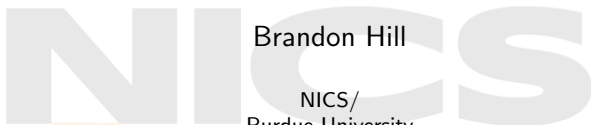


NEMO-3D: Walking through an actual GPU port

NATIONAL INSTITUTE FOR COMPUTATIONAL SCIENCES



Brandon Hill
NICS/
Purdue University
Computer Science Dept.

August 9, 2010

- 1 Porting an app
 - NEMO-3D
 - The target
 - The easy bits
 - The real work

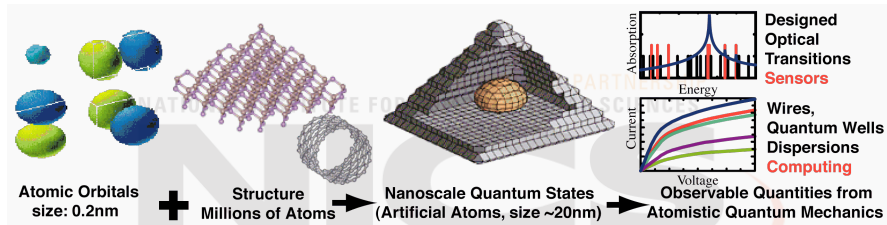
- 2 Optimization
 - Priorities
 - More involved

- 3 Issues of everyday coding
 - Issues
 - Error Messages
 - Debugging
 - Different HW
 - OpenCL
 - Compiler magic?



NEMO-3D

NEMO-3D calculates the eigenstates in arbitrarily shaped semiconductor structures to determine the available quantum energy states available.



Examining the properties of these structures requires building the Hamiltonian operator, and finding its eigenvalues.

$$\hat{H} = \sum_i \varepsilon_i^{(\nu)} c_{i,\nu}^+ c_{i,\nu} + \sum_{i,\nu,\mu} t_i^{(\nu\mu)} c_{i,\nu}^+ c_{i,\mu} + \sum_{i,j,\nu,\mu} t_{ij}^{(\nu\mu)} c_{i,\nu}^+ c_{j,\mu}$$

The Hamiltonian can have 20 row/columns for every atom modeled. Reasonably sized models can easily reach 1 billion atoms.

Profiling

Profile, profile, profile.

One million atoms, 1 processor:

%	cumulative	self		self	total	
time	seconds	seconds	calls	s/call	s/call	name
74.98	1153.81	1153.81	399	2.89	2.89	Hv_stored_20_HzbComplex()
9.82	1304.93	151.12	307	0.49	0.69	calc_estrain()
5.93	1396.15	91.22				single_lanczos_iteration()

⋮

One million atoms, 8 processors:

%	cumulative	self		self	total	
time	seconds	seconds	calls	s/call	s/call	name
69.15	156.79	156.79	399	0.39	0.39	Hv_stored_20_HzbComplex()
13.18	186.67	29.88				single_lanczos_iteration()
9.05	207.18	20.51	307	0.07	0.09	calc_estrain()

Now profile some more.

The algorithm

NEMO-3D uses a version of the Lanczos algorithm to find the eigenstates for the Hamiltonian. This is a dominant computational kernel in all forms of this code and thus the first target.

Lanczos(H)

```
v1 ← random vector with norm 1
v0 ← 0
β1 ← 0
for j = 1 to m do
  ωj ← Hvj - βjvj-1
  αj ← ωj · vj
  ωj ← ωj - αjvj
  βj+1 ← ||ωj||
  vj+1 ← ωj/βj+1
end for
```

$$\Rightarrow T_{mm} = \begin{pmatrix} \alpha_1 & \beta_2 & & & 0 \\ \beta_2 & \alpha_2 & \beta_3 & & \\ & \beta_3 & \alpha_3 & \ddots & \\ & & \ddots & \ddots & \beta_{m-1} \\ 0 & & & \beta_{m-1} & \alpha_{m-1} & \beta_m \\ & & & & \beta_m & \alpha_m \end{pmatrix}$$

$V_m = (v_1, v_2, \dots, v_m)$

Find the eigenvalues $\lambda_i^{(m)}$ and eigenvectors $u_i^{(m)}$ of T_{mm} .

Solve $y_i = V_m u_i^{(m)}$, where y_i are the Ritz eigenvectors of H .

Using the known layout of the atoms, the iterative matrix-vector multiplication in this Lanczos algorithm is reduced to item-wise vector-vector multiplications that are easily split across thousands of processors.

70% execution time is spent on these vector-vector multiplications.

Target Function

Hv_stored_20_HzbComplex(args,...)

for $i = 1$ to numAtoms*numBasisStates **do**

$y[i].r += Hdd[i] * x[i].r$

$y[i].i += Hdd[i] * x[i].i$

end for

⋮

for $i = 1$ to numAtoms*numBasisStates **do**

$idx_a \leftarrow$ some val

$idx_b \leftarrow idx_b + 1$

$y[idx_a].r += Hdu[idx_b].r * x[idx_a].r - Hdu[idx_b].i * x[idx_a].i$

$y[idx_a].i += Hdu[idx_b].r * x[idx_a].i + Hdu[idx_b].i * x[idx_a].r$

end for

⋮

for $i = 1$ to numOffProcNeighbors **do**

$kdotl \leftarrow$ phaseBloch()

for $j = 1$ to numBasisStates **do**

$idx_a \leftarrow$ some val

for $k = 1$ to numBasisStates **do**

$idx_b \leftarrow$ some val

$y[idx_a] += Hu[idx].r * x[idx_b].r - Hu[idx].i * x[idx_b].i$

$y[idx_a] += Hu[idx].r * x[idx_b].b + Hu[idx].i * x[idx_b].r$

$y[idx_b] += Hu[idx].r * x[idx_a].r - Hu[idx].i * x[idx_a].i$

$y[idx_b] += Hu[idx].r * x[idx_a].b + Hu[idx].i * x[idx_a].r$

$idx \leftarrow idx + 1$

end for

end for

end for

Starting the port

Many first-pass translations end up being trivial:

```
for  $i = 1$  to numAtoms*numBasisStates do
```

```
   $y[i].r+ = Hdd[i] * x[i].r$ 
```

```
   $y[i].i+ = Hdd[i] * x[i].i$ 
```

```
end for
```

A UI/ORNL PARTNERSHIP
NATIONAL INSTITUTE FOR COMPUTATIONAL SCIENCES

Becomes:

```
int i = blockIdx.x * blockDim.x + threadIdx.x;
```

```
int N = nAtom * nBstates;
```

```
/* y += Hdd * x */
```

```
if( i < N ) {
```

```
   $y[i].x += H[i] * x[i].x;$ 
```

```
   $y[i].y += H[i] * x[i].y;$ 
```

```
}
```

Some times the translation seems obvious, but don't get sloppy when using loops as the default item to spread a thread over. Loops often have side effects.

```
Hc_IJ = &d->HzbComplex.Hdu[0];
for (atom=0; atom<d->HzbComplex.Natom; atom++) {
    int offset = atom*d->NBasisStates;
    int I,J;
```

This:

```
I = offset + 2; J = offset + 3;
y[I].r += Hc_IJ->r * x[J].r - Hc_IJ->i * x[J].i;
y[I].i += Hc_IJ->r * x[J].i + Hc_IJ->i * x[J].r;
y[J].r += Hc_IJ->r * x[I].r + Hc_IJ->i * x[I].i;
y[J].i += Hc_IJ->r * x[I].i - Hc_IJ->i * x[I].r;
Hc_IJ++;
```

Doesn't become:

```
int i = blockIdx.x * blockDim.x + threadIdx.x;
int offset, I, J;
```

```
if( i < nAtom ) {
    offset = i*nBstates;
```

```
I = offset + 2; J = offset + 3;
y[I].x += Hc_IJ->x * x[J].x - Hc_IJ->y * x[J].y;
y[I].y += Hc_IJ->x * x[J].y + Hc_IJ->y * x[J].x;
y[J].x += Hc_IJ->x * x[I].x + Hc_IJ->y * x[I].y;
y[J].y += Hc_IJ->x * x[I].y - Hc_IJ->y * x[I].x;
```

Consider:

```
for  $i = 1$  to numOffProcNeighbors do  
   $kdotl \leftarrow$  phaseBloch( $i, L$ )  
  for  $j = 1$  to numBasisStates do  
     $idx\_a \leftarrow$  HuRows[ $k$ ] +  $j$   
    for  $k = 1$  to numBasisStates do  
       $idx\_b \leftarrow$  HuCols[ $k$ ] +  $k$   
       $Hu[idx].r \leftarrow$   $\cos(kdotl) * Hu[idx].r - \sin(kdotl) * Hu[idx].i$   
       $Hu[idx].i \leftarrow$   $\sin(kdotl) * Hu[idx].r - \cos(kdotl) * Hu[idx].i$   
       $y[idx\_a].r += Hu[idx].r * x[idx\_b].r - Hu[idx].i * x[idx\_b].i$   
       $y[idx\_a].i += Hu[idx].r * x[idx\_b].i + Hu[idx].i * x[idx\_b].r$   
       $y[idx\_b].r += Hu[idx].r * x[idx\_a].r - Hu[idx].i * x[idx\_a].i$   
       $y[idx\_b].i += Hu[idx].r * x[idx\_a].i + Hu[idx].i * x[idx\_a].r$   
       $idx \leftarrow idx + 1$   
    end for  
  end for  
end for
```

While a first-pass port might look like:

```
int i = blockIdx.x * blockDim.x + threadIdx.x;
int j,k;

if( i < numOffProcNeighbors ) {
  for(j=0; j<nBstates;j++) {
    idx_a = Hu_row[i] + j;
    for(k=0;k<nBstates;k++) {
      idx_b = Hu_col[i] + k;

      Hu[idx].r = cos(kdotl[i]) * Hu[idx].r - sin(kdotl[i]) * Hu[idx].i;
      Hu[idx].i = sin(kdotl[i]) * Hu[idx].r - cos(kdotl[i]) * Hu[idx].i;
      y[idx_a].x += Hu[idx].x * x[idx_b].x - Hu[idx].y * x[idx_b].y;
      y[idx_a].y += Hu[idx].x * x[idx_b].y + Hu[idx].y * x[idx_b].x;
      y[idx_b].x += Hu[idx].x * x[idx_a].x - Hu[idx].y * x[idx_a].y;
      y[idx_b].y += Hu[idx].x * x[idx_a].y + Hu[idx].y * x[idx_a].x;
      idx++;
    }
  }
}
```

It returns the wrong answer. Why?

Data dependencies: *idx_a* and *idx_b* both have repeated values. Two threads updating the same *y*[] location at the same time collide and only one result is written back.

Code like this can require that data structures are rewritten.

- ▶ Bundle all updates so a single thread handles all updates
- ▶ Actually accelerates CPU version of the code as well
 - ▶ Many GPU tunings are tricks that work for CPU's as well, but weren't required
- ▶ This can be a big part of the cost of a port since a kernel's needs require changes that force updates throughout the code base.

Resolution

New data structure and code:

Before first iteration:

- 1 cast `Hu_rows` and `Hu_cols` as Thrust STL vectors
- 2 create a copy sorted by `Hu_rows` and another by `Hu_cols`
- 3 create a prefix count array that tracks the offsets of unique indices for each vector
- 4 pre-solve for all `kdotl` values
 - ▶ All items are stores as a Structure of Arrays (SoA) instead of an Array of Structures (AoS)

For each iteration:

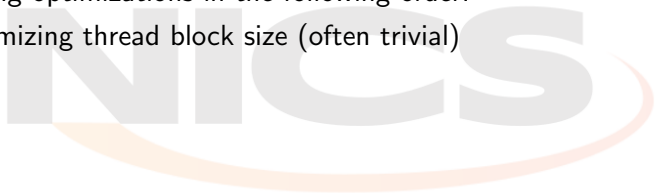
- 1 each thread solves all updates for each item in the sorted `Hu_rows` set
 - ▶ the loop for each index is unrolled for all the basis states
- 2 repeat for each item in the sorted `Hu_cols` set

Typical benefits for optimization types

What comes after the first pass?

Recent studies are finding the biggest returns can typically be found by approaching optimizations in the following order:

- 1 Maximizing thread block size (often trivial)



Typical benefits for optimization types

What comes after the first pass?

Recent studies are finding the biggest returns can typically be found by approaching optimizations in the following order:

- 1 Maximizing thread block size (often trivial)
- 2 Loop-unrolling

Typical benefits for optimization types

What comes after the first pass?

Recent studies are finding the biggest returns can typically be found by approaching optimizations in the following order:

- 1 Maximizing thread block size (often trivial)
- 2 Loop-unrolling
 - ▶ though too much can over run register space

Typical benefits for optimization types

What comes after the first pass?

Recent studies are finding the biggest returns can typically be found by approaching optimizations in the following order:

- 1 Maximizing thread block size (often trivial)
- 2 Loop-unrolling
 - ▶ though too much can over run register space
 - ▶ can reduce shared memory conflicts

Typical benefits for optimization types

What comes after the first pass?

Recent studies are finding the biggest returns can typically be found by approaching optimizations in the following order:

- 1 Maximizing thread block size (often trivial)
- 2 Loop-unrolling
 - ▶ though too much can over run register space
 - ▶ can reduce shared memory conflicts
 - ▶ some are using the Boost preprocessor to do this in an automated fashion

Typical benefits for optimization types

What comes after the first pass?

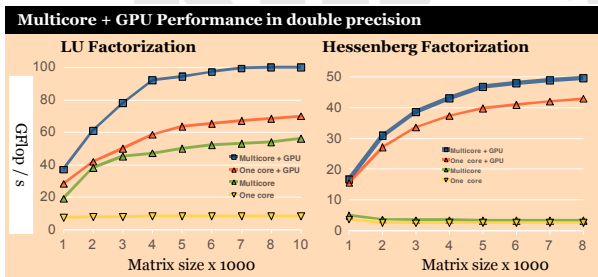
Recent studies are finding the biggest returns can typically be found by approaching optimizations in the following order:

- 1 Maximizing thread block size (often trivial)
- 2 Loop-unrolling
 - ▶ though too much can over run register space
 - ▶ can reduce shared memory conflicts
 - ▶ some are using the Boost preprocessor to do this in an automated fashion
- 3 Using shared memory

Other optimizations

Co-computing This is the ideal. So far the only major code I have seen doing this is MAGMA. Some codes can't, others may require too much tuning.

- ▶ specialized work splitting - giving the CPU the work it is best at (e.g. easily vectorized code)
- ▶ work-share splitting - give the CPU a share that should take as long as the GPU share



Jack Dongarra,
Stan Tomov,
and Rajib Nath

GPU : NVIDIA GeForce GTX 280

CPU : Intel Xeon dual socket quad-core @2.33 GHz

GPU BLAS : CUBLAS 2.2, dgemm peak: 75 GFlop/s

CPU BLAS : MKL 10.0 , dgemm peak: 65 GFlop/s

Other optimizations

Co-computing This is the ideal. So far the only major code I have seen doing this is MAGMA. Some codes can't, others may require too much tuning.

- ▶ specialized work splitting - giving the CPU the work it is best at (e.g. easily vectorized code)
- ▶ work-share splitting - give the CPU a share that should take as long as the GPU share

Mixed Precision Several projects are getting around the limited floating point abilities of cards by looking at what parts of an algorithm are actually sensitive to precision issues. Much more flexibility exists than many imagine, and the performance gains can be significant.

Other optimizations

Co-computing This is the ideal. So far the only major code I have seen doing this is MAGMA. Some codes can't, others may require too much tuning.

- ▶ specialized work splitting - giving the CPU the work it is best at (e.g. easily vectorized code)
- ▶ work-share splitting - give the CPU a share that should take as long as the GPU share

Mixed Precision Several projects are getting around the limited floating point abilities of cards by looking at what parts of an algorithm are actually sensitive to precision issues. Much more flexibility exists that many imagine, and the performance gains can be significant.

GPU libraries Libraries like CUBLAS, MAGMA, and Thrust (STL) are all making it much easier to write fewer kernels.

Other optimizations

Co-computing This is the ideal. So far the only major code I have seen doing this is MAGMA. Some codes can't, others may require too much tuning.

- ▶ specialized work splitting - giving the CPU the work it is best at (e.g. easily vectorized code)
- ▶ work-share splitting - give the CPU a share that should take as long as the GPU share

Mixed Precision Several projects are getting around the limited floating point abilities of cards by looking at what parts of an algorithm are actually sensitive to precision issues. Much more flexibility exists that many imagine, and the performance gains can be significant.

GPU libraries Libraries like CUBLAS, MAGMA, and Thrust (STL) are all making it much easier to write fewer kernels.

Driver upgrades Don't ignore driver upgrades! New features may change how you want to write your kernels.

Everyday things

Silly gotchas:

- ▶ Remember that double precision is turned off by default. System silently converts values to floats. Enable with `--arch sm_13` when compiling and `f` after all float literals.
- ▶ Other options like ECC also need to be turned on or off.
- ▶ If you are developing on your own machine, remember that kernel updates require that the driver is reinstalled.

Scary gotchas:

- ▶ Some memory allocation errors can lock a card enough that hard reboots are required. Using a non-production development machine is advised.

Error Messages

CUDA error messages don't always relate to the real problem. Sometimes they do, sometimes they don't. Code and test incrementally to get the best idea of the cause.

Unspecified launch failure Usually a seg. fault, but really a wild card
Memory size or pointer value too large to fit into 32bits Often thrown for odd things (e.g. caused by an unneeded free, thrown by CudaMalloc())

Invalid Arguments Illegal number of threads

Invalid Configuration Arguments Illegal dimensions to a block or grid

Remember to find your card limits from deviceQuery:

Maximum number of threads per block:	512
Maximum sizes of each dimension of a block:	512 x 512 x 64
Maximum sizes of each dimension of a grid:	65535 x 65535 x 1

Debugging Options

- ▶ Free tools
- ▶ Alinea's DDT
- ▶ Totalview

A /ORNL PARTNERSHIP
NATIONAL INSTITUTE FOR COMPUTATIONAL SCIENCES

NICS

What you get for free

Without a commercial debugger, programmers are currently stuck with only primitive debugging tools.

debug variables Create additional variables to export and examine intermediate values in host code

- ▶ Work intensive

printf() With the latest version (3.1) of the CUDA library comes an old standby.

Commercial debuggers

In general

- ▶ All commercial debuggers rely on the interface that Nvidia provides, so features may not significantly differ.
- ▶ GT200 based cards don't allow thread continuation. Kernels are paused all together. Fermi allows single threads to be paused.



Commercial debuggers

DDT

- ▶ Easy to use, brings standard debugging tools to GPU
- ▶ Inspect values and individual threads
- ▶ Makes subtle problems much easier to detect

The screenshot displays the Alinea Distributed Debugging Tool (DDT) v2.6 interface. The main window shows the source code for `Hmult_stored_cuda.cu` at line 110. The code is as follows:

```
110 float *H_Ip = H_IJ[1*5];
111 y[i].x += (*H_Ip) * x[j].y;
112 y[j].x += (*H_Ip) * x[i].y;
113 y[i].y += (*H_Ip) * x[j].x;
114 y[j].y += (*H_Ip) * x[i].x;
115
116 ii = offset + 1;
117 jj = offset + 2;
118 float *H_Ip = H_IJ[1*5];
119 y[i].x += (*H_Ip) * x[j].y;
120 y[j].x += (*H_Ip) * x[i].y;
121 y[i].y += (*H_Ip) * x[j].x;
122 y[j].y += (*H_Ip) * x[i].x;
```

The interface includes a top menu bar (Session, Control, Search, View, Help), a toolbar, and a status bar. The 'Threads' panel shows two threads, with the first thread selected. The 'CUDA Threads (Process 0)' panel shows the current thread's state: Block 0, Thread 32, Grid size 254x1, Block size 512x1x1. The 'Project Files' panel shows the file structure, with `Hmult_stored_cuda.cu` selected. The 'Locals' panel shows the current thread's local variables:

Variable Name	Value
-H	0x510000
-nAtom	6480
-nBstates	20
-x	0x310000
-y	0x110000
-yHx	0x4eeae6 <yHx(int, dot

The 'Stacks' panel shows the call stack, with the current thread's function `Hmult_stored_20_HzbComplex_cuda` selected. The 'Evaluate' panel is empty.

Commercial debuggers

In general

- ▶ All commercial debuggers rely on the interface that Nvidia provides, so features may not significantly differ.
- ▶ GT200 based cards don't allow thread continuation. Kernels are paused all together. Fermi allows single threads to be paused.

DDT

- ▶ Easy to use, brings standard debugging tools to GPU
- ▶ Inspect values and individual threads
- ▶ Makes subtle problems much easier to detect

Totalview

Totalview still in beta, but presumed to have similar behaviors.

Fermi vs GT200

The differences between the Fermi and GT200 architectures are large enough that many codes only really make sense on the new architectures.

Real Floating Point 8x more FPUs - IEEE 754-2008 compliant - Fused Multiply Add - 16-bit fp memory format - all rounding modes - Double precision now only half the speed of single precision (vs 1/10th in GT100)

64-bit Virtual Address Space More memory

Faster Atomic Functions Enables more complex kernels to be accelerated

ECC Better accuracy

What matters for the code

New features that change how you might develop:

Caches 48KB first level caches (per SM) and 768 KB shared second level caches - Still less than the total register space (128 KB / SM), but critical for some codes

Concurrent kernels More than one kernel can run at once

Better debugging support Real traps, breakpoints, etc.

Writing for both cards

Since Fermi cards are hard to find, writing for both cards is necessary:

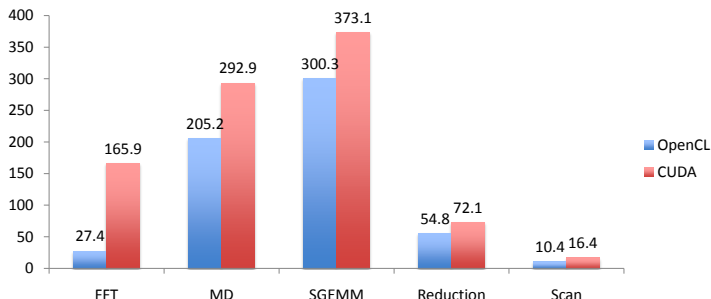
```
#if __CUDA_ARCH__ == 100
    // Code for older cards
#elif __CUDA_ARCH__ == 200
    // Code for Fermi cards
#elif !defined(__CUDA_ARCH__)
    // Code for the CPU
#endif
```

It is also wise call to `deviceQuery` and parse the result for expected/changeable values in the makefile.

Portability and OpenCL

What about OpenCL?

- ▶ Current studies are finding at least 20% slowdowns compared to CUDA



Tesla C1060, Single Precision, CUDA, and OpenCL 3.0

Portability and OpenCL

What about OpenCL?

- ▶ Current studies are finding at least 20% slowdowns compared to CUDA
- ▶ Better results require the code being highly tuned to one platform or another
- ▶ Portable performance looks to be a long ways of
 - ▶ Most current HW advances all require coding to take advantage off them

Bottom line: For now, portability and high performance are competing goals

PGI's attempt to make your life easier

Much like the pragma's that OpenMP uses to add easy multi-core parallelization to loops, Portland Group is offering accelerator commands to allow sections of code to be auto-tuned for the GPU:

```
!$acc region
do k = 1,n1
  do i = 1,n3
    c(i,k) = 0.0
    do j = 1,n2
      c(i,k) = c(i,k) + a(i,j) * b(j,k)
    enddo
  enddo
enddo
!$acc end region
```

For those with little time to optimize and experiment this may very well be worth a try.