GPU Computing with CUDA

Dan Negrut
Andrew Seidl
Daniel Melanz

Simulation-Based Engineering Lab
Department of Mechanical Engineering
University of Wisconsin-Madison

© Dan Negrut, 2012
UW-Madison

Bucuresti
6-8 Iunie 2012
Before we get started...

- **Yesterday: CUDA basics**
  - Get familiar with the memory hierarchy on NVIDIA’s GPUs
  - Understand the scheduling of threads for execution on an SM
  - Issues related to writing effective CUDA code
    - Warp divergence, use of shared memory, etc.
  - Conclude with another hands-on session: focus on use of shared memory

- **Today: CUDA productivity tools**
  - GPU computing with **thrust**
  - The CUDA library ecosystem
  - Profiling with **nvvp**
  - Debugging with **cuda-gdb**
GPU Computing using thrust
3 Ways to Accelerate on GPU

Application

Libraries
Directives
Programming Languages

Easiest Approach → Maximum Performance

Direction of increased performance (and effort)
Acknowledgments

- The **thrust** slides include material provided by Nathan Bell of NVIDIA
- Any mistakes in these slides belong to me
Design Philosophy, **thrust**

- Increase programmer productivity
  - Build complex applications quickly

- Encourage generic programming
  - Leverage parallel primitives

- Should run fast
  - Efficient mapping to hardware
What is **thrust**?

- A template library for CUDA
  - Mimics the C++ STL

- Containers
  - On host and device

- Algorithms
  - Sorting, reduction, scan, etc.
What is **thrust**?

[Cntd.]

- **thrust** is a header library – all the functionality is accessed by `#include`-ing the appropriate `thrust` header file

- Program is compiled with **nvcc** as per usual, no special tools are required

- Lots of C++ syntax, related to high-level host-side code that you write
  - The concept of execution configuration, shared memory, etc. : all gone
Why Should One Use thrust?

- Extensively tested
  - 600+ unit tests

- Open Source
  - Permissive License (Apache v2)

- Active community
Example: Vector Addition

```c
for (int i = 0; i < N; i++)
    Z[i] = X[i] + Y[i];
```
#include <thrust/device_vector.h>
#include <thrust/transform.h>
#include <thrust/functional.h>
#include <iostream>

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


    thrust::transform(X.begin(), X.end(),
                      Y.begin(),
                      Z.begin(),
                      thrust::plus<float>());

    for (size_t i = 0; i < Z.size(); i++)
        std::cout << "Z[" << i << "] = " << Z[i] << 

    return 0;
}
Example, Vector Addition

[negrut@euler01 CodeBits]$ nvcc --version
nvcc: NVIDIA (R) Cuda compiler driver
Copyright (c) 2005-2011 NVIDIA Corporation
Built on Thu_Jan_12_14:41:45_PST_2012
Cuda compilation tools, release 4.1, V0.2.1221
[negrut@euler01 CodeBits]$ nvcc -O2 exThrust.cu -o exThrust.exe
[negrut@euler01 CodeBits]$ ./exThrust.exe
Z[0] = 25
Z[1] = 55
Z[2] = 40
[negrut@euler01 CodeBits]$ 

- Note: file extension should be .cu
Example: SAXPY

```c
for (int i = 0; i < N; i++)
  Z[i] = a * X[i] + Y[i];
```

![Diagram of SAXPY operation]
struct saxpy
{
    float a;

    saxpy(float a) : a(a) {}  

    __host__ __device__
    float operator()(float x, float y)
    {
        return a * x + y;
    }
};

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


    float a = 2.0f;

    thrust::transform(X.begin(), X.end(),
                      Y.begin(),
                      Z.begin(),
                      saxpy(a));

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

    return 0;
}
#include <thrust/host_vector.h>
#include <thrust/device_vector.h>
#include <thrust/sort.h>

int main(void) {
    // generate 32M random numbers on the host
    thrust::host_vector<int> h_vec(32 << 20);
    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 (846M keys per 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;
}
Containers

- Concise and readable code
  - Avoids common memory management errors
    - e.g.: Vectors automatically release memory when they go out of scope

```cpp
// allocate host vector with two elements
thrust::host_vector<int> h_vec(2);

// copy host vector to device
thrust::device_vector<int> d_vec = h_vec;

// write device values from the host
d_vec[0] = 13;
d_vec[1] = 27;

// read device values from the host
std::cout << "sum: " << d_vec[0] + d_vec[1] << std::endl;
```
Containers

- Compatible with STL containers

```cpp
// list container on host
std::list<int> h_list;
h_list.push_back(13);
h_list.push_back(27);

// copy list to device vector
thrust::device_vector<int> d_vec(h_list.size());
thrust::copy(h_list.begin(), h_list.end(), d_vec.begin());

// alternative method using vector constructor
thrust::device_vector<int> d_vec(h_list.begin(), h_list.end());
```
Namespaces

- Avoid name collisions

```cpp
// allocate host memory
thrust::host_vector<int> h_vec(10);

// call STL sort
std::sort(h_vec.begin(), h_vec.end());

// call Thrust sort
thrust::sort(h_vec.begin(), h_vec.end());

// for brevity
using namespace thrust;

// without namespace
int sum = reduce(h_vec.begin(), h_vec.end());
```
Iterators

A pair of iterators defines a “range”

```cpp
// allocate device memory
device_vector<int> d_vec(10);

// declare iterator variables
device_vector<int>::iterator begin  = d_vec.begin();
device_vector<int>::iterator end    = d_vec.end();
device_vector<int>::iterator middle = begin + d_vec.size()/2;
// sum first and second halves
int sum_half1 = reduce(begin, middle);
int sum_half2 = reduce(middle, end);

// empty range
int empty = reduce(begin, begin);
```
Iterators

- Iterators act like pointers

```cpp
// declare iterator variables
device_vector<int>::iterator begin = d_vec.begin();
device_vector<int>::iterator end = d_vec.end();

// pointer arithmetic
begin++;

// dereference device iterators from the host
int a = *begin;
int b = begin[3];

// compute size of range [begin,end)
int size = end - begin;
```
Iterators

- Encode memory location
  - Automatic algorithm selection

```cpp
// initialize random values on host
host_vector<int> h_vec(100);
thrust::generate(h_vec.begin(), h_vec.end(), rand);

// copy values to device
device_vector<int> d_vec = h_vec;

// compute sum on host
int h_sum = thrust::reduce(h_vec.begin(), h_vec.end());

// compute sum on device
int d_sum = thrust::reduce(d_vec.begin(), d_vec.end());
```
Algorithms

- **Elementwise operations**
  - `for_each, transform, gather, scatter` ...

- **Reductions**
  - `reduce, inner_product, reduce_by_key` ...

- **Prefix-Sums**
  - `inclusive_scan, inclusive_scan_by_key` ...

- **Sorting**
  - `sort, stable_sort, sort_by_key` ...
Thrust Example: Sort

```c++
#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;
}
```
```cpp
#include <thrust/device_vector.h>
#include <thrust/reduce.h>
#include <thrust/functional.h>
#include <iostream>

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

    X[0] = 10.0f; X[1] = 30.0f; X[2] = 20.0f;

    float init = 0.0f;

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

```c
// 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);
// 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 / 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);
```
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)

NVIDIA [N. Bell]→
## 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

Unary Transformation

\[ \text{for } (\text{int } i = 0; i < N; i++) \]
\[ X[i] = f(A[i]); \]

Binary Transformation

\[ \text{for } (\text{int } i = 0; i < N; i++) \]
\[ X[i] = f(A[i], B[i]); \]

Ternary Transformation

\[ \text{for } (\text{int } i = 0; i < N; i++) \]
\[ X[i] = f(A[i], B[i], C[i]); \]

General Transformation

\[ \text{for } (\text{int } i = 0; i < N; i++) \]
\[ X[i] = f(A[i], B[i], C[i], ...); \]

- Like the STL, \texttt{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
#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: 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;
}
```
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.

<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

| day  | [0 0 1 2 5 5 6 6 7 8 ... ] |
| site | [2 3 0 1 1 2 0 1 2 1 ... ] |
| m.   | [9 5 6 3 3 8 2 6 5 10 ... ] |

- 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>())
};
```
Total Rainfall Between Given Days

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)
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;
}
```
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());
}

| day      | [0 0 1 2 5 5 6 6 7 8 ...] |
| measurement | [9 + 5 6 3 3 + 8 2 + 6 5 10 ...] |

| day_output | [0 1 2 5 6 7 8 ...] |
| measurement_output | [14 6 3 11 8 5 10 ...] |

NVIDIA [N. Bell]→
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]
Arithmetic Intensity
[short detour: 2/3]

- Memory bound
- Compute bound

FLOP/Byte

SAXPY

FFT

SGEMM

NVIDIA [N. Bell]→
### 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
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

GPU

DRAM

4 Bytes

8 Bytes

4 Bytes

GPU

DRAM

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
  - 12 Bytes
  - 4 Bytes

- DRAM
  - 12 Bytes
  - 4 Bytes

Optimized Implementation

- GPU
  - 12 Bytes

- DRAM
  - 8 Bytes

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

\[
\text{for (int } i = 0; i < N; i++) \quad Y[i] = F(X[i]);
\]

\[
\text{for (int } i = 0; i < N; i++) \quad \text{sum += } Y[i];
\]

Loop Fusion

\[
\text{for (int } i = 0; i < N; i++) \quad \text{sum += } F(X[i]);
\]
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

GPU DRAM

4 Bytes

4 Bytes

4 Bytes

4 Bytes

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

NVIDIA [N. Bell]→
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”

This chapter demonstrates how to leverage the Thrust parallel template library to implement high-performance applications with minimal programming effort. Thrust brings a familiar high-level interface to the realm of GPU Computing while remaining fully interoperable with the rest of the CUDA software ecosystem. Applications written with Thrust are concise, readable, and efficient.

26.1 MOTIVATION

With the introduction of CUDA C++, developers can harness the massive parallelism of the GPU through a standard programming language. CUDA allows developers to make fine-grained decisions about how computations are decomposed into parallel threads and executed on the device. The level of control offered by CUDA C++ (henceforth CUDA C) is an important feature; it facilitates the development of high-performance algorithms for a variety of computationally demanding tasks which (1) merit significant optimization and (2) profit from low-level control of the mapping onto hardware. For this class of computational tasks CUDA C is an excellent solution.

Thrust (1) solves a complementary set of problems, namely those that are (1) implemented efficiently without a detailed mapping of work onto the target architecture or those that (2) do not merit or simply will not receive significant optimization effort by the user. With Thrust, developers describe their computation using a collection of high-level algorithms and completely delegate the decision of how to implement the computation to the library. This abstract interface allows programmers to describe what to compute without placing any additional restrictions on how to carry out the computation. By constraining the programmer’s intent at a high level, Thrust has the discretion to make informed
3 Ways to Accelerate on GPU

Application

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 = 3.141592653589793238462643;
    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 \( \pi \) [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
- Programmer familiar with OpenMP
- Some performance loss
  - Cray goal: 90% of CUDA

The OpenACC™ API QUICK REFERENCE GUIDE

The OpenACC Application Program Interface describes a collection of compiler directives to specify loops and regions of code in standard C, C++ and Fortran to be offloaded from a host CPU to an attached accelerator, providing portability across operating systems, host CPUs and accelerators.

Most OpenACC directives apply to the immediately following structured block or loop; a structured block is a single statement or a compound statement (C or C++) or a sequence of statements (Fortran) with a single entry point at the top and a single exit at the bottom.
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

NVIDIA []→
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

\[
\begin{bmatrix}
  y_1 \\
  y_2 \\
  y_3 \\
  y_4
\end{bmatrix} = \alpha \begin{bmatrix}
  2 & -1 \\
  4 & -1 \\
  5 & 9 & 1 \\
  -1 & 8 & 3
\end{bmatrix} \begin{bmatrix}
  -1 \\
  2 \\
  1 \\
  3
\end{bmatrix} + \beta \begin{bmatrix}
  2 \\
  0 \\
  -1 \\
  2
\end{bmatrix}
\]
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

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

\[ f(n) = \frac{1}{N} \sum_{n=0}^{N-1} F(x)e^{j2\pi(x\frac{n}{N})} \]
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
- NVIDIA C Compiler
- NVIDIA Cuda
- CUDA Fortran
- OpenCL
- OpenGL
- Microsoft DirectX 11
- 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
- HPCToolkit (Under Development)
MPI & CUDA Support

Announced beta at SC2011
Announced pre-release at SC2011
As of OFED 1.5.2
Announced beta at SC2011

NVIDIA [C. Woolley]→
Cluster Management & Job Scheduling

LSF, HPC, Cluster Manager
Bright Cluster Manager
PBS Professional
NVML Plugin for GPUs
Univa Grid Engine
Execution Profiling
Premature Optimization is the Root of All Evil. Yet,…

“Programmers waste enormous amounts of time thinking about, or worrying about, the speed of noncritical parts of their programs, and these attempts at efficiency actually have a strong negative impact when debugging and maintenance are considered. We should forget about small efficiencies, say about 97% of the time: premature optimization is the root of all evil. Yet we should not pass up our opportunities in that critical 3%.”

Donald Knuth
In “Structured Programming With Go To Statements”
Computing Surveys, Vol. 6, No. 4, December 1974
Available on class website.
Regarding Code Optimization…

- In 99% of the cases, “Code Optimization” is not about writing 40 lines of convoluted code to save two additions and one multiplication at the price of no able human being being able understand what you did there [see, I just tried to optimize this statement]

- For all purposes, especially when it comes to GPU computing, you can basically forget about the math overhead (+, -, *, /, reciprocal, square root, sin, cos, etc.)
  - Pretend they don’t exist

- Focus on the operands: what you are left with once you get rid of the math

- Example:
  - Suppose you have this:
    
    ```
    >> c[i] = mypi*sin(a[i])+2.f;
    ```
  - Then, concentrate on this:
    - Where are c[i], mypi, a[i] coming from?
    - Maybe also, “Can I use Single Precision or should I use Double Precision?”
Regarding Code Optimization…

[Cntd.]

- Why you shouldn’t probably worry about math
  - One global memory transaction requires 400-600 cycles
  - Math operations (1.3 architecture, C1060):
    - 4 clock cycles for an integer or single-precision floating-point arithmetic instruction
    - 16 clock cycles for a single-precision floating-point transcendental instruction
    - 2.0 architecture is even better
GPU Computing: Putting Things in Perspective

- Example: 1 Tflops GPU needs a lot of data to reach peak rate
  - Assume that you want to add different numbers and reach 1 Tflops: 1E12 ops/second
  - You need to feed 2E12 operands per second...
  - If each number is stored using 4 bytes (float), then you need to fetch 2E12*4 bytes in a second. This is 8E12 B/s, which is 8 TB/s...
  - The memory bandwidth on the GPU is in the neighborhood of 0.15 TB/s, about 50 times less than what you need (and you haven’t taken into account that you probably want to send back the outcome of the operation that you carry out)
Another example: quick back-of-the-envelope computation to illustrate the crunching number power of a modern GPU

- Suppose it takes 4 microseconds (4E-6) to launch a kernel
- Suppose you use a 1 Tflops (1E12) Fermi-type GPU to add (in 4 cycles) floats
- Then, to break even with the amount of time it took you to invoke execution on the GPU in the first place you’d have to carry out about 1 million floating point ops on the GPU
  - [if everything was in registers and basically the only thing you did was crunch numbers]
Important Point

- In GPU computing, memory transactions are perhaps most relevant in determining the overall efficiency (performance) of your code.

<table>
<thead>
<tr>
<th>Memory Space</th>
<th>Bandwidth</th>
</tr>
</thead>
<tbody>
<tr>
<td>Register memory</td>
<td>≈ 8,000 GB/s</td>
</tr>
<tr>
<td>Shared memory</td>
<td>≈ 1,600 GB/s</td>
</tr>
<tr>
<td>Global memory</td>
<td>≈ 177 GB/s</td>
</tr>
<tr>
<td>Mapped memory</td>
<td>≈ 8 GB/s</td>
</tr>
</tbody>
</table>

Source: Rob Farber
“CUDA Application Design and Development”
Next, the discussion focuses on tools you can use to find that 3% of the code worth optimizing…
Code Timing/Profiling

- The expeditious solution
  - Do nothing, instruct the executable to register limited profiling info

- Advanced approach: use NVIDIA’s **nvvp** Visual Profiler
  - Visualize CPU and GPU activity
  - Identify optimization opportunities
  - Allows for automated analysis
  - **nvvp** is a cross platform tool (linux, mac, windows)
1D Stencil: A Common Algorithmic Pattern
[Problem Used to Introduce Profiling]

- Applying a 1D stencil to a 1D array of elements
  - Function of input elements within a radius

- Fundamental to many algorithms
  - Standard discretization methods, interpolation, convolution, filtering, …

- This example will use weighted arithmetic mean
Serial Algorithm

(radius = 3)

=NVIDIA [S. Satoor]→

= CPU Thread
Serial Algorithm

```
in  ...  out  ...
```

(radius = 3)

```
\( \Rightarrow = CPU \ Thread \)
```

```
Repeat for each element
```

NVIDIA [S. Satoor]→
int main() {
    int size = N * sizeof(float);
    int wsize = (2 * RADIUS + 1) * sizeof(float);
    //allocate resources
    float *weights = (float *)malloc(wsize);
    float *in = (float *)malloc(size);
    float*out= (float *)malloc(size);
    initializeWeights(weights, RADIUS);
    initializeArray(in, N);

    applyStencil1D(RADIUS,N-RADIUS,weights,in,out);

    //free resources
    free(weights); free(in); free(out);
}

void applyStencil1D(int sIdx, int eIdx, const float *weights, float *in, float *out) {
    for (int i = sIdx; i < eIdx; i++) {
        out[i] = 0;
        //loop over all elements in the stencil
        for (int j = -RADIUS; j <= RADIUS; j++) {
            out[i] += weights[j + RADIUS] * in[i + j];
        }
        out[i] = out[i] / (2 * RADIUS + 1);
    }
}
int main() {
    int size = N * sizeof(float);
    int wsize = (2 * RADIUS + 1) * sizeof(float);
    //allocate resources
    float *weights = (float *)malloc(wsize);
    float *in = (float *)malloc(size);
    float *out = (float *)malloc(size);
    initializeWeights(weights, RADIUS);
    initializeArray(in, N);
    applyStencil1D(RADIUS, N-RADIUS, weights, in, out);

    //free resources
    free(weights); free(in); free(out);
}

void applyStencil1D(int sIdx, int eIdx, const float *weights, float *in, float *out) {
    for (int i = sIdx; i < eIdx; i++) {
        out[i] = 0;
        //loop over all elements in the stencil
        for (int j = -RADIUS; j <= RADIUS; j++) {
            out[i] += weights[j + RADIUS] * in[i + j];
        }
        out[i] = out[i] / (2 * RADIUS + 1);
    }
}

Serial Implementation
int main() {
  int size = N * sizeof(float);
  int wsize = (2 * RADIUS + 1) * sizeof(float);
  // allocate resources
  float *weights = (float *)malloc(wsize);
  float *in = (float *)malloc(size);
  float *out = (float *)malloc(size);
  initializeWeights(weights, RADIUS);
  initializeArray(in, N);
  applyStencil1D(RADIUS, N-RADIUS, weights, in, out);
  // free resources
  free(weights); free(in); free(out);
}

void applyStencil1D(int sIdx, int eIdx, const float *weights, float *in, float *out) {
  for (int i = sIdx; i < eIdx; i++) {
    out[i] = 0;
    // loop over all elements in the stencil
    for (int j = -RADIUS; j <= RADIUS; j++) {
      out[i] += weights[j + RADIUS] * in[i + j];
    }
    out[i] = out[i] / (2 * RADIUS + 1);
  }
}
int main() {
    int size = N * sizeof(float);
    int wsize = (2 * RADIUS + 1) * sizeof(float);
    // allocate resources
    float *weights = (float *)malloc(wsize);
    float *in = (float *)malloc(size);
    float *out = (float *)malloc(size);
    initializeWeights(weights, RADIUS);
    initializeArray(in, N);

    applyStencil1D(RADIUS, N-RADIUS, weights, in, out);

    // free resources
    free(weights); free(in); free(out);
}

void applyStencil1D(int sIdx, int eIdx, const float *weights, float *in, float *out) {
    for (int i = sIdx; i < eIdx; i++) {
        out[i] = 0;
        // loop over all elements in the stencil
        for (int j = -RADIUS; j <= RADIUS; j++) {
            out[i] += weights[j + RADIUS] * in[i + j];
        }
        out[i] = out[i] / (2 * RADIUS + 1);
    }
}

<table>
<thead>
<tr>
<th>CPU</th>
<th>MEElements/s</th>
</tr>
</thead>
<tbody>
<tr>
<td>i7-930</td>
<td>30</td>
</tr>
</tbody>
</table>
Parallel Algorithm

Serial: One element at a time

Parallel: Many elements at a time

\[ \downarrow = \text{Thread} \]
The Parallel Implementation

```c
void main() {
    int size = N * sizeof(float);
    int wsize = (2 * RADIUS + 1) * sizeof(float);
    //allocate resources
    float *weights = (float*)malloc(wsize);
    float *in = (float*)malloc(size);
    float *out = (float*)malloc(size);
    initializeWeights(weights, RADIUS);
    initializeArray(in, N);
    float *d_weights; cudaMalloc(&d_weights, wsize);
    float *d_in; cudaMalloc(&d_in, size);
    float *d_out; cudaMalloc(&d_out, size);
    cudaMemcpy(d_weights, weights, wsize, cudaMemcpyHostToDevice);
    cudaMemcpy(d_in, in, size, cudaMemcpyHostToDevice);
    applyStencil1D<<<N/512, 512>>>(RADIUS, N-RADIUS, d_weights, d_in, d_out);
    cudaMemcpy(out, d_out, size, cudaMemcpyDeviceToHost);
    //free resources
    free(weights); free(in); free(out);
    cudaFree(d_weights); cudaFree(d_in); cudaFree(d_out);
}
```

```c
__global__ void applyStencil1D(int sIdx, int eIdx, const float *weights, float *in, float *out) {
    int i = sIdx + blockIdx.x*blockDim.x + threadIdx.x;
    if (i < eIdx) {
        out[i] = 0;
        //loop over all elements in the stencil
        for (int j = -RADIUS; j <= RADIUS; j++) {
            out[i] += weights[j + RADIUS] * in[i + j];
        }
        out[i] = out[i] / (2 * RADIUS + 1);
    }
}
```

The GPU kernel
The Parallel Implementation

```c
__global__ void applyStencil1D(int sIdx, int eIdx, const float *weights, float *in, float *out) {
    int i = sIdx + blockIdx.x*blockDim.x + threadIdx.x;
    if (i < eIdx) {
        out[i] = 0;
        //loop over all elements in the stencil
        for (int j = -RADIUS; j <= RADIUS; j++) {
            out[i] += weights[j + RADIUS] * in[i + j];
        }
        out[i] = out[i] / (2 * RADIUS + 1);
    }
}
```

```c
global__ void main() {
    int size = N * sizeof(float);
    int wsize = (2 * RADIUS + 1) * sizeof(float);
    //allocate resources
    float *weights = (float *)malloc(wsize);
    float *in = (float *)malloc(size);
    float *out = (float *)malloc(size);
    initializeWeights(weights, RADIUS);
    initializeArray(in, N);
    float *d_weights; cudaMalloc(&d_weights, wsize);
    float *d_in; cudaMalloc(&d_in, size);
    float *d_out; cudaMalloc(&d_out, size);
    cudaMemcpy(d_weights, weights, wsize, cudaMemcpyHostToDevice);
    cudaMemcpy(d_in, in, size, cudaMemcpyHostToDevice);
    applyStencil1D<<<N/512, 512>>>(RADIUS, N-RADIUS, d_weights, d_in, d_out);
    cudaMemcpy(out, d_out, size, cudaMemcpyDeviceToHost);
    //free resources
    free(weights); free(in); free(out);
    cudaFree(d_weights); cudaFree(d_in); cudaFree(d_out);
}
```

Allocate GPU memory
The Parallel Implementation

```c
void main() {
  int size = N * sizeof(float);
  int wsize = (2 * RADIUS + 1) * sizeof(float);
  //allocate resources
  float *weights = (float *)malloc(wsize);
  float *in = (float *)malloc(size);
  float *out = (float *)malloc(size);
  initializeWeights(weights, RADIUS);
  initializeArray(in, N);
  float *d_weights; cudaMalloc(&d_weights, wsize);
  float *d_in; cudaMalloc(&d_in, size);
  float *d_out; cudaMalloc(&d_out, size);
  cudaMemcpy(d_weights, weights, wsize, cudaMemcpyHostToDevice);
  cudaMemcpy(d_in, in, size, cudaMemcpyHostToDevice);
  applyStencil1D<<<N/512, 512>>>(RADIUS, N-RADIUS, d_weights, d_in, d_out);
  cudaMemcpy(out, d_out, size, cudaMemcpyDeviceToHost);
  //free resources
  free(weights); free(in); free(out);
  cudaFree(d_weights); cudaFree(d_in); cudaFree(d_out);
}
```

```c
__global__ void applyStencil1D(int sIdx, int eIdx, const float *weights, float *in, float *out) {
  int i = sIdx + blockIdx.x*blockDim.x + threadIdx.x;
  if (i < eIdx) {
    out[i] = 0;
    //loop over all elements in the stencil
    for (int j = -RADIUS; j <= RADIUS; j++) {
      out[i] += weights[j + RADIUS] * in[i + j];
    }
    out[i] = out[i] / (2 * RADIUS + 1);
  }
}
```
void main() {
    int size = N * sizeof(float);
    int wsize = (2 * RADIUS + 1) * sizeof(float);
    //allocate resources
    float *weights = (float *)malloc(wsize);
    float *in = (float *)malloc(size);
    float *out = (float *)malloc(size);
    initializeWeights(weights, RADIUS);
    initializeArray(in, N);
    float *d_weights; cudaMalloc(&d_weights, wsize);
    float *d_in; cudaMalloc(&d_in, size);
    float *d_out; cudaMalloc(&d_out, size);
    cudaMemcpy(d_weights, weights, wsize, cudaMemcpyHostToDevice);
    cudaMemcpy(d_in, in, size, cudaMemcpyHostToDevice);
    applyStencil1D<<<N/512, 512>>>(RADIUS, N-RADIUS, d_weights, d_in, d_out);
    cudaMemcpy(out, d_out, size, cudaMemcpyDeviceToHost);
    //free resources
    free(weights); free(in); free(out);
    cudaFree(d_weights); cudaFree(d_in); cudaFree(d_out);
}

__global__ void applyStencil1D(int sIdx, int eIdx, const float *weights, float *in, float *out) {
    int i = sIdx + blockIdx.x*blockDim.x + threadIdx.x;
    if( i < eIdx ) {
        out[i] = 0;
        //loop over all elements in the stencil
        for( int j = -RADIUS; j <= RADIUS; j++) {
            out[i] += weights[j + RADIUS] * in[i + j];
        }
        out[i] = out[i] / (2 * RADIUS + 1);
    }
}
The Parallel Implementation

void main() {
    int size = N * sizeof(float);
    int wsize = (2 * RADIUS + 1) * sizeof(float);

    //allocate resources
    float *weights = (float *)malloc(wsize);
    float *in = (float *)malloc(size);
    float *out = (float *)malloc(size);
    initializeWeights(weights, RADIUS);
    initializeArray(in, N);

    float *d_weights; cudaMalloc(&d_weights, wsize);
    float *d_in; cudaMalloc(&d_in, size);
    float *d_out; cudaMalloc(&d_out, size);

    cudaMemcpy(d_weights, weights, wsize, cudaMemcpyHostToDevice);
    cudaMemcpy(d_in, in, size, cudaMemcpyHostToDevice);
    applyStencil1D<<<N/512, 512>>>(RADIUS, N-RADIUS, d_weights, d_in, d_out);
    cudaMemcpy(out, d_out, size, cudaMemcpyDeviceToHost);

    //free resources
    free(weights); free(in); free(out);
    cudaFree(d_weights); cudaFree(d_in); cudaFree(d_out);
}

__global__ void applyStencil1D(int sIdx, int eIdx, const float *weights, float *in, float *out) {
    int i = sIdx + blockIdx.x*blockDim.x + threadIdx.x;
    if (i < eIdx) {
        out[i] = 0;
        //loop over all elements in the stencil
        for (int j = -RADIUS; j <= RADIUS; j++) {
            out[i] += weights[j + RADIUS] * in[i + j];
        }
        out[i] = out[i] / (2 * RADIUS + 1);
    }
}
The Parallel Implementation

```c
void main() {
    int size = N * sizeof(float);
    int wsize = (2 * RADIUS + 1) * sizeof(float);
    // allocate resources
    float *weights = (float *)malloc(wsize);
    float *in = (float *)malloc(size);
    float *out = (float *)malloc(size);
    initializeWeights(weights, RADIUS);
    initializeArray(in, N);
    float *d_weights; cudaMalloc(&d_weights, wsize);
    float *d_in; cudaMalloc(&d_in, size);
    float *d_out; cudaMalloc(&d_out, size);
    cudaMemcpy(d_weights, weights, wsize, cudaMemcpyHostToDevice);
    cudaMemcpy(d_in, in, size, cudaMemcpyHostToDevice);
    applyStencil1D<<<N/512, 512>>>(RADIUS, N-RADIUS, d_weights, d_in, d_out);
    cudaMemcpy(out, d_out, size, cudaMemcpyDeviceToHost);
    // free resources
    free(weights); free(in); free(out);
    cudaFree(d_weights); cudaFree(d_in); cudaFree(d_out);
}

__global__ void applyStencil1D(int sIdx, int eIdx, const float *weights, float *in, float *out) {
    int i = sIdx + blockIdx.x*blockDim.x + threadIdx.x;
    if (i < eIdx) {
        out[i] = 0;
        // loop over all elements in the stencil
        for (int j = -RADIUS; j <= RADIUS; j++) {
            out[i] += weights[j + RADIUS] * in[i + j];
        }
        out[i] = out[i] / (2 * RADIUS + 1);
    }
}
```

NVIDIA [S. Satoor]
void main() {
    int size = N * sizeof(float);
    int wsize = (2 * RADIUS + 1) * sizeof(float);
    //allocate resources
    float *weights = (float *)malloc(wsize);
    float *in = (float *)malloc(size);
    float *out= (float *)malloc(size);
    initializeWeights(weights, RADIUS);
    initializeArray(in, N);
    float *d_weights;  cudaMalloc(&d_weights, wsize);
    float *d_in;       cudaMalloc(&d_in, size);
    float *d_out;      cudaMalloc(&d_out, size);
    cudaMemcpy(d_weights,weights,wsize,cudaMemcpyHostToDevice);
    cudaMemcpy(d_in, in, size, cudaMemcpyHostToDevice);
    applyStencil1D<<<N/512, 512>>>(RADIUS, N-RADIUS, d_weights, d_in, d_out);
    cudaMemcpy(out, d_out, size, cudaMemcpyDeviceToHost);
    free(weights); free(in); free(out);
    cudaFree(d_weights); cudaFree(d_in); cudaFree(d_out);
}

<table>
<thead>
<tr>
<th>Device</th>
<th>Algorithm</th>
<th>MEElements/s</th>
<th>Speedup</th>
</tr>
</thead>
<tbody>
<tr>
<td>i7-930*</td>
<td>Optimized &amp; Parallel</td>
<td>130</td>
<td>1x</td>
</tr>
<tr>
<td>Tesla C2075</td>
<td>Simple</td>
<td>285</td>
<td>2.2x</td>
</tr>
</tbody>
</table>

__global__ void applyStencil1D(int sIdx, int eIdx, const float *weights, float *in, float *out) {
    int i = sIdx + blockIdx.x*blockDim.x + threadIdx.x;
    if( i < eIdx ) {
        out[i] = 0;
        //loop over all elements in the stencil
        for (int j = -RADIUS; j <= RADIUS; j++) {
            out[i] += weights[j + RADIUS] * in[i + j];
        }
        out[i] = out[i] / (2 * RADIUS + 1);
    }
}
Gauging Code Performance: The Expeditious Solution...

- Set the right environment variable and run your executable [illustrated on Euler]:

```
>> nvcc -O3 -gencode arch=compute_20,code=sm_20 testV4.cu -o testV4_20
>> export CUDA_PROFILE=1
>> ./testV4_20
>> cat cuda_profile_0.log
```

```bash
# CUDA_PROFILE_LOG_VERSION 2.0
# CUDA_DEVICE 0 GeForce GTX 480
# TIMESTAMPFACTOR fffff6c689a404a8
method, gputime, cputime, occupancy
method=[ memcpyHtoD ] gputime=[ 1001.952 ] cputime=[ 1197.000 ]
method=[ memcpyDtoH ] gputime=[ 1394.144 ] cputime=[ 2533.000 ]
```
Gauging Code Performance: The Expeditious Solution...

```bash
>> nvcc -O3 -gencode arch=compute_20,code=sm_20 testV4.cu -o testV4_20
>> ./testV4_20

# CUDA_PROFILE_LOG_VERSION 2.0
# CUDA_DEVICE 0 GeForce GTX 480
# TIMESTAMPFACTOR fffff6c689a404a8
method,gputime,cputime,occupancy
method=[ memcpyHtoD ] gputime=[ 1001.952 ] cputime=[ 1197.000 ]
method=[ memcpyDtoH ] gputime=[ 1394.144 ] cputime=[ 2533.000 ]
```

```bash
>> nvcc -O3 -gencode arch=compute_10,code=sm_10 testV4.cu -o testV4_10
>> ./testV4_10

# CUDA_PROFILE_LOG_VERSION 2.0
# CUDA_DEVICE 0 GeForce GT 130M
# TIMESTAMPFACTOR 12764ee9b183e71e
method,gputime,cputime,occupancy
method=[ memcpyHtoD ] gputime=[ 1815.424 ] cputime=[ 2787.856 ]
method=[ _Z14applyStencil1DiiPKfPfS1_ ] gputime=[ 47332.9 ] cputime=[ 8.469 ] occupancy=[0.67]
method=[ memcpyDtoH ] gputime=[ 3535.648 ] cputime=[ 4555.577 ]
```
Gauging Code Performance: The Expeditious Solution...

```c
>> nvcc -O3 -gencode arch=compute_20,code=sm_20 testV4.cu -o testV4_20
>> ./testV4_20

# CUDA_PROFILE_LOG_VERSION 2.0
# CUDA_DEVICE 0 GeForce GTX 480
# TIMESTAMPFACTOR fffffff6c689a404a8
method, gputime, cputime, occupancy
method=[ memcpyHtoD ] gputime=[ 1001.952 ] cputime=[ 1197.000 ]
method=[ memcpyDtoH ] gputime=[ 1394.144 ] cputime=[ 2533.000 ]
```

```c
>> nvcc -O3 -gencode arch=compute_10,code=sm_10 testV4.cu -o testV4_10
>> ./testV4_10

# CUDA_PROFILE_LOG_VERSION 2.0
# CUDA_DEVICE 0 GeForce GT 130M
# TIMESTAMPFACTOR 12764ee9b183e71e
method, gputime, cputime, occupancy
method=[ memcpyHtoD ] gputime=[ 1815.424 ] cputime=[ 2787.856 ]
method=[ _Z14applyStencil1DiPKfPfS1_ ] gputime=[ 47332.9 ] cputime=[ 8.469 ] occupancy=[ 0.67 ]
method=[ memcpyDtoH ] gputime=[ 3535.648 ] cputime=[ 4555.577 ]
```
Gauging Code Performance: The Expeditious Solution...

Compute capability 2.0 (Fermi)

```bash
>> nvcc -O3 -gencode arch=compute_20,code=sm_20 testV4.cu -o testV4_20
>> ./testV4_20
```

```c
# CUDA_PROFILE_LOG_VERSION 2.0
# CUDA_DEVICE 0 GeForce GTX 480
# TIMESTAMPFACTOR fffff6c689a404a8
method, gputime, cputime, occupancy
method=[ memcpyHtoD ] gputime=[ 1001.952 ] cputime=[ 1197.000 ]
method=[ memcpyDtoH ] gputime=[ 1394.144 ] cputime=[ 2533.000 ]
```

Compute capability 1.0 (Tesla/G80)

```bash
>> nvcc -O3 -gencode arch=compute_10,code=sm_10 testV4.cu -o testV4_10
>> ./testV4_10
```

```c
# CUDA_PROFILE_LOG_VERSION 2.0
# CUDA_DEVICE 0 GeForce GT 130M
# TIMESTAMPFACTOR 12764ee9b183e71e
method, gputime, cputime, occupancy
method=[ memcpyHtoD ] gputime=[ 1815.424 ] cputime=[ 2787.856 ]
method=[ __Z14applyStencillDiiPKfPfs1__ ] gputime=[ 47332.9 ] cputime=[ 8.469 ] occupancy=[0.67]
method=[ memcpyDtoH ] gputime=[ 3535.648 ] cputime=[ 4555.577 ]
```
nvvp—the NVIDIA Visual Profiler: 
The Advanced Approach

- Available on Euler
- Provides a nice GUI and broad but detailed information regarding your run
- Compared to the “the expeditious approach”, it represents one step up in terms of amount of information and level of detail/resolution
- Many bells & whistles, covering here the basics using 1D stencil example
- Acknowledgement: Discussion on nvvp uses material from NVIDIA (S. Satoor).
  - Slides that include this material marked by “NVIDIA [S. Satoor]→” sign at bottom of slide
Application Optimization Process [Revisited]

- Identify Optimization Opportunities
  - 1D stencil algorithm

- Parallelize with CUDA, confirm functional correctness
  - `cuda-gdb`, `cuda-memcheck`

- Optimize
  - …dealing with this next
NVIDIA Visual Profiler

Timeline of CPU and GPU activity

Kernel and memcpy details
Detecting Low Memory Throughput

- Spend majority of time in data transfer
  - Often can be overlapped with preceding or following computation

- From timeline can see that throughput is low
  - PCIe x16 can sustain > 5GB/s
How do we know when there is an optimization opportunity?
- Timeline visualization seems to indicate an opportunity
- Documentation gives guidance and strategies for tuning
  - CUDA Best Practices Guide
  - CUDA Programming Guide

Visual Profiler analyzes your application
- Uses timeline and other collected information
- Highlights specific guidance from Best Practices
- Like having a customized Best Practices Guide for your application
Visual Profiler Analysis

Several types of analysis are provided.

Analysis pointing out low memcpy throughput.
Online Optimization Help

Low Memcpy Throughput [ 997.19 MB/s avg, for memcpys accounting for 68.1% of all memcpy time ]
The memory copies are not fully using the available host to device bandwidth.

Each analysis has link to Best Practices documentation

Pinned Memory

Page-locked or pinned memory transfers attain the highest bandwidth between the host and the device. On PCIe ×16 Gen2 cards, for example, pinned memory can attain greater than 5 GBps transfer rates.

Pinned memory is allocated using the cudaMallocHost() or cudaHostAlloc() functions in the Runtime API. The bandwidthTest.cu program in the CUDA SDK shows how to use these functions as well as how to measure memory transfer performance.

Pinned memory should not be overused. Excessive use can reduce overall system performance because pinned memory is a scarce resource. How much is too much is difficult to tell in advance, so as with all optimizations, test the applications and the systems they run on for optimal performance parameters.

Parent topic: Data Transfer Between Host and Device

NVIDIA [S. Satoor] →
int main() {
    int size = N * sizeof(float);
    int wsize = (2 * RADIUS + 1) * sizeof(float);
    //allocate resources
    float *weights; cudaMallocHost(&weights, wsize);
    float *in; cudaMallocHost(&in, size);
    float *out; cudaMallocHost(&out, size);
    initializeWeights(weights, RADIUS);
    initializeArray(in, N);
    float *d_weights; cudaMalloc(&d_weights);
    float *d_in; cudaMalloc(&d_in);
    float *d_out; cudaMalloc(&d_out);
    ...
}

CPU allocations use pinned memory to enable fast memcpy

No other changes
Pinned CPU Memory Result

GPU PINNED: 0.0297912 seconds, 4.50528 GBytes/s, 0.563158 GElements/s
Pinned CPU Memory Result

<table>
<thead>
<tr>
<th>Device</th>
<th>Algorithm</th>
<th>MEElements/s</th>
<th>Speedup</th>
</tr>
</thead>
<tbody>
<tr>
<td>i7-930*</td>
<td>Optimized &amp; Parallel</td>
<td>130</td>
<td>1x</td>
</tr>
<tr>
<td>Tesla C2075</td>
<td>Simple</td>
<td>285</td>
<td>2.2x</td>
</tr>
<tr>
<td>Tesla C2075</td>
<td>Pinned Memory</td>
<td>560</td>
<td>4.3x</td>
</tr>
</tbody>
</table>

*4 cores + hyperthreading
Application Optimization Process

[Revisited]

- Identify Optimization Opportunities
  - 1D stencil algorithm

- Parallelize with CUDA, confirm functional correctness
  - Debugger
  - Memory Checker

- Optimize
  - Profiler (pinned memory)
Application Optimization Process
[Revisited]

- Identify Optimization Opportunities
  - 1D stencil algorithm

- Parallelize with CUDA, confirm functional correctness
  - Debugger
  - Memory Checker

Optimize
  - Profiler (pinned memory)
- Advanced optimization
  - Larger time investment
  - Potential for larger speedup

Asynchronous Transfers and Overlapping Transfers with Computation

Data transfers between the host and the device using `cudaMemcpy()` are blocking transfers; that is, control is returned to the host thread only after the data transfer is complete. The `cudaMemcpyAsync()` function is a non-blocking variant of `cudaMemcpy()` in which control is returned immediately to the host thread. In contrast with `cudaMemcpy()`, the asynchronous transfer version requires pinned host memory (see Pinned Memory), and it contains an additional argument, a stream ID. A stream is simply a sequence of operations that are performed in order on the device. Operations in different streams can be interleaved and in some cases overlapped—a property that can be used to hide data transfers between the host and the device.

Asynchronous transfers enable overlap of data transfers with computation in two different ways. On all CUDA-enabled devices, it is possible to overlap host computation with asynchronous data transfers and with device computations. For example, Overlapping computation and data transfers demonstrates how host computation in the...
Data Partitioning Example

Partition data into TWO chunks

chunk 1

chunk 2

in

out
Data Partitioning Example

chunk 1

memcpy
compute
memcpy

chunk 2

in

out
Data Partitioning Example

chunk 1

in

memcpy

compute

memcpy

compute

memcpy

chunk 2

out

NVIDIA [S. Satoor]
Overlapped Compute/Memcpy

[problem broken into 16 chunks]
Overlapped Compute/Memcpy

Compute time completely “hidden”

Exploit dual memcpy engines
## Overlapped Compute/Memcpy

<table>
<thead>
<tr>
<th>Device</th>
<th>Algorithm</th>
<th>MEElements/s</th>
<th>Speedup</th>
</tr>
</thead>
<tbody>
<tr>
<td>i7-930*</td>
<td>Optimized &amp; Parallel</td>
<td>130</td>
<td>1x</td>
</tr>
<tr>
<td>Tesla C2075</td>
<td>Simple</td>
<td>285</td>
<td>2.2x</td>
</tr>
<tr>
<td>Tesla C2075</td>
<td>Pinned Memory</td>
<td>560</td>
<td>4.3x</td>
</tr>
<tr>
<td>Tesla C2075</td>
<td>Overlap</td>
<td>935</td>
<td>7.2x</td>
</tr>
</tbody>
</table>

ME964: Use of multiple streams covered in 10 days
Application Optimization Process
[Revisited]

- Identify Optimization Opportunities
  - 1D stencil algorithm

- Parallelize with CUDA, confirm functional correctness
  - Debugger
  - Memory Checker

- Optimize
  - Profiler (pinned memory)
  - Profiler (overlap memcpy and compute)
Iterative Optimization

- Identify Optimization Opportunities
- Parallelize
- Optimize
Optimization Summary
[Looking back at this journey…]

- Initial CUDA parallelization
  - Expeditious, kernel is almost word-for-word replica of sequential code
  - 2.2x speedup

- Optimize memory throughput
  - Expeditious, need to know about pinned memory
  - 4.3x speedup

- Overlap compute and data movement
  - More involved, need to know about the inner works of CUDA
  - Problem should be large enough to justify mem-transfer/execution
  - 7.2x speedup
Take Home Message...

- Regard CUDA as a way to accelerate the compute-intensive parts of your application

- Visual profiler (nvpp) helps in performance analysis and optimization
CUDA Debugging
Acknowledgments

- CUDA debugging slides include material provided by Sanjiv Satoor of NVIDIA India

- Mistakes, if any, are due to my changes/edits
Terminology

[Review]

- **Kernel**
  - Function to be executed in parallel on one CUDA device
  - A kernel is executed by multiple blocks of threads

- **Block**
  - 3-dimensional
  - Made up of a collection of threads

- **Warp**
  - Group of 32 threads scheduled for execution as a unit

- **Thread**
  - Smallest unit of work
Terminology

[Review]

- **Divergence**
  - Occurs when any two threads on the same warp are slated to execute different instructions
  - Active threads within a warp: threads currently executing device code
  - Divergent threads within a warp: threads that are waiting for their turn or are done with their turn.

- **Program Counter (PC)**
  - A processor register that holds the address (in the virtual address space) of the next instruction in the instruction sequence
  - Each thread has a PC
  - Useful with **cuda-gdb** debugging to navigate the instruction sequence
Debugging Solutions

CUDA-GDB
(Linux & Mac)

CUDA-MEMCHECK
(Linux, Mac, & Windows)

NVIDIA Parallel NSight
(Windows)

Allinea DDT

Rogue Wave TotalView
CUDA-GDB GUI Wrappers

GNU DDD

GNU Emacs
CUDA-GDB Main Features

- All the standard GDB debugging features
- Integrated CPU and GPU debugging within a single session
- Breakpoints and Conditional Breakpoints
- Inspect memory, registers, local/shared/global variables
- Supports multiple GPUs, multiple contexts, multiple kernels
- Source and Assembly (SASS) Level Debugging
- Runtime Error Detection (stack overflow,...)
Recommended Compilation Flags

- Compile code for your target architecture:
  - Tesla: -gencode arch=compute_10,code=sm_10
  - Fermi: -gencode arch=compute_20,code=sm_20

- Compile code with the debug flags:
  - Host code: -g
  - Device code: -G

- Example:

  ```
  $ nvcc -g -G -gencode arch=compute_20,code=sm_20 test.cu -o test
  ```
Usage
[On your desktop]

- Invoke `cuda-gdb` from the command line:

```bash
$ cuda-gdb my_application
(cuda-gdb) 
```
When you request resources for running a debugging session on Euler don’t reserve the GPU in exclusive mode (use “default”):

```
>> qsub -I -l nodes=1:gpus=1:default -X (qsub –eye –ell node…)
>> cuda-gdb myApp
```

You can use `ddd` as a front end (you need the `-X` in the `qsub` command):

```
>> ddd --debugger cuda-gdb myApp
```

If you don’t have `ddd` around, use at least

```
>> cuda-gdb -tui myApp
```
Program Execution Control
Execution Control

- Execution Control is identical to host debugging:
  - Launch the application
    
    \texttt{(cuda-gdb) run}
  
  - Resume the application (all host threads and device threads)
    
    \texttt{(cuda-gdb) continue}
  
  - Kill the application
    
    \texttt{(cuda-gdb) kill}
  
  - Interrupt the application: CTRL-C
Execution Control

- Single-Stepping

<table>
<thead>
<tr>
<th>Single-Stepping</th>
<th>At the source level</th>
<th>At the assembly level</th>
</tr>
</thead>
<tbody>
<tr>
<td>Over function calls</td>
<td>next</td>
<td>nexti</td>
</tr>
<tr>
<td>Into function calls</td>
<td>step</td>
<td>stepi</td>
</tr>
</tbody>
</table>

- Behavior varies when stepping `__syncthreads()`

<table>
<thead>
<tr>
<th>PC at a barrier?</th>
<th>Single-stepping applies to</th>
<th>Notes</th>
</tr>
</thead>
<tbody>
<tr>
<td>Yes</td>
<td>Active and divergent threads of the warp in focus and all the warps that are running the same block.</td>
<td>Required to step over the barrier.</td>
</tr>
<tr>
<td>No</td>
<td><strong>Active threads</strong> in the warp in focus only.</td>
<td></td>
</tr>
</tbody>
</table>
Breakpoints

- By name
  
  ```
  (cuda-gdb) break my_kernel
  (cuda-gdb) break _Z6kernelIfiEvPT_PT0
  ```

- By file name and line number
  
  ```
  (cuda-gdb) break test.cu:380
  ```

- By address
  
  ```
  (cuda-gdb) break *0x3e840a8
  (cuda-gdb) break *$pc
  ```

- At every kernel launch
  
  ```
  (cuda-gdb) set cuda break_on_launch application
  ```
Conditional Breakpoints

- Only reports hit breakpoint if condition is met
  - All breakpoints are still hit
  - Condition is evaluated every time for all the threads
  - Typically slows down execution

- Condition
  - C/C++ syntax
  - No function calls
  - Supports built-in variables (blockIdx, threadIdx, ...)

155
Conditional Breakpoints

- Set at breakpoint creation time
  
  ```(cuda-gdb) break my_kernel if threadIdx.x == 13```

- Set after the breakpoint is created
  
  - Breakpoint 1 was previously created
  
  ```(cuda-gdb) condition 1 blockIdx.x == 0 && n > 3```
Thread Focus
Thread Focus

- There are thousands of threads to deal with. Can’t display all of them

- Thread focus dictates which thread you are looking at

- Some commands apply only to the thread in focus
  - Print local or shared variables
  - Print registers
  - Print stack contents

- Attributes of the “thread focus”
  - Kernel index : unique, assigned at kernel’s launch time
  - Block index : the application blockIdx
  - Thread index : the application threadIdx
Devices

- To obtain the list of devices in the system:

```
(cuda-gdb) info cuda devices
```

<table>
<thead>
<tr>
<th>Dev</th>
<th>Desc</th>
<th>Type</th>
<th>SMs</th>
<th>Wps/SM</th>
<th>Lns/Wp</th>
<th>Regs/Ln</th>
<th>Active SMs</th>
<th>Mask</th>
</tr>
</thead>
<tbody>
<tr>
<td>*</td>
<td>0</td>
<td>gf100</td>
<td>sm_20</td>
<td>14</td>
<td>48</td>
<td>32</td>
<td>64</td>
<td>0xfff</td>
</tr>
<tr>
<td>1</td>
<td>gt200</td>
<td>sm_13</td>
<td>30</td>
<td>32</td>
<td>32</td>
<td>128</td>
<td>0</td>
<td>0x0</td>
</tr>
</tbody>
</table>

- The * indicates the device of the kernel currently in focus
- Provides an overview of the hardware that supports the code
Kernels

To obtain the list of running kernels:

(cuda-gdb) info cuda kernels

<table>
<thead>
<tr>
<th>Kernel</th>
<th>Dev</th>
<th>Grid</th>
<th>SMs Mask</th>
<th>GridDim</th>
<th>BlockDim</th>
<th>Name Args</th>
</tr>
</thead>
<tbody>
<tr>
<td>*</td>
<td>1</td>
<td>0</td>
<td>0x3fff</td>
<td>(240,1,1)</td>
<td>(128,1,1)</td>
<td>acos parms=...</td>
</tr>
<tr>
<td>2</td>
<td>0</td>
<td>3</td>
<td>0x4000</td>
<td>(240,1,1)</td>
<td>(128,1,1)</td>
<td>asin parms=...</td>
</tr>
</tbody>
</table>

- The * indicates the kernel currently in focus
- There is a one-to-one mapping between a kernel id (unique id across multiple GPUs) and a (dev,grid) tuple. The grid id is unique per GPU only
- The name of the kernel is displayed as are its size and its parameters
- Provides an overview of the code running on the hardware
Thread Focus

- To switch focus to any currently running thread

```plaintext
(cuda-gdb) cuda kernel 2 block 1,0,0 thread 3,0,0
(Switching focus to CUDA kernel 2 block (1,0,0), thread (3,0,0))

(cuda-gdb) cuda kernel 2 block 2 thread 4
(Switching focus to CUDA kernel 2 block (2,0,0), thread (4,0,0))

(cuda-gdb) cuda thread 5
(Switching focus to CUDA kernel 2 block (2,0,0), thread (5,0,0))
```
Thread Focus

- To obtain the current focus:

  (cuda-gdb) cuda kernel block thread
  kernel 2 block (2,0,0), thread (5,0,0)

  (cuda-gdb) cuda thread
  thread (5,0,0)
Threads

- To obtain the list of running threads for kernel 2:

```
(cuda-gdb) info cuda threads kernel 2
```

<table>
<thead>
<tr>
<th>Block</th>
<th>Thread</th>
<th>To Block</th>
<th>Thread</th>
<th>Cnt</th>
<th>PC</th>
<th>Filename</th>
<th>Line</th>
</tr>
</thead>
<tbody>
<tr>
<td>*</td>
<td>(0,0,0)</td>
<td>(3,0,0)</td>
<td>(7,0,0)</td>
<td>32</td>
<td>0x7fae70</td>
<td>acos.cu</td>
<td>380</td>
</tr>
<tr>
<td></td>
<td>(4,0,0)</td>
<td>(7,0,0)</td>
<td>(7,0,0)</td>
<td>32</td>
<td>0x7fae60</td>
<td>acos.cu</td>
<td>377</td>
</tr>
</tbody>
</table>

- Threads are displayed in (block, thread) ranges
- Divergent threads are in separate ranges
- The * indicates the range where the thread in focus resides
- Threads displayed as (block, thread) ranges
- **Cnt** indicates the number of threads within each range
  - All threads in the same range are contiguous (no hole)
  - All threads in the same range shared the same PC (and filename/line number)
Program State Inspection
Stack Trace

- Same (aliased) commands as in `gdb`:
  
  ```
  (cuda-gdb) where
  (cuda-gdb) bt
  (cuda-gdb) info stack
  ```

- Applies to the thread in focus

- On Tesla, all the functions are always inlined
Stack Trace

(cuda-gdb) info stack

#0 fibo_aux (n=6) at fibo.cu:88
#1 0x7bbda0 in fibo_aux (n=7) at fibo.cu:90
#2 0x7bbda0 in fibo_aux (n=8) at fibo.cu:90
#3 0x7bbda0 in fibo_aux (n=9) at fibo.cu:90
#4 0x7bbda0 in fibo_aux (n=10) at fibo.cu:90
#5 0x7cfdb8 in fibo_main<<<(1,1,1),(1,1,1)>>>(...) at fibo.cu:95
Source Variables

- Source variable must be live (in the scope)
- Read a source variable

```
(cuda-gdb) print my_variable
$1 = 3
(cuda-gdb) print &my_variable
$2 = (@global int *) 0x200200020
```

- Write a source variable

```
(cuda-gdb) print my_variable = 5
$3 = 5
```
Memory

- Memory read & written like source variables
  ```
  (cuda-gdb) print *my_pointer
  ```

- May require storage specifier when ambiguous
  ```
  @global, @shared, @local
  @generic, @texture, @parameter
  ```
  ```
  (cuda-gdb) print * (@global int *) my_pointer
  (cuda-gdb) print ((@texture float **) my_texture)[0][3]
  ```
Hardware Registers

- CUDA Registers
  - Virtual PC: $pc (read-only)
  - SASS registers: $R0, $R1,...

- Show a list of registers (no argument to get all of them)

```
(cuda-gdb) info registers R0 R1 R4
R0  0x6  6
R1  0xffffc68 16776296
R4  0x6  6
```

- Modify one register

```
(cuda-gdb) print $R3 = 3
```
Code Disassembly

- Must have `cuobjdump` in `$PATH`

```
(cuda-gdb) x/10i $pc
0x123830a8 <_Z9my_kernel10params+8>:       MOV R0, c [0x0] [0x8]
0x123830b0 <_Z9my_kernel10params+16>:      MOV R2, c [0x0] [0x14]
0x123830b8 <_Z9my_kernel10params+24>:      IMUL.U32.U32 R0, R0, R2
0x123830c0 <_Z9my_kernel10params+32>:      MOV R2, R0
0x123830c8 <_Z9my_kernel10params+40>:      S2R R0, SR_CTAid_X
0x123830d0 <_Z9my_kernel10params+48>:      MOV R0, R0
0x123830d8 <_Z9my_kernel10params+56>:      MOV R3, c [0x0] [0x8]
0x123830e0 <_Z9my_kernel10params+64>:      IMUL.U32.U32 R0, R0, R3
0x123830e8 <_Z9my_kernel10params+72>:      MOV R0, R0
0x123830f0 <_Z9my_kernel10params+80>:      MOV R0, R0
```
Run-Time Error Detection
CUDA-MEMCHECK

- Stand-alone run-time error checker tool
- Detects memory errors like stack overflow, illegal global address,...
- Similar to valgrind
- No need to recompile the application
- Not all the error reports are precise
- Once used within cuda-gdb, the kernel launches are blocking
# CUDA-Memcheck Errors

<table>
<thead>
<tr>
<th>Error Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>Illegal global address</td>
</tr>
<tr>
<td>Misaligned global address</td>
</tr>
<tr>
<td>Stack memory limit exceeded</td>
</tr>
<tr>
<td>Illegal shared/local address</td>
</tr>
<tr>
<td>Misaligned shared/local address</td>
</tr>
<tr>
<td>Instruction accessed wrong memory</td>
</tr>
<tr>
<td>PC set to illegal value</td>
</tr>
<tr>
<td>Illegal instruction encountered</td>
</tr>
<tr>
<td>Illegal global address</td>
</tr>
</tbody>
</table>
CUDA-MEMCHECK

- Integrated in `cuda-gdb`
  - More precise errors when used from `cuda-gdb`
  - Must be activated before the application is launched

```
(cuda-gdb) set cuda memcheck on
```

- What does it mean “more precise”?
  - Precise
    - Exact thread idx
    - Exact PC
  - Not precise
    - A group of threads or blocks
    - The PC is several instructions after the offending load/store
(cuda-gdb) set cuda memcheck on

(cuda-gdb) run
[Launch of CUDA Kernel 0 (applyStencil1D) on Device 0]
Program received signal CUDA_EXCEPTION_1, Lane Illegal Address.
applyStencil1D<<<(32768,1,1),(512,1,1)>>> at stencil1d.cu:60

(cuda-gdb) info line stencil1d.cu:60
out[ i ] += weights[ j + RADIUS ] * in[ i + j ];
increase precision

- Single-stepping
  - Every exception is automatically precise

- The "autostep" command
  - Define a window of instructions where we think the offending load/store occurs
  - `cuda-gdb` will single-step all the instructions within that window automatically and without user intervention

(cuda-gdb) autostep foo.cu:25 for 20 lines

(cuda-gdb) autostep *$pc for 20 instructions
New in CUDA 4.1

- Source base upgraded to \texttt{gdb} 7.2
- Simultaneous \texttt{cuda-gdb} sessions support
- Multiple context support
- Device assertions support
- New “\texttt{autostep}” command

More info:
- \url{http://sbel.wisc.edu/Courses/ME964/2012/Documents/cuda-gdb41.pdf}
Tips & Miscellaneous Notes
1. Determine scope of the bug
   - Incorrect result
   - Unspecified Launch Failure (ULF)
   - Crash
   - Hang
   - Slow execution

2. Reproduce with a debug build
   - Compile your app with –g –G
   - Rerun, hopefully you can reproduce problem in debug model
3. Correctness Issues
   - First throw `cuda-memcheck` at it in stand-alone
   - Then `cuda-gdb` and `cuda-memcheck` if needed
   - `printf` is also an option, but not recommended

4. Performance Issues (once no bugs evident)
   - Use a profiler (already discussed)
   - Change the code, might have to go back to Step 1. above…
Tips

- Always check the return code of the CUDA API routines!

- If you use `printf` from the device code…
  - Make sure to synchronize so that buffers are flushed
Tips

- To hide devices, launch the application with
  \texttt{CUDA\_VISIBLE\_DEVICES=0, 3}
  where the numbers are device indexes.

- To increase determinism, launch the kernels synchronously:
  \texttt{CUDA\_LAUNCH\_BLOCKING=1}
To print multiple consecutive elements in an array, use @:

```
(cuda-gdb) print array[3]@4
```

To find the mangled name of a function:

```
(cuda-gdb) set demangle-style none
(cuda-gdb) info function my_function_name
```
On a Final Note...

- For GPU computing questions, email Big Dan and/or Andrew:
  - melanz@wis.edu or aaseidl@wisc.edu

- If after this short course you end up using GPU computing, I would *love* to hear any success story you might have
  - negrut@wisc.edu

- Other resources:
  - HPC class at Wisconsin: http://sbel.wisc.edu/Courses/ME964/2012/
    - All material available online for download/viewing
Further Information

- **More resources:**
  - CUDA tutorials video/slides at GTC
  - CUDA webinars covering many introductory to advanced topics

- **Other related topic:**
  - Performance Optimization Using the NVIDIA Visual Profiler