ME759
High Performance Computing for Engineering Applications

GPU Computing with thrust, wrap up
The CUDA IDE & library ecosystem
October 21, 2013

“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, Streams in CUDA
  - GPU computing w/ **thrust**
    - New concept: the *functor* as a provider of the “call” operator to customize **thrust** behavior

- **Today:**
  - Wrap up GPU computing w/ **thrust**
  - Wrap up GPU computing discussion

- **Miscellaneous**
  - HW due today at 11:59 pm
  - New HW posted online later today. Due on Oct. 28 at 11:59 PM
  - Due date for midterm project topic is Oct 23, 11:59 PM (upload in Learn@UW)
  - Exam moved back from November 8 to November 25 at 7:15 PM (Room TBA)
    - Review session held during regular class hour (show up only if you think it’s useful)
Looking Ahead

[Forum Post]

- **Oct. 23, 11:59 PM (Learn@UW submission)**
  - Proposal for Midterm Project is due. Default Midterm Project available if undecided.
  - One page of text (doesn’t include title page and references, if any). If going with default, submit one line saying so.

- **Nov. 15, 11:59 PM (Learn@UW submission)**
  - Midterm Project due
  - Seven pages of narrative at the most (this includes pictures but doesn’t include title page and references, if any)

- **Nov. 15, 11:59 PM (Learn@UW submission)**
  - Proposal for Final Project is due
  - Two pages of text max (this includes pictures but doesn’t include title page and references, if any)

- **Nov. 25, 7:15 – 9:15 PM – midterm exam (Room TBA)**

- **Dec. 15, 11:59 PM (Learn@UW submission)**
  - Final Project due
  - Ten pages of narrative at the most (this includes pictures but doesn’t include title page and references, if any)

- Use these templates for all documents you submit (stick with this formatting, font size, etc.):
  - LaTeX: [http://sbel.wisc.edu/documents/Latextemplate.zip](http://sbel.wisc.edu/documents/Latextemplate.zip)
Algorithms

● Elementwise operations
  ● for_each, transform, gather, scatter ...

● Reductions
  ● reduce, inner_product, reduce_by_key ...

● Prefix Sums [scans]
  ● inclusive_scan, inclusive_scan_by_key ...

● Sorting
  ● sort, stable_sort, sort_by_key ...
<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>
Thrust Example: Sort

```cpp
#include <thrust/host_vector.h>
#include <thrust/device_vector.h>
#include <thrust/sort.h>

int main(void) {
    // generate 16M random numbers on the host
    thrust::host_vector<int> h_vec(1 << 24);
    thrust::generate(h_vec.begin(), h_vec.end(), rand);

    // transfer data to the device
    thrust::device_vector<int> d_vec = h_vec;

    // sort data on the device (805 Mkeys/sec on GeForce GTX 480)
    thrust::sort(d_vec.begin(), d_vec.end());

    // transfer data back to host
    thrust::copy(d_vec.begin(), d_vec.end(), h_vec.begin());

    return 0;
}
```
Leveraging Parallel Primitives

- 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

![Graph showing sorting rate (Mkey/s) vs. key bits]

Sorting Rate (Mkey/s)

Key Bits

NVIDIA [N. Bell]→
#include <thrust/device_vector.h>
#include <thrust/reduce.h>
#include <thrust/functional.h>
#include <iostream>

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


    float init = 0.0f;

    float result = thrust::reduce(X.begin(), X.end(),
        init,
        thrust::maximum<float>())

    std::cout << "maximum is " << result << "\n";

    return 0;
}
Algorithms

- Process one or more ranges

```cpp
// copy values to device
device_vector<int> A(10);
device_vector<int> B(10);
device_vector<int> C(10);

// sort A in-place
sort(A.begin(), A.end());

// copy A -> B
copy(A.begin(), A.end(), B.begin());

// transform A + B -> C
transform(A.begin(), A.end(), B.begin(), C.begin(), plus<int>());
```
Algorithms

- Standard operators

```cpp
// allocate memory
device_vector<int> A(10);
device_vector<int> B(10);
device_vector<int> C(10);

// transform A + B -> C
transform(A.begin(), A.end(), B.begin(), C.begin(), plus<int>());

// transform A - B -> C
transform(A.begin(), A.end(), B.begin(), C.begin(), minus<int>());

// multiply reduction
int product = reduce(A.begin(), A.end(), 1, multiplies<int>());
```
Algorithms

- Standard data types

```cpp
// allocate device memory
device_vector<int> i_vec = ...;
device_vector<float> f_vec = ...;

// sum of integers
int i_sum = reduce(i_vec.begin(), i_vec.end());

// sum of floats
float f_sum = reduce(f_vec.begin(), f_vec.end());
```
struct negate_float2
{
    __host__ __device__
    float2 operator()(float2 a)
    {
        return make_float2(-a.x, -a.y);
    }
};

// declare storage
device_vector<float2> input = ... 
device_vector<float2> output = ...

// create function object or ‘functor’
negate_float2 func;

// negate vectors
transform(input.begin(), input.end(), output.begin(), func);
Custom Types & Operators

// compare x component of two float2 structures
struct compare_float2
{
    __host__ __device__
    bool operator()(float2 a, float2 b)
    {
        return a.x < b.x;
    }
};

// declare storage
device_vector<float2> vec = ...

// create comparison functor
compare_float2 comp;

// sort elements by x component
sort(vec.begin(), vec.end(), comp);
// return true if x is greater than threshold
struct is_greater_than
{
    int threshold;

    is_greater_than(int t) { threshold = t; }

    __host__ __device__
    bool operator()(int x) { return x > threshold; }
};

device_vector<int> vec = ...

// create predicate functor (returns true for x > 10)
is_greater_than pred(10);

// count number of values > 10
int result = count_if(vec.begin(), vec.end(), pred);
Interoperability

- Convert iterators to raw pointers

```cpp
// allocate device vector
thrust::device_vector<int> d_vec(4);

// obtain raw pointer to device vector's memory
int * ptr = thrust::raw_pointer_cast(&d_vec[0]);

// use ptr in a CUDA C kernel
my_kernel<<< (N+255) / 256, 256 >>>(N, ptr);

// use ptr in a CUDA API function
cudaMemcpyAsync(ptr, ...);
```
Interoperability

- Wrap raw pointers with `device_ptr`

```c
// raw pointer to device memory
int * raw_ptr;
cudaMalloc((void **) &raw_ptr, N * sizeof(int));

// wrap raw pointer with a device_ptr
thrust::device_ptr<int> dev_ptr(raw_ptr);

// use device_ptr in thrust algorithms
thrust::fill(dev_ptr, dev_ptr + N, (int) 0);

// access device memory through device_ptr
dev_ptr[0] = 1;

// free memory
cudaFree(raw_ptr);
```
General Transformations

- **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

A B C
X Y Z

zip_iterator

NVIDIA [N. Bell]→
```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;
}
```

Example: General Transformations

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;
}  
```
thrust, Efficiency Issues
[fusing transformations]
Performance Considerations

[short detour: 1/3]

- Picture below shows key parameters
  - Peak flop rate
  - Max bandwidth

![Diagram showing Tesla C2050 with SMs and DRAM, peak flop rate of 1030 GFLOP/s (SinglePrecision) and max bandwidth of 144 GB/s.](image)
Arithmetic Intensity
[short detour: 2/3]
# Arithmetic Intensity

[short detour: 3/3]

## Kernel FLOP/Byte*

<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>

* excludes indexing overhead

## Hardware**

<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>

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

“Byte” refers to a Global Memory byte
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 → DRAM
  - 12 Bytes
  - 4 Bytes

Optimized Implementation

- GPU → DRAM
  - 12 Bytes
  - 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];
```

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

Loop Fusion
Fusing Transformations

```cpp
#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(),
        _1 * _1,
        0.0f,
        thrust::plus<float>())
        std::cout << "sum of squares is " << result << "\n";
    return 0;
}
```
Fusing Transformations

Original Implementation

Optimized Implementation

Try to answer this: how many times will we be able to run faster if we fuse?
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;
}
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

- Try to answer this: how many times will we be able to run faster if we fuse?
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 chart showing speedups compared to CPU execution for reduce, transform, scan, sort operations.](image)
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)

http://code.google.com/p/thrust/
thrust in “GPU Computing Gems”
Example, **thrust**: Processing Rainfall Data

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

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

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
Total Rainfall at a Given Site

```cpp
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);
}

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

For this to fly, you’ll need to include several header files (not all for the code snippet above)
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;
}
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());
}
3 Ways to Accelerate on GPU

- Libraries
- Directives
- Programming Languages

Easiest Approach → Maximum Performance

Direction of increased performance (and effort)
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 \( \pi \):

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

int main( int argc, char *argv[] )
{
    const double PI25DT = 3.141592653589793;
    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 $\pi$ [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
- Very much like OpenMP
- Only small performance loss
  - Cray goal: 90% of CUDA

NVIDIA [C. Woolley]→
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”
- Adopt a “crawl – walk – run” approach
- Go back and profile/optimize once you have something working
- To “have something working” debug like a pro (cuda-gdb and cuda-memchk)
Libraries...
CUDA Libraries

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

- In addition to these widely adopted 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
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

\[
\begin{bmatrix}
  y_1 \\
  y_2 \\
  y_3 \\
  y_4 \\
\end{bmatrix} = \alpha \begin{bmatrix}
  2 & -1 \\
  4 & -1 & 2 \\
  5 & 9 & 1 & 1 \\
  -1 & 8 & 3 & -1 \\
\end{bmatrix} + \beta \begin{bmatrix}
  2 \\
  0 \\
  -1 \\
  2 \\
\end{bmatrix}
\]
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
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
Development, Debugging, and Deployment Tools
[Rounding Up the CUDA Ecosystem]
Programming Languages & APIs

- HMPP Compiler
- Python for CUDA
- NVIDIA C Compiler
- CUDA Fortran
- OpenCL
- NVIDIA CUDA
- OpenGL
- PGI Accelerator
- NVIDIA DirectX
- 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

<table>
<thead>
<tr>
<th>Tool Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>NVIDIA Parallel Nsight</td>
<td>for Visual Studio</td>
</tr>
<tr>
<td>Vampir Trace Collector</td>
<td></td>
</tr>
<tr>
<td>Tuning and Analysis Utilities</td>
<td>TAU Performance System</td>
</tr>
<tr>
<td>Performance API Library</td>
<td></td>
</tr>
<tr>
<td>NVIDIA Visual Profiler</td>
<td>for Linux &amp; Mac</td>
</tr>
<tr>
<td>HPCToolkit</td>
<td>Under Development</td>
</tr>
</tbody>
</table>
MPI & CUDA Support

Platform MPI
Announced beta at SC2011

GPUDirect™

InfiniBand

Peer-Peer Transfers

CUDA Memcpy()
Cluster Management & Job Scheduling

- LSF, HPC, Cluster Manager
- Bright Cluster Manager
- PBS Professional
- NVML Plugin for GPUs
- Univa Grid Engine