Programming models for quantum many-body methods on multicore and manycore processors

Jeff Hammond\textsuperscript{1} and Eugene DePrince\textsuperscript{2}

\textsuperscript{1} Argonne
\textsuperscript{2} Georgia Tech

6 February 2011
The growing diversity in computer processor architectures poses a serious challenge to the computational chemistry community. This talk considers some of the key issues, including disjoint address spaces, non-standard architectures and execution models, and the different APIs required to use them. Specifically, we will describe our experiences in developing coupled-cluster methods for Intel multicore, Intel MIC, NVIDIA Fermi and Blue Gene/Q in both a clean-sheet implementation and NWChem. Of fundamental interest is the development of codes that scale not only within the node but across thousands of nodes; hence, the interaction between the processor and the network will be analyzed in detail.
Software Automation
What does it do?

1. GUI input quantum many-body theory e.g. CCSD.
2. Operator specification of theory.
3. Apply Wick’s theory to transform operator expressions into array expressions.
4. Transform input array expression to operation tree using many types of optimization.
5. Produce Fortran+Global Arrays+NXTVAL implementation.

Developer can intercept at various stages to modify theory, algorithm or implementation.
The practical TCE – Success stories

- First parallel implementation of many (most) CC methods.
- First truly generic CC code (not string-based):
  \[ \{ \text{RHF, ROHF, UHF} \} \times \text{CC} \{ \text{SD, SDT, SDTQ} \} \times \{ T/\Lambda, \text{EOM, LR/QR} \} \]
- Most of the largest calculations of their kind employ TCE:
  CR-EOMCCSD(T), CCSD-LR \( \alpha \), CCSD-QR \( \beta \), CCSDT-LR \( \alpha \)
- Reduces implementation time for new methods from years to hours, TCE codes are easy to verify.

Significant hand-tuning by Karol Kowalski and others at PNNL was required to make TCE run efficiently and scale to 1000 processors and beyond.
Before TCE

CCSD/aug-cc-pVDZ – 192 b.f. – days on 1 processor

Benzene is close to crossover point between small and large.
Linear response polarizability

CCSD/Z3POL – 1080 b.f. – 40 hours on 1024 processors

This problem is 20,000 times larger on the computer than benzene.

Quadratic response hyperpolarizability

CCSD/d-aug-cc-pVTZ – 812 b.f. – 20 hours on 1024 processors

Lower levels of theory are not reliable for this system.

Charge-transfer excited-states of biomolecules

CR-EOMCCSD(T)/6-31G* – 584 b.f. – 1 hour on 256 cores

Lower levels of theory are wildly incorrect for this system.
Excited-state calculation of conjugated arrays

CR-EOMCCSD(T)/6-31+G* – 1096 b.f. – 15 hours on 1024 cores

### Summary of TCE module

http://cloc.sourceforge.net v 1.53  T=30.0 s

<table>
<thead>
<tr>
<th>Language</th>
<th>files</th>
<th>blank</th>
<th>comment</th>
<th>code</th>
</tr>
</thead>
<tbody>
<tr>
<td>Fortran 77</td>
<td>11451</td>
<td>1004</td>
<td>115129</td>
<td>2824724</td>
</tr>
</tbody>
</table>

**SUM:** 11451 1004 115129 2824724

Only $< 25$ KLOC are hand-written; $\sim 100$ KLOC is utility code following TCE data-parallel template.
http://cloc.sourceforge.net v 1.53  T=13.0 s

<table>
<thead>
<tr>
<th>Language</th>
<th>files</th>
<th>blank</th>
<th>comment</th>
<th>code</th>
</tr>
</thead>
<tbody>
<tr>
<td>Fortran 77</td>
<td>5757</td>
<td>0</td>
<td>29098</td>
<td>983284</td>
</tr>
</tbody>
</table>

SUM: 5757 0 29098 983284

Total does not include ~1M LOC that was reused (EOM).

CCSD quadratic response hyperpolarizability was derived, implemented and verified during a two week trip to PNNL. Over 100 KLOC were “written” in under an hour.
Before Automation

(The Benz comes before the Ford)
What code are we going to generate?

- Legacy design limits options
- NVIDIA hardware was changing fast
- CUDA Fortran didn’t exist in 2009
- Eugene had a loop-based CEPA code in C...
- Assume that data motion across PCI bus was limiting
- Code changed a lot as CUBLAS supported streams, etc.
Coupled-cluster theory
Coupled-cluster theory

The coupled–cluster (CC) wavefunction ansatz is

$$|CC\rangle = e^T |HF\rangle$$

where $T = T_1 + T_2 + \cdots + T_n$.

$T$ is an excitation operator which promotes $n$ electrons from occupied orbitals to virtual orbitals in the Hartree-Fock Slater determinant.

Inserting $|CC\rangle$ into the Schrödinger equation:

$$\hat{H} e^T |HF\rangle = E_{CC} e^T |HF\rangle \quad \hat{H} |CC\rangle = E_{CC} |CC\rangle$$
Coupled-cluster theory

\[ |CC\rangle = \exp(T)|0\rangle \]
\[ T = T_1 + T_2 + \cdots + T_n \quad (n \ll N) \]
\[ T_1 = \sum_{ia} t_i^a \hat{a}_a \hat{a}_i \]
\[ T_2 = \sum_{ijab} t_{ij}^{ab} \hat{a}_a \hat{a}_b \hat{a}_j \hat{a}_i \]
\[ |\Psi_{CCD}\rangle = \exp(T_2)|\Psi_{HF}\rangle \]
\[ = (1 + T_2 + T_2^2)|\Psi_{HF}\rangle \]
\[ |\Psi_{CCSD}\rangle = \exp(T_1 + T_2)|\Psi_{HF}\rangle \]
\[ = (1 + T_1 + \cdots + T_4^1 + T_2 + T_2^2 + T_1 T_2 + T_1^2 T_2)|\Psi_{HF}\rangle \]
Coupled-cluster theory

Projective solution of CC:

\[
E_{CC} = \langle HF | e^{-T} He^T | HF \rangle \\
0 = \langle X | e^{-T} He^T | HF \rangle \quad (X = S, D, \ldots)
\]

CCD is:

\[
E_{CC} = \langle HF | e^{-T_2} He^{T_2} | HF \rangle \\
0 = \langle D | e^{-T_2} He^{T_2} | HF \rangle
\]

CCSD is:

\[
E_{CC} = \langle HF | e^{-T_1-T_2} He^{T_1+T_2} | HF \rangle \\
0 = \langle S | e^{-T_1-T_2} He^{T_1+T_2} | HF \rangle \\
0 = \langle D | e^{-T_1-T_2} He^{T_1+T_2} | HF \rangle
\]
\[ H = H_1 + H_2 = F + V \]

\( F \) is the Fock matrix. CC only uses the diagonal in the canonical formulation.

\( V \) is the fluctuation operator and is composed of two-electron integrals as a 4D array.

\( V \) has 8-fold permutation symmetry in \( V_{pq}^{rs} \) and is divided into six blocks: \( V_{ij}^{kl}, V_{ij}^{ka}, V_{ia}^{jb}, V_{ij}^{ab}, V_{ia}^{bc}, V_{ab}^{cd} \).

Indices \( i, j, k, \ldots (a, b, c, \ldots) \) run over the occupied (virtual) orbitals.
CCD Equations

\[ R_{ij}^{ab} = V_{ij}^{ab} + P(ia, jb) \left[ T_{ij}^{ae} I_e^b - T_{im}^{ab} I_m^j + \frac{1}{2} V_{ef}^{ab} T_{ij}^{ef} + \frac{1}{2} T_{mn}^{ab} I_{ij}^{mn} - T_{mj}^{ae} I_e^m - I_{ie}^m T_{mj}^{eb} + (2 T_{mi}^{ea} - T_{im}^{ea}) I_{ej}^{mb} \right] \]

\[ I_a^i = (2V_{mn}^{mi} - V_{mn}^{im}) T_{mn}^{ea} \]
\[ I_j^i = (V_{ef}^{mi} - V_{ef}^{im}) T_{mj}^{ef} \]
\[ I_{ij}^{ij} = V_{kl}^{ij} + V_{ef}^{ij} T_{kl}^{ef} \]
\[ I_{ij}^{ia} = V_{jb}^{ia} - \frac{1}{2} V_{eb}^{im} T_{jm}^{ea} \]
\[ I_{ij}^{ia} = V_{bj}^{ia} + V_{be}^{im} (T_{mj}^{ea} - \frac{1}{2} T_{mj}^{ae}) - \frac{1}{2} V_{be}^{mi} T_{mj}^{ae} \]
Some tensor contractions are trivially mapped to GEMM:

\[
\begin{align*}
I_{ij}^{(kl)} & = V_{ef}^{ij} T_{kl}^{ef} \\
I_{(ij)}^{(kl)} & = V_{(ef)}^{(ij)} T_{(kl)}^{(ef)} \\
I_{a}^{b} & = V_{c}^{b} T_{a}^{c}
\end{align*}
\]

Other contractions require reordering to use BLAS:

\[
\begin{align*}
I_{bji} & = V_{be}^{im} T_{mj}^{ea} \\
I_{bji,ia} & = V_{be,im} T_{mj,ea} \\
J_{ibi,ja} & = W_{bi,me} U_{me,ja} \\
J_{ja}^{bi} & = W_{bi}^{me} U_{me}^{ja} \\
J_{j(a)}^{(bi)} & = W_{j(a)}^{(me)} U_{(me)}^{(bi)} \\
J_{x}^{z} & = W_{x}^{y} U_{y}^{z}
\end{align*}
\]
Reordering can take as much time as GEMM. Why?

<table>
<thead>
<tr>
<th>Routine</th>
<th>flops</th>
<th>mops</th>
<th>pipelined</th>
</tr>
</thead>
<tbody>
<tr>
<td>GEMM</td>
<td>$O(mnk)$</td>
<td>$O(mn + mk + kn)$</td>
<td>yes</td>
</tr>
<tr>
<td>reorder</td>
<td>0</td>
<td>$O(mn + mk + kn)$</td>
<td>no</td>
</tr>
</tbody>
</table>

Increased memory bandwidth on GPU makes reordering less expensive (compare matrix transpose).

(There is a chapter in my thesis with profiling results and more details if anyone cares.)
## Hardware Details

<table>
<thead>
<tr>
<th></th>
<th>CPU</th>
<th>GPU</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>X5550</td>
<td>2 X5550</td>
</tr>
<tr>
<td>processor speed (MHz)</td>
<td>2660</td>
<td>2660</td>
</tr>
<tr>
<td>memory bandwidth (GB/s)</td>
<td>32</td>
<td>64</td>
</tr>
<tr>
<td>memory speed (MHz)</td>
<td>1066</td>
<td>1066</td>
</tr>
<tr>
<td>ECC available</td>
<td>yes</td>
<td>yes</td>
</tr>
<tr>
<td>SP peak (GF)</td>
<td>85.1</td>
<td>170.2</td>
</tr>
<tr>
<td>DP peak (GF)</td>
<td>42.6</td>
<td>83.2</td>
</tr>
<tr>
<td>power usage (W)</td>
<td>95</td>
<td>190</td>
</tr>
</tbody>
</table>

Note that power consumption is apples-to-oranges since CPU does not include DRAM, whereas GPU does.
Relative Performance of GEMM

GPU versus SMP CPU (8 threads):

Maximum:
CPU = 156.2 GF
GPU = 717.6 GF

Maximum:
CPU = 79.2 GF
GPU = 335.6 GF

We expect roughly 4-5 times speedup based upon this evaluation.
## Chemistry Details

<table>
<thead>
<tr>
<th>Molecule</th>
<th>o</th>
<th>v</th>
</tr>
</thead>
<tbody>
<tr>
<td>C$<em>8$H$</em>{10}$</td>
<td>21</td>
<td>63</td>
</tr>
<tr>
<td>C$_{10}$H$_8$</td>
<td>24</td>
<td>72</td>
</tr>
<tr>
<td>C$<em>{10}$H$</em>{12}$</td>
<td>26</td>
<td>78</td>
</tr>
<tr>
<td>C$<em>{12}$H$</em>{14}$</td>
<td>31</td>
<td>93</td>
</tr>
<tr>
<td>C$<em>{14}$H$</em>{10}$</td>
<td>33</td>
<td>99</td>
</tr>
<tr>
<td>C$<em>{14}$H$</em>{16}$</td>
<td>36</td>
<td>108</td>
</tr>
<tr>
<td>C$_{20}$</td>
<td>40</td>
<td>120</td>
</tr>
<tr>
<td>C$<em>{16}$H$</em>{18}$</td>
<td>41</td>
<td>123</td>
</tr>
<tr>
<td>C$<em>{18}$H$</em>{12}$</td>
<td>42</td>
<td>126</td>
</tr>
<tr>
<td>C$<em>{18}$H$</em>{20}$</td>
<td>46</td>
<td>138</td>
</tr>
</tbody>
</table>

- 6-31G basis set
- C$_1$ symmetry
- $F$ and $V$ from PSI3.
- GPU code now runs from PSI3.

Talk to Eugene about PSI4 (GPL).
CCD Algorithm

Copy V and F to GPU (cudaMemcpy)

**WHILE** (|dT|>eps) **DO**

**IF** $\mathbf{v}_{cd}^{ab}$ fits in GPU memory **THEN**
Update residual (cudaDgemm)

**ELSE**

**FOR** all tiles **DO**

- Copy $\mathbf{v}_{cd}^{ab}$ to GPU (cudaMemcpy)
- Contract $\mathbf{v}_{ef}^{ab} t_{ij}^{ef}$ (cudaDgemm)

**END DO**

Update residual with remaining terms (cudaDgemm)

**END IF**

Update T with residual
Compute |dT| (cudaDnrm2)

**END WHILE**

Copy amplitudes from GPU (cudaMemcpy)

$E = t.v$
## CCD Performance Results

### Iteration time in seconds

<table>
<thead>
<tr>
<th></th>
<th>our DP code</th>
<th></th>
<th>X5550</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>C2050</td>
<td>C1060</td>
<td>X5550</td>
</tr>
<tr>
<td>C₈H₁₀</td>
<td>0.3</td>
<td>0.8</td>
<td>1.3</td>
</tr>
<tr>
<td>C₁₀H₈</td>
<td>0.5</td>
<td>1.5</td>
<td>2.5</td>
</tr>
<tr>
<td>C₁₀H₁₂</td>
<td>0.8</td>
<td>2.5</td>
<td>3.5</td>
</tr>
<tr>
<td>C₁₂H₁₄</td>
<td>2.0</td>
<td>7.1</td>
<td>10.0</td>
</tr>
<tr>
<td>C₁₄H₁₀</td>
<td>2.7</td>
<td>10.2</td>
<td>13.9</td>
</tr>
<tr>
<td>C₁₄H₁₆</td>
<td>4.5</td>
<td>16.7</td>
<td>21.6</td>
</tr>
<tr>
<td>C₂₀</td>
<td>8.8</td>
<td>29.9</td>
<td>40.3</td>
</tr>
<tr>
<td>C₁₆H₁₈</td>
<td>10.5</td>
<td>35.9</td>
<td>50.2</td>
</tr>
<tr>
<td>C₁₈H₁₂</td>
<td>12.7</td>
<td>42.2</td>
<td>50.3</td>
</tr>
<tr>
<td>C₁₈H₂₀</td>
<td>20.1</td>
<td>73.0</td>
<td>86.6</td>
</tr>
</tbody>
</table>
CCD Performance Summary

- C1060 SP
- C1060 DP
- C2050 SP
- C2050 DP
- Xeon X5550 SP
- Xeon X5550 DP

Performance (gigaflop/s) vs. molecule: C8H10, C10H8, C10H12, C12H14, C14H10, C14H16, C20, C16H18, C18H12, C18H20

Jeff Hammond

Electronic Structure Calculation Methods on Accelerators
### Numerical Precision versus Performance

#### Iteration time in seconds

<table>
<thead>
<tr>
<th>molecule</th>
<th>C1060</th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>SP</td>
<td>DP</td>
<td>SP</td>
<td>DP</td>
<td>SP</td>
<td>DP</td>
<td>SP</td>
<td>DP</td>
</tr>
<tr>
<td>C$<em>8$H$</em>{10}$</td>
<td>0.2</td>
<td>0.8</td>
<td>0.2</td>
<td>0.3</td>
<td>0.7</td>
<td>1.3</td>
<td></td>
<td></td>
</tr>
<tr>
<td>C$_{10}$H$_8$</td>
<td>0.4</td>
<td>1.5</td>
<td>0.2</td>
<td>0.5</td>
<td>1.3</td>
<td>2.5</td>
<td></td>
<td></td>
</tr>
<tr>
<td>C$<em>{10}$H$</em>{12}$</td>
<td>0.7</td>
<td>2.5</td>
<td>0.4</td>
<td>0.8</td>
<td>2.0</td>
<td>3.5</td>
<td></td>
<td></td>
</tr>
<tr>
<td>C$<em>{12}$H$</em>{14}$</td>
<td>1.8</td>
<td>7.1</td>
<td>1.0</td>
<td>2.0</td>
<td>5.6</td>
<td>10.0</td>
<td></td>
<td></td>
</tr>
<tr>
<td>C$<em>{14}$H$</em>{10}$</td>
<td>2.6</td>
<td>10.2</td>
<td>1.5</td>
<td>2.7</td>
<td>8.4</td>
<td>13.9</td>
<td></td>
<td></td>
</tr>
<tr>
<td>C$<em>{14}$H$</em>{16}$</td>
<td>4.1</td>
<td>16.7</td>
<td>2.4</td>
<td>4.5</td>
<td>12.1</td>
<td>21.6</td>
<td></td>
<td></td>
</tr>
<tr>
<td>C$_{20}$</td>
<td>6.7</td>
<td>29.9</td>
<td>4.1</td>
<td>8.8</td>
<td>22.3</td>
<td>40.3</td>
<td></td>
<td></td>
</tr>
<tr>
<td>C$<em>{16}$H$</em>{18}$</td>
<td>9.0</td>
<td>35.9</td>
<td>5.0</td>
<td>10.5</td>
<td>28.8</td>
<td>50.2</td>
<td></td>
<td></td>
</tr>
<tr>
<td>C$<em>{18}$H$</em>{12}$</td>
<td>10.1</td>
<td>42.2</td>
<td>5.6</td>
<td>12.7</td>
<td>29.4</td>
<td>50.3</td>
<td></td>
<td></td>
</tr>
<tr>
<td>C$<em>{18}$H$</em>{20}$</td>
<td>17.2</td>
<td>73.0</td>
<td>10.1</td>
<td>20.1</td>
<td>47.0</td>
<td>86.6</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Almost no work on single-precision or mixed-precision for standard CPU packages even though it is worth 2x.
CCSD Equations

\[ t_{ij}^{ab} d_{ij}^{ab} = v_{ij}^{ab} + P(ia, jb)R_{ij}^{ab} \]

\[ R_{ij}^{ab} = \frac{1}{2} v_{ij}^{ab} c_{ij}^{ef} \]

\[ R_{ij}^{ab} = \frac{1}{2} c_{mn}^{ab} t_{ij}^{mn} \]

\[ R_{ij}^{ab} = t_{mj}^{ae} I_{ie}^{mb} + I_{ie}^{ma} t_{mj}^{eb} \]

\[ I_{ij}^{ab} = v_{ij}^{ab} c_{ij}^{ef} + P(ik, jl) t_{ik}^{ef} v_{el}^{ij} \]

\[ I_{ij}^{ia} = v_{ij}^{ia} - \frac{1}{2} v_{ij}^{im} (t_{jm}^{ea} + 2 t_{jm}^{ea}) + v_{ij}^{im} t_{jm}^{ea} - v_{ij}^{ie} t_{jm}^{eb} - v_{ij}^{im} t_{jm}^{eb} \]

\[ I_{ij}^{ia} = v_{ij}^{ia} + v_{ij}^{im} t_{jm}^{eb} - v_{ij}^{im} t_{jm}^{eb} \]

\[ I_{ij}^{ia} = v_{ij}^{ia} + v_{ij}^{im} t_{jm}^{eb} - v_{ij}^{im} t_{jm}^{eb} \]

\[ I_{ij}^{ia} = v_{ij}^{ia} + v_{ij}^{im} t_{jm}^{eb} - v_{ij}^{im} t_{jm}^{eb} \]

\[ I_{ij}^{ia} = v_{ij}^{ia} + v_{ij}^{im} t_{jm}^{eb} - v_{ij}^{im} t_{jm}^{eb} \]

\[ I_{ij}^{ia} = v_{ij}^{ia} + v_{ij}^{im} t_{jm}^{eb} - v_{ij}^{im} t_{jm}^{eb} \]

\[ I_{ij}^{ia} = v_{ij}^{ia} + v_{ij}^{im} t_{jm}^{eb} - v_{ij}^{im} t_{jm}^{eb} \]

\[ I_{ij}^{ia} = v_{ij}^{ia} + v_{ij}^{im} t_{jm}^{eb} - v_{ij}^{im} t_{jm}^{eb} \]

\[ I_{ij}^{ia} = v_{ij}^{ia} + v_{ij}^{im} t_{jm}^{eb} - v_{ij}^{im} t_{jm}^{eb} \]

\[ I_{ij}^{ia} = v_{ij}^{ia} + v_{ij}^{im} t_{jm}^{eb} - v_{ij}^{im} t_{jm}^{eb} \]

\[ I_{ij}^{ia} = v_{ij}^{ia} + v_{ij}^{im} t_{jm}^{eb} - v_{ij}^{im} t_{jm}^{eb} \]

\[ \pi_i^a = \pi_i^a + R_i^a \]

\[ R_i^a = \pi_i^a t_e^e \]

\[ R_i^a = \pi_i^a t_e^e \]

\[ R_i^a = \pi_i^a t_e^e \]

\[ R_i^a = \pi_i^a t_e^e \]

\[ R_i^a = \pi_i^a t_e^e \]

\[ R_i^a = \pi_i^a t_e^e \]

\[ R_i^a = \pi_i^a t_e^e \]
CCSD Algorithm

Guiding principles:

- Too many arrays to fit into GPU memory.
- Copy-in every iteration in CCD was not a problem.
- Want multi-GPU, mixed CPU-GPU algorithms.

Design:

- Persistent buffers but push all large arrays every iteration.
- $O(N^6)$, $O(N^5)$ on GPU.
- $O(N^5)$, $O(N^4)$ on CPU.
- Dynamically schedule some diagrams each iteration to load-balance.
- Overlap computation and communication with CUDA streams (CUBLAS compatible now).
Hybrid CCSD

<table>
<thead>
<tr>
<th></th>
<th>Hybrid</th>
<th>CPU</th>
<th>Molpro</th>
<th>NWChem</th>
<th>PSI3</th>
<th>TCE</th>
<th>GAMESS</th>
</tr>
</thead>
<tbody>
<tr>
<td>C&lt;sub&gt;8&lt;/sub&gt;H&lt;sub&gt;10&lt;/sub&gt;</td>
<td>0.6</td>
<td>1.4</td>
<td>2.4</td>
<td>3.6</td>
<td>7.9</td>
<td>8.4</td>
<td>7.2</td>
</tr>
<tr>
<td>C&lt;sub&gt;10&lt;/sub&gt;H&lt;sub&gt;8&lt;/sub&gt;</td>
<td>0.9</td>
<td>2.6</td>
<td>5.1</td>
<td>8.2</td>
<td>17.9</td>
<td>16.8</td>
<td>15.3</td>
</tr>
<tr>
<td>C&lt;sub&gt;10&lt;/sub&gt;H&lt;sub&gt;12&lt;/sub&gt;</td>
<td>1.4</td>
<td>4.1</td>
<td>7.2</td>
<td>11.3</td>
<td>23.6</td>
<td>25.2</td>
<td>23.6</td>
</tr>
<tr>
<td>C&lt;sub&gt;12&lt;/sub&gt;H&lt;sub&gt;14&lt;/sub&gt;</td>
<td>3.3</td>
<td>11.1</td>
<td>19.0</td>
<td>29.4</td>
<td>54.2</td>
<td>64.4</td>
<td>65.1</td>
</tr>
<tr>
<td>C&lt;sub&gt;14&lt;/sub&gt;H&lt;sub&gt;10&lt;/sub&gt;</td>
<td>4.4</td>
<td>15.5</td>
<td>31.0</td>
<td>49.1</td>
<td>61.4</td>
<td>90.7</td>
<td>92.9</td>
</tr>
<tr>
<td>C&lt;sub&gt;14&lt;/sub&gt;H&lt;sub&gt;16&lt;/sub&gt;</td>
<td>6.3</td>
<td>24.1</td>
<td>43.1</td>
<td>65.0</td>
<td>103.4</td>
<td>129.2</td>
<td>163.7</td>
</tr>
<tr>
<td>C&lt;sub&gt;20&lt;/sub&gt;</td>
<td>10.5</td>
<td>43.2</td>
<td>102.0</td>
<td>175.7</td>
<td>162.6</td>
<td>233.9</td>
<td>277.5</td>
</tr>
<tr>
<td>C&lt;sub&gt;16&lt;/sub&gt;H&lt;sub&gt;18&lt;/sub&gt;</td>
<td>10.0</td>
<td>38.9</td>
<td>84.1</td>
<td>117.5</td>
<td>192.4</td>
<td>267.9</td>
<td>345.8</td>
</tr>
<tr>
<td>C&lt;sub&gt;18&lt;/sub&gt;H&lt;sub&gt;12&lt;/sub&gt;</td>
<td>14.1</td>
<td>57.1</td>
<td>116.2</td>
<td>178.6</td>
<td>216.4</td>
<td>304.5</td>
<td>380.0</td>
</tr>
<tr>
<td>C&lt;sub&gt;18&lt;/sub&gt;H&lt;sub&gt;20&lt;/sub&gt;</td>
<td>22.5</td>
<td>95.9</td>
<td>161.4</td>
<td>216.3</td>
<td>306.9</td>
<td>512.0</td>
<td>641.3</td>
</tr>
</tbody>
</table>

We do at least twice as many flops as Molpro due to CC formalism.
### More hybrid CCSD

<table>
<thead>
<tr>
<th>molecule</th>
<th>Basis</th>
<th>$o$</th>
<th>$v$</th>
<th>Hybrid</th>
<th>CPU</th>
<th>Molpro</th>
<th>CPU</th>
<th>Molpro</th>
</tr>
</thead>
<tbody>
<tr>
<td>CH$_3$OH</td>
<td>aTZ</td>
<td>7</td>
<td>175</td>
<td>2.5</td>
<td>4.5</td>
<td>2.8</td>
<td>1.8</td>
<td>1.1</td>
</tr>
<tr>
<td>benzene</td>
<td>aDZ</td>
<td>15</td>
<td>171</td>
<td>5.1</td>
<td>14.7</td>
<td>17.4</td>
<td>2.9</td>
<td>3.4</td>
</tr>
<tr>
<td>C$_2$H$_6$SO$_4$</td>
<td>aDZ</td>
<td>23</td>
<td>167</td>
<td>9.0</td>
<td>33.2</td>
<td>31.2</td>
<td>3.7</td>
<td>3.5</td>
</tr>
<tr>
<td>C$<em>{10}$H$</em>{12}$</td>
<td>DZ</td>
<td>26</td>
<td>164</td>
<td>10.7</td>
<td>39.5</td>
<td>56.8</td>
<td>3.7</td>
<td>5.3</td>
</tr>
<tr>
<td>C$<em>{10}$H$</em>{12}$</td>
<td>6-31G</td>
<td>26</td>
<td>78</td>
<td>1.4</td>
<td>4.1</td>
<td>7.2</td>
<td>2.9</td>
<td>5.1</td>
</tr>
</tbody>
</table>

Molpro optimized for $v/o \gg 1$.

Our code doesn’t favor any limit except $o, v \gg 1$.  

---

**Jeff Hammond**

*Electronic Structure Calculation Methods on Accelerators*
Programming Models
Categorizing parallel models

- **MPI-1** Private address space, explicit communication, independent execution.
- **MPI-RMA** Private-ish address space, explicit communication, independent execution.
- **OpenMP (loops)** Shared address space, implicit communication, collective execution.
- **OpenMP (sections)** Shared address space, implicit communication, semi-independent execution.
- **Pthreads** Shared address space, independent execution.
- **GA** Global view of data (not coherent), explicit communication, independent execution.
- **PGAS** Shared address space, implicit/explicit communication, independent execution.

OpenMP is always implemented on top of Pthreads.
PGAS can use MPI and/or Pthreads.
Categorizing models

- Execution model: independent/tasks vs. cooperative/data-parallel
- Memory sharing: private by default vs. public by default
- Threading: parallel by default vs. serial by default
MPI exists in a constructive cycle of ubiquity and optimization.

OpenMP is reaching ubiquity; optimization is debatable.

Pthreads is ubiquitous but invisible (system programmers only).

PGAS portability, especially performance portability, is still emerging.
OpenMP: from 1 to N or N to 1?
OpenMP: from 1 to N or N to 1?

```c
#pragma omp parallel
{ /* thread-safe */ }

#pragma omp single
{ /* thread-unsafe */ }

#pragma omp parallel for
{ /* threaded loops */ }

#pragma omp sections
{ /* threaded work */ }
```

```c
{ /* thread-unsafe */ }
#pragma omp parallel for
{ /* threaded loops */ }
{ /* thread-unsafe */ }
#pragma omp parallel for
{ /* threaded loops */ }
#pragma omp parallel for
{ /* thread-unsafe work */ }
```
Addition thoughts

- Pthreads very useful for driving communication
- OpenMP NUMA catastrophe (on Linux
- MPI+SHM vs. Pthreads (shared vs. private question)
- Learn from linear algebra community:
  - DAG-scheduling with Pthreads not OpenMP
  - SK+KK TCE task pools in EOMCCSD, MRCC
  - our GPU-CC code using task parallelism throughout
Beyond one node

- NWChem, MADNESS, MPQC, GAMESS (?) burn core(s) for communication
- Multi-socket, multi-GPU, multi-rail all gets nasty
- Pthread support for affinity becomes more critical
- NWChem data-server uses ~ 25% of cores with Infiniband
- Do your GPUs need communication helper threads?
Pragmas
Q Can I call a CUDA kernel function from my PGI-compiled code?

A PGI is working on the design of a feature to allow you to call kernel functions written in CUDA or PTX or other languages directly from your C or Fortran program. We will announce this feature when it is available.

Simple tensor contractions are done with DGEMM; nonsimple ones redistribution to simple ones (permutation).

- CUBLAS DGEMM with CUDA permutation codes
- Pragma DGEMM with pragma permutation codes
- Pragma code only works with Fortran (C is a terrible language to optimize)
#pragma acc region
{
    for(i=0; i<num; i++)
        for(j=0; i<num; j++)
            for(k=0; i<num; k++)
                c[i*1000+j] += ( a[i*1000+k] * b[k*1000+j] );
}
42, Accelerator region ignored
44, Accelerator restriction: size of the GPU copy of an array depends on values computed in this loop
45, Accelerator restriction: size of the GPU copy of 'c' is unknown
Accelerator restriction: size of the GPU copy of an array depends on values computed in this loop
Accelerator restriction: one or more arrays have unknown size
46, Accelerator restriction: size of the GPU copy of 'a' is unknown
Accelerator restriction: size of the GPU copy of 'b' is unknown
Accelerator restriction: one or more arrays have unknown size
GotoBLAS on my Core2 laptop does 20 GF.
Pragmas

Usage:

■ Must allow native function calls
■ Pragmas to generate native object rather than offload
■ Explicit data movement

*Explicit data movement is something scientists can do!*

Things compilers cannot do well:

■ Optimize for cache (Netlib vs. Goto)
■ Generate network communication (PGAS)
■ Move data across PCI bus (GPU pragmas)

Prove me wrong...
Acknowledgments

Computational Fellowship (AED)
ANL-MCS Breadboard cluster
ANL-LCRC Fusion cluster

Dirac cluster
Shameless promotion
What: Symposium on Application Accelerators in HPC
When: July 10-12, 2012
Where: Argonne (Chicago, IL)
TCE on GPUs
NWChem TCE module

- Automatically generated code: easy to rewrite (in principle).
- Data-parallel over tiles using Global Arrays.
- Naïve dynamic load-balancing (shared counter).
- Standard GA programming model: check out, compute, check in
for (P3, P4, H1, H2) in all (P, P, H, H) tiles:
    if (my_turn) and (nonzero_symmetry):
        allocate_and_zero buffer Jc
    for (P5, P6) in all (P, P) tiles:
        if (nonzero_symmetry):
            allocate_and_zero buffer Tc
            get Tb from global T
            reorder Tc
            allocate_and_zero buffer Ic
            get Ib from global I
            reorder Ic
            compute Jc += Tc*Ic
        reorder Jc
    acc Jc onto global J
TCE GPU algorithm

```plaintext
grab_from_pool_and_zero buffer pair (Tc,Tg)
grab_from_pool_and_zero buffer pair (Ic,Ig)
if (push_gpucompute_pull < cpucompute):
    iget_and_push Tg from global T
    iget_and_push Ig from global I
    igpu_reorder Tg
    igpu_reorder Ig
    gpucompute Jg += Tg*Ig
else:
    iget Tc from global T
    iget Ic from global I
    compute Jc += Tc*Ic
ireorder Jg
reorder Jc
pull_and_acc Jc += Jg
acc Jc onto global J
```

Jeff Hammond
Electronic Structure Calculation Methods on Accelerators
Evaluating the GPU algorithm

- Tilesizes not big enough to justify GPU all the time (bad).
- $Jg$ stays on the GPU through each pass (good).
- Possibly good overlap of data movement (good).
- Can unroll inner loop for maximum overlap (good), but this doubles the memory required (bad).
- Threaded CPU code helps memory issues but GA/ARMCI not thread-safe (funneled or serialized should be okay).

Many of the GPU-oriented optimizations will help with multicore CPUs. Fully rewritten GPU TCE in NWChem will probably coincide with porting to BGQ for this reason.
Partial bibliography
All major MD packages have or will soon have a GPU implementation.


DFT on GPUs


MP2 using the resolution-of-identity approximation involves a few large GEMMs.


QMC is ridiculously parallel at the node-level hence implementation of the kernel implies the potential for scaling to many thousands of GPUs.


Since we started this project, two groups have implemented non-iterative triples corrections to CCSD on GPUs. These procedures involve a few very large GEMMs and a reduction. At least two other groups are working on CC on GPUs but have not reported any results.

Quantum chemistry methods spend a lot of time generating matrix elements of operators in a Gaussian basis set. All published implementations are either closed-source (commercial) or the source is unpublished, otherwise we would be using these in our code.

