- \textit{bytes} is an optional integer variable that designates the amount of memory available to the kernel

- \textit{streamid} is an optional integer variable to determine in which stream the kernel is executed

Cellulose: BIOMASS CONVERSION

Chain: linear polymer of ß-(1, 4)-D-glucopyranose units

Crystal: I-ß cellulose (in plants), monoclinic, H-bonded sheets
Accelerator Directives

• Benefits
  – Similar to OpenMP
    • Single source code
    • Target multiple architectures
    • Open standard
    • Incremental programming

• Compiler: Portland Group (PGI)
  – Supports mix directives and CUDA Fortran
Future Work

• DFT: Fock Build (20-24%)
• DFT: Gradient Transformations (6-16%)
• DFT Gradients
• TDDFT Energy
• Low level performance tuning with CUDA Fortran
• Multi-node, single-gpu
• Multi-node, multi-gpu
CCSD(T): Step 1

- Sequential C++ CPU code
- (T) only
- 6-31G(d) Caffeine, 230 AOs, Xeon 2 GHz CPU
- Wall Times
  - Current GAMESS: ~11 hours
  - C++ code: ~7.3 hours (includes 0.3 hour for I/O)
- Memory
  - Current GAMESS: 4.6 GB distributed, 1 GB local
  - C++ code: ~258 MB