ME964
High Performance Computing
for Engineering Applications

Wrap-up, thrust
CUDA GPU Computing Ecosystem
GPU Computing/CUDA Wrap-up
March 27, 2012

“The first 90 percent of the code accounts for the first 90 percent of the development time. The remaining 10 percent of the code accounts for the other 90 percent of the development time.”
—Tom Cargill
Before We Get Started...

- Last time
  - Wrap up CUDA streams and applications
  - GPU programming with **thrust**

- Today
  - Wrap up GPU programming with **thrust**
  - GPU computing with compiler directives
  - Brief overview of CUDA ecosystem
  - GPU programming with CUDA wrap-up

- Other issues
  - Assignment 9 due on Sunday, April 1 at 23:59 pm
    - Related to **thrust**
  - Preliminary Midterm Project due on Th, March 29 at 11:59 PM
ME964 Project Related Issues…

- Midterm Project progress report: Up to two page write-up
  - Due March 29 at 11:59 PM

- Midterm Project progress write-up: Needs to include descriptions of
  - [actual problem you solve] + [software & algorithm design solution]
    - What’s input & what’s output
    - How the output is obtained from the input [the algorithm]
    - How the algorithm is mapped onto the underlying hardware [software design]
    - If you use pictures (encouraged) you can use up to three pages

- Midterm Project Due Date: April 12, 11:59 PM

- Speaking of ME964 projects
  - Final Project proposal due on April 10
  - Can be a continuation of the Midterm Project

- Midterm & Final Projects: you can work individually or in groups of two
Leveraging Parallel Primitives
[Cntd., previous lecture]

- Test: sort 32M keys on each platform
  - Performance measured in millions of keys per second [higher is better]
- Conclusion: Use sort liberally, it’s highly optimized

<table>
<thead>
<tr>
<th>data type</th>
<th>std::sort</th>
<th>tbb::parallel_sort</th>
<th>thrust::sort</th>
</tr>
</thead>
<tbody>
<tr>
<td>char</td>
<td>25.1</td>
<td>68.3</td>
<td>3532.2</td>
</tr>
<tr>
<td>short</td>
<td>15.1</td>
<td>46.8</td>
<td>1741.6</td>
</tr>
<tr>
<td>int</td>
<td>10.6</td>
<td>35.1</td>
<td>804.8</td>
</tr>
<tr>
<td>long</td>
<td>10.3</td>
<td>34.5</td>
<td>291.4</td>
</tr>
<tr>
<td>float</td>
<td>8.7</td>
<td>28.4</td>
<td>819.8</td>
</tr>
<tr>
<td>double</td>
<td>8.5</td>
<td>28.2</td>
<td>358.9</td>
</tr>
</tbody>
</table>

Intel Core i7 950 @3.07 GHz
NVIDIA GeForce 480
Input-Sensitive Optimizations

Sorting 32M unsigned integers (uniformly distributed) with different numbers of occupied key bits.
For example, Key Bits = 20 means all keys are in the range [0, 2^{20})
# thrust: partial list of algorithms supported

<table>
<thead>
<tr>
<th>Algorithm</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>reduce</td>
<td>Sum of a sequence</td>
</tr>
<tr>
<td>find</td>
<td>First position of a value in a sequence</td>
</tr>
<tr>
<td>mismatch</td>
<td>First position where two sequences differ</td>
</tr>
<tr>
<td>inner_product</td>
<td>Dot product of two sequences</td>
</tr>
<tr>
<td>equal</td>
<td>Whether two sequences are equal</td>
</tr>
<tr>
<td>min_element</td>
<td>Position of the smallest value</td>
</tr>
<tr>
<td>count</td>
<td>Number of instances of a value</td>
</tr>
<tr>
<td>is_sorted</td>
<td>Whether sequence is in sorted order</td>
</tr>
<tr>
<td>transform_reduce</td>
<td>Sum of transformed sequence</td>
</tr>
</tbody>
</table>
## General Transformations

<table>
<thead>
<tr>
<th>Transformation</th>
<th>Code</th>
</tr>
</thead>
</table>
| **Unary Transformation** | `for (int i = 0; i < N; i++)
X[i] = f(A[i]);` |
| **Binary Transformation** | `for (int i = 0; i < N; i++)
X[i] = f(A[i], B[i]);` |
| **Ternary Transformation** | `for (int i = 0; i < N; i++)
X[i] = f(A[i], B[i], C[i]);` |
| **General Transformation** | `for (int i = 0; i < N; i++)
X[i] = f(A[i], B[i], C[i], ...);` |

- Like the STL, **thrust** provides built-in support for unary and binary transformations.
- Transformations involving 3 or more input ranges must use a different approach.
General Transformations Preamble:
The Zipping Operation

Multiple Distinct Sequences

Unique Sequence of Tuples

zip_iterator
Example: General Transformations

```cpp
#include <thrust/device_vector.h>
#include <thrust/transform.h>
#include <thrust/iterator/zip_iterator.h>
#include <iostream>

struct linear_combo {
    __host__ __device__
    float operator()(thrust::tuple<float, float, float> t) {
        float x, y, z;
        thrust::tie(x, y, z) = t;
        return 2.0f * x + 3.0f * y + 4.0f * z;
    }
};

int main(void) {
    thrust::device_vector<float> X(3), Y(3), Z(3);
    thrust::device_vector<float> U(3);


    thrust::transform
        (thrust::make_zip_iterator(thrust::make_tuple(X.begin(), Y.begin(), Z.begin())),
         thrust::make_zip_iterator(thrust::make_tuple(X.end(), Y.end(), Z.end())),
         U.begin(),
         linear_combo());

    for (size_t i = 0; i < Z.size(); i++)
        std::cout << "U[" << i << "] = " << U[i] << "\n";
    return 0;
}
```

Functor Definition

These are the important parts: three different entities are zipped together in one big one.
Example: thrust::transform_reduce

```cpp
#include <thrust/transform_reduce.h>
#include <thrust/device_vector.h>
#include <thrust/iterator/zip_iterator.h>
#include <iostream>

struct linear_combo {
    __host__ __device__
    float operator() (thrust::tuple<float, float, float> t) {
        float x, y, z;
        thrust::tie(x, y, z) = t;
        return 2.0f * x + 3.0f * y + 4.0f * z;
    }
};

int main(void) {
    thrust::device_vector<float> X(3), Y(3), Z(3), U(3);


    thrust::plus<float> binary_op;
    float init = 0.f;

    float myResult = thrust::transform_reduce
        (thrust::make_zip_iterator(thrust::make_tuple(X.begin(), Y.begin(), Z.begin())),
         thrust::make_zip_iterator(thrust::make_tuple(X.end(), Y.end(), Z.end())),
         linear_combo(),
         init,
         binary_op);

    std::cout << myResult << std::endl;
    return 0;
}
```

Useful, Problem 4 Assignment 9
typedef thrust::tuple<int, int> Tuple;

struct max_index {
    __host__ __device__
    Tuple operator()(Tuple a, Tuple b) {
        if (thrust::get<0>(a) > thrust::get<0>(b))
            return a;
        else
            return b;
    }
};

int main(void) {
    thrust::device_vector<int> X(3), Y(3);
    X[0] = 10; X[1] = 30; X[2] = 20;  // values
    Y[0] = 0; Y[1] = 1; Y[2] = 2;  // indices
    Tuple init(X[0], Y[0]);
    Tuple result = thrust::reduce
        (thrust::make_zip_iterator(thrust::make_tuple(X.begin(), Y.begin())),
         thrust::make_zip_iterator(thrust::make_tuple(X.end(), Y.end())),
         init,
         max_index());

    int value, index; thrust::tie(value, index) = result;
    std::cout << "maximum value is " << value << " at index " << index << "\n";
    return 0;
}
Example: Processing Rainfall Data

Rain situation, end of first day, for a set of five observation stations. Results, summarized over a period of time, summarized in the table below.

| day  | 0 0 1 2 5 5 6 6 7 8 ... |
| site | 2 3 0 1 1 2 0 1 2 1 ... |
| measurement | 9 5 6 3 3 8 2 6 5 10 ... |

Remarks:
1) Time series sorted by day
2) Measurements of zero are excluded from the time series
Example: Processing Rainfall Data

- Given the data above, here’re some questions you might ask:
  - Total rainfall at a given site
  - Total rainfall between given days
  - Total rainfall on each day
  - Number of days with any rainfall
struct one_site_measurement
{
    int siteOfInterest;

    one_site_measurement(int site) : siteOfInterest(site) {}  

    __host__ __device__
    int operator()(thrust::tuple<int, int> t)
    {
        if (thrust::get<0>(t) == siteOfInterest)
            return thrust::get<1>(t);
        else
            return 0;
    }
};

template <typename Vector>
int compute_total_rainfall_at_one_site(int siteID, const Vector& site, const Vector& measurement) 
{
    return thrust::transform_reduce
        (thrust::make_zip_iterator(thrust::make_tuple(site.begin(), measurement.begin())),
         thrust::make_zip_iterator(thrust::make_tuple(site.end(), measurement.end())),
         one_site_measurement(siteID),
         0,
         thrust::plus<int>())
;
template<typename Vector>
int compute_total_rainfall_between_days(int first_day, int last_day,
const Vector& day, const Vector& measurement)
{
    int first = thrust::lower_bound(day.begin(), day.end(), first_day) - day.begin();
    int last = thrust::upper_bound(day.begin(), day.end(), last_day) - day.begin();
    return thrust::reduce(measurement.begin() + first, measurement.begin() + last);
}
Total Rainfall on Each Day

```cpp
template <typename Vector>
void compute_total_rainfall_per_day(const Vector& day, const Vector& measurement,
                                     Vector& day_output, Vector& measurement_output)
{
  size_t N = compute_number_of_days_with_rainfall(day); //see previous slide

day_output.resize(N);
measurement_output.resize(N);

thrust::reduce_by_key(day.begin(), day.end(),
                       measurement.begin(),
                       day_output.begin(),
                       measurement_output.begin());
}
```

<table>
<thead>
<tr>
<th>day</th>
<th>[0 0 1 2 5 5 6 6 7 8 ... ]</th>
</tr>
</thead>
<tbody>
<tr>
<td>measurement</td>
<td>[9 + 5 6 3 3 + 8 2 + 6 5 10 ... ]</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>day_output</th>
<th>[0 1 2 5 6 7 8 ... ]</th>
</tr>
</thead>
<tbody>
<tr>
<td>measurement_output</td>
<td>[14 6 3 11 8 5 10 ... ]</td>
</tr>
</tbody>
</table>
Number of Days with Any Rainfall

template <typename Vector>
int compute_number_of_days_with_rainfall(const Vector& day) {
    return thrust::inner_product(day.begin(), day.end() - 1,
                                  day.begin() + 1, 0,
                                  thrust::plus<int>(),
                                  thrust::not_equal_to<int>()) + 1;
}
Problem 3, Assignment 9

- Number of days where rainfall exceeded 5
  - Use `count_if` and a placeholder

- Total rainfall at each site
  - Use `sort_by_key` and `reduce_by_key`
thrust, Efficiency Issues

[fusing transformations]
typedef thrust::tuple<int, int> Tuple;

struct max_index {
    __host__ __device__
    Tuple operator()(Tuple a, Tuple b) {
        if (thrust::get<0>(a) > thrust::get<0>(b))
            return a;
        else
            return b;
    }
};

int main(void) {
    thrust::device_vector<int> X(3), Y(3);
    X[0] = 10; X[1] = 30; X[2] = 20; // values
    Y[0] = 0; Y[1] = 1; Y[2] = 2; // indices
    Tuple init(X[0], Y[0]);

    Tuple result = thrust::reduce
    (thrust::make_zip_iterator(thrust::make_tuple(X.begin(), Y.begin())),
     thrust::make_zip_iterator(thrust::make_tuple(X.end(), Y.end())),
     init,
     max_index());

    int value, index; thrust::tie(value, index) = result;

    std::cout << "maximum value is " << value << " at index " << index << "\n";
    return 0;
}
Performance Considerations
[short detour: 1/3]

1030 GFLOP/s [SP]

GPU

144 GB/s

DRAM

Tesla C2050
Arithmetic Intensity
[short detour: 2/3]

Memory bound

SAXPY

FFT

SGEMM

FLOP/Byte

Compute bound
## Arithmetic Intensity

*short detour: 3/3*

<table>
<thead>
<tr>
<th>Kernel</th>
<th>FLOP/Byte*</th>
</tr>
</thead>
<tbody>
<tr>
<td>Vector Addition</td>
<td>1 : 12</td>
</tr>
<tr>
<td>SAXPY</td>
<td>2 : 12</td>
</tr>
<tr>
<td>Ternary Transformation</td>
<td>5 : 20</td>
</tr>
<tr>
<td>Sum</td>
<td>1 : 4</td>
</tr>
<tr>
<td>Max Index</td>
<td>1 : 12</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Hardware**</th>
<th>FLOP/Byte</th>
</tr>
</thead>
<tbody>
<tr>
<td>GeForce GTX 280</td>
<td>~7.0 : 1</td>
</tr>
<tr>
<td>GeForce GTX 480</td>
<td>~7.6 : 1</td>
</tr>
<tr>
<td>Tesla C870</td>
<td>~6.7 : 1</td>
</tr>
<tr>
<td>Tesla C1060</td>
<td>~9.1 : 1</td>
</tr>
<tr>
<td>Tesla C2050</td>
<td>~7.1 : 1</td>
</tr>
</tbody>
</table>

* excludes indexing overhead

** lists the number of flop per byte of data to reach peak Flop/s rate

“Byte” refers to a Global Memory byte

NVIDIA [N. Bell]→
typedef thrust::tuple<int, int> Tuple;

struct max_index {
    __host__ __device__
    Tuple operator()(Tuple a, Tuple b) {
        if (thrust::get<0>(a) > thrust::get<0>(b))
            return a;
        else
            return b;
    }
};

int main(void) {
    thrust::device_vector<int>     X(3);
    thrust::counting_iterator<int> Y(0);
    Tuple init(X[0], Y[0]);
    Tuple result = thrust::reduce
        (thrust::make_zip_iterator(thrust::make_tuple(X.begin(), Y)),
        thrust::make_zip_iterator(thrust::make_tuple(X.end(), Y + X.size())),
        init,
        max_index());
    int value, index; thrust::tie(value, index) = result;
    std::cout << "maximum value is " << value << " at index " << index << "\n";
    return 0;
}
Maximum Index (Optimized)

Original Implementation

Optimized Implementation

NVIDIA [N. Bell]→
Fusing Transformations

for (int i = 0; i < N; i++)
    U[i] = F(X[i], Y[i], Z[i]);

for (int i = 0; i < N; i++)
    V[i] = G(X[i], Y[i], Z[i]);

Loop Fusion

● One way to look at things…
  ● Zipping: reorganizing data for thrust processing
  ● Fusing: reorganizing computation for efficient thrust processing
typedef thrust::tuple<float,float> Tuple2;
typedef thrust::tuple<float,float,float> Tuple3;

struct linear_combo {
    __host__ __device__
    Tuple2 operator()(Tuple3 t) {
        float x, y, z; thrust::tie(x,y,z) = t;

        float u = 2.0f * x + 3.0f * y + 4.0f * z;
        float v = 1.0f * x + 2.0f * y + 3.0f * z;

        return Tuple2(u,v);
    }
};

int main(void) {
    thrust::device_vector<float> X(3), Y(3), Z(3);
    thrust::device_vector<float> U(3), V(3);


    thrust::transform
        (thrust::make_zip_iterator(thrust::make_tuple(X.begin(), Y.begin(), Z.begin())),
         thrust::make_zip_iterator(thrust::make_tuple(X.end(), Y.end(), Z.end())),
         thrust::make_zip_iterator(thrust::make_tuple(U.begin(), V.begin())),
         linear_combo());

    return 0;
}
Fusing Transformations

Original Implementation

- GPU to DRAM: 12 Bytes
- DRAM to GPU: 4 Bytes

Optimized Implementation

- GPU to DRAM: 12 Bytes
- DRAM to GPU: 8 Bytes

- Since the operation is completely memory bound the expected speedup is ~1.6x (=32/20)
Fusing Transformations

for (int i = 0; i < N; i++)
    Y[i] = F(X[i]);

for (int i = 0; i < N; i++)
    sum += Y[i];

Loop Fusion

for (int i = 0; i < N; i++)
    sum += F(X[i]);
#include <thrust/device_vector.h>
#include <thrust/transform_reduce.h>
#include <thrust/functional.h>
#include <iostream>

using namespace thrust::placeholders;

int main(void) {
    thrust::device_vector<float> X(3);


    float result = thrust::transform_reduce
        (X.begin(), X.end(), X.end(),
        _1 * _1,
        0.0f,
        thrust::plus<float>()));

    std::cout << "sum of squares is " << result << "\n";
    return 0;
}
Fusing Transformations

Original Implementation

Optimized Implementation

NVIDIA [N. Bell]
Good Speedups Compared to Multi-threaded CPU Execution

- CUDA 4.1 on Tesla M2090, ECC on
- MKL 10.2.3, TYAN FT72-B7015 Xeon x5680 Six-Core @ 3.33 GHz
thrust Wrap-Up

- Significant boost in productivity at the price of small performance penalty
  - No need to know of execution configuration, shared memory, etc.

- Key concepts
  - Functor
  - Fusing operations
  - Zipping data
thrust on Google Code

- Quick Start Guide
- Examples
- News
- Documentation
- Mailing List (thrust-users)
thrust in “GPU Computing Gems”

PDF available at http://goo.gl/adj9S
3 Ways to Accelerate on GPU

Application

Libraries

Directives

Programming Languages

Easiest Approach → Maximum Performance

Direction of increased performance (and effort)

NVIDIA [C. Woolley]
Directives...
OpenACC

- Seeks to become:
  - A standard for directives-based Parallel Programming
  - Provide portability across hardware platforms and compiler vendors

- Promoted by NVIDIA, Cray, CAPS, PGI
OpenACC Specification

- Hardware agnostic and platform independent (CPU only, different GPUs)

- OpenACC is an open standard for directives based computing

- Announced at SC11 [November 2011]

- Caps, Cray, and PGI to ship OpenACC Compilers beginning Q1 2012

- Very early in the release cycle, you can only download and install a trial version
  - Right now it’s more of an vision…
The OpenACC Idea

- Host code computes an approximation for $\pi$:

```cpp
#include <iostream>
#include <math.h>
using namespace std;

int main( int argc, char *argv[] )
{
    const double PI25DT = 3.141592653589793238462643;
    const int n=1000000;
    double h   = 1.0 / (double) n;
    double sum = 0.0;

    for( int i=0; i<=n; i++ )
    {
        double x = h * ((double)i - 0.5);
        sum += (4.0 / (1.0 + x*x));
    }
    double mypi = h * sum;

    cout << "Approx. value: " << mypi << endl;
    cout << "Error: " << fabs(mypi-PI25DT) << endl;
    return 0;
}
```
The OpenACC Idea

- Code computes an approximation for π [might use multi-core or GPU]

```cpp
#include <iostream>
#include <math.h>
using namespace std;

int main( int argc, char *argv[] )
{
    const double PI25DT = 3.141592653589793238462643;

    const int n=1000000;
    double h   = 1.0 / (double) n;
    double sum = 0.0;
    // #pragma acc region for
    for( int i=0; i<=n; i++ ) {
        double x = h * ((double)i - 0.5);
        sum += (4.0 / (1.0 + x*x));
    }
    double mypi = h * sum;

    cout << "Approx. value: " << mypi << endl;
    cout << "Error: " << fabs(mypi-PI25DT) << endl;
    return 0;
}
```

Add one line of code (a directive): provides a hint to the compiler about opportunity for parallelism
OpenACC Target Audience

- OpenACC targets three classes of users:
  - Users with parallel codes, ideally with some OpenMP experience, but less GPU knowledge
  - Users with serial codes looking for portable parallel performance with and without GPUs
  - "Hardcore" GPU programmers with existing CUDA ports
OpenACC Perceived Benefits

- Code easier to maintain
- Helps with legacy code bases
- Portable:
  - Can run same code CPU/GPU
- Programmer familiar with OpenMP
- Some performance loss
  - Cray goal: 90% of CUDA
Libraries...
CUDA Libraries

- Math, Numerics, Statistics
- Dense & Sparse Linear Algebra
- Algorithms (sort, etc.)
- Image Processing
- Signal Processing
- Finance

- In addition to these well celebrated libraries, several less established ones available in the community

cuBLAS: Dense linear algebra on GPUs

- Complete BLAS implementation plus useful extensions
  - Supports all 152 standard routines for single, double, complex, and double complex
  - Levels 1, 2, and 3 BLAS

- New features in CUDA 4.1:
  - New batched GEMM API provides >4x speedup over MKL
  - Useful for batches of 100+ small matrices from 4x4 to 128x128
  - 5%-10% performance improvement to large GEMMs
Speedups Compared to Multi-threaded CPU Execution

- CUDA 4.1 on Tesla M2090, ECC on
- MKL 10.2.3, TYAN FT72-B7015 Xeon x5680 Six-Core @ 3.33 GHz

**cuBLAS**

- SGEMM
- SSYMM
- SSYRK
- STRMM
- STRSM
cuSPARSE: Sparse linear algebra routines

- Sparse matrix-vector multiplication & triangular solve
  - APIs optimized for iterative methods

- New features in 4.1:
  - Tri-diagonal solver with speedups up to 10x over Intel MKL
  - ELL-HYB format offers 2x faster matrix-vector multiplication

\[
\begin{bmatrix}
y_1 \\
y_2 \\
y_3 \\
y_4 \\
\end{bmatrix} = \alpha \begin{bmatrix} 2 & -1 \\ 4 & -1 \\ 5 & 9 & 1 \\ -1 & 8 \\
\end{bmatrix} \begin{bmatrix} 2 \\ -1 \\ 2 \\ 3 \\
\end{bmatrix} + \beta \begin{bmatrix} 0 \\ 0 \\ 1 \\ -1 \\
\end{bmatrix}
\]
Good Speedups Compared to Multi-threaded CPU Execution

Sparse matrix test cases on following slides come from:
1. The University of Florida Sparse Matrix Collection
   http://www.cise.ufl.edu/research/sparse/matrices/
   http://www.nvidia.com/object/nvidia_research_pub_001.html

- CUDA 4.1 on Tesla M2090, ECC on
- MKL 10.2.3, TYAN FT72-B7015 Xeon x5680 Six-Core @ 3.33 GHz
cuFFT: Multi-dimensional FFTs

- Algorithms based on Cooley-Tukey and Bluestein
- Simple interface, similar to FFTW
- Streamed asynchronous execution
- 1D, 2D, 3D transforms of complex and real data
- Double precision (DP) transforms
- 1D transform sizes up to 128 million elements
- Batch execution for doing multiple transforms
- In-place and out-of-place transforms

\[
F(x) = \sum_{n=0}^{N-1} f(n)e^{-j2\pi(x_n^n)}
\]

\[
f(n) = \frac{1}{N} \sum_{n=0}^{N-1} F(x)e^{j2\pi(x_n^n)}
\]
Speedups Compared to Multi-Threaded CPU Execution

- CUDA 4.1 on Tesla M2090, ECC on
- MKL 10.2.3, TYAN FT72-B7015 Xeon x5680 Six-Core @ 3.33 GHz
cuRAND: Random Number Generation

- Pseudo- and Quasi-RNGs
  - Supports several output distributions
  - Statistical test results reported in documentation

- New RNGs in CUDA 4.1:
  - MRG32k3a RNG
  - MTGP11213 Mersenne Twister RNG
NPP: NVIDIA Performance Primitives

- Arithmetic, Logic, Conversions, Filters, Statistics, Signal Processing, etc.
- This is where GPU computing shines
- 1,000+ new image primitives in 4.1
CUDA Progress on Library Development

- **2007**: CUDA Toolkit 1.x
  - Single precision
  - cuBLAS
  - cuFFT
  - math.h

- **2008**: CUDA Toolkit 2.x
  - Double Precision support in all libraries

- **2009**: CUDA Toolkit 3.x
  - cuSPARSE
  - cuRAND
  - printf()
  - malloc()

- **2010**: CUDA Toolkit 4.x
  - Thrust
  - NPP
  - assert()
Development, Debugging, and Deployment Tools

[Rounding Up the CUDA Ecosystem]
Programming Languages & APIs

- HMPP Compiler
- Python for CUDA
- NVIDIA C Compiler
- CUDA Fortran
- NVIDIA C Compiler
- OpenCL
- Microsoft DirectX 11
- NVIDIA C Compiler
- PGI Accelerator
- Microsoft AMP C/C++
Debugging Tools

- NVIDIA Parallel Nsight for Visual Studio
- NVIDIA CUDA-MEMCHECK for Linux & Mac
- Allinea DDT with CUDA Distributed Debugging Tool
- NVIDIA CUDA-GDB for Linux & Mac
- TotalView for CUDA for Linux Clusters
Performance Analysis Tools

- NVIDIA Parallel Nsight for Visual Studio
- Vampir Trace Collector
- TAU Performance System
- Performance API Library
- NVIDIA Visual Profiler for Linux & Mac
- Under Development
MPI & CUDA Support

GPUDirect™ Peer-Peer Transfers

InfiniBand

OpenFabrics Alliance
As of OFED 1.5.2

MVAPICH
Announced pre-release at SC2011

Platform Computing
Platform MPI
Announced beta at SC2011
Cluster Management & Job Scheduling

- **Platform Computing**: LSF, HPC, Cluster Manager
- **Bright Computing**: Bright Cluster Manager
- **Adaptive Computing**: NVML Plugin for GPUs
- **ROCKS+ MOAB**: Univa Grid Engine
- **PBS Works**: PBS Professional
- **Ganglia**: NVML Plugin for GPUs
- **UNIVA**: Univa Grid Engine
CUDA: Getting More Info…

● More information on this

● CUDA Tools and Ecosystem
  ● Described in detail on NVIDIA Developer Zone
    http://developer.nvidia.com/category/zone/cuda-zone
First question you need to ask: is there a GPU library that I can use?

In your GPU implementation the code is likely going to be memory bound

- Move data to GPU and keep it here
- Understand the GPU memory ecosystem and the costs associated with accessing various memory spaces
- Algorithms that have higher arithmetic intensity will fare well

JUST DO IT!

- Avoid “analysis paralysis”
- Go back and profile/optimize once you have something working
- To “have something working” debug like a pro (cuda-gdb and cuda-memchk)
Parallel programming skills are in demand...

- Received this yesterday:
  - “NVIDIA is looking for smart, passionate software engineers for its high-performance computing team. Relevant domains include geosciences, life sciences, computational fluid dynamics, computational chemistry, computational physics, computational finance, data mining, medical imaging and more. Learn more: http://bit.ly/w00tMm”
End GPU Segment of ME964