CME213/ME339
Lecture 11

Erich Elsen

Department of Mechanical Engineering
Institute for Computational and Mathematical Engineering
Stanford University

Spring 2013

What is Thrust?

- Template Header library similar to the C++ STL but targeted to parallel execution
- Performance portable way to express parallel algorithms
- Take advantage of expertly tuned implementations targeted for each GPU architecture
- Multiple Backends - run code on GPUs with CUDA or on multiple cores with OMP or TBB
- Highly productive way to program with CUDA

First Example

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

int main(void) {
  thrust::device_vector<int> d_data(1000);

  //generate a sequence counting down from 1000
  thrust::sequence(d_data.begin(), d_data.end(), 1000, -1);

  d_data[30] = -6;

  thrust::sort(d_data.begin(), d_data.end());

  thrust::host_vector<int> h_data = d_data;
}
```

Advantages

- Automatic Memory Management with containers (no cudaMalloc / cudaFree)
- Cleaner code for memory movement (no verbose cudaMemcpys)
- Read / Write to specific array locations very useful for debugging
- Invoking an algorithm with a device vector will run it on the device
- Invoking an algorithm with a host vector will run it on the host
- Meaning of "device" and "host" can be changed when compiling
- `-DTHRUST_DEVICE_SYSTEM=THRUST_HOST_SYSTEM_OMP`

STL and Functional Programming

Say we want to double every number in a list. The procedural way is:

```
for (int i = 0; i < N; ++i)
  mylist[i] *= 2;
```

We declare not only what we want done (doubling) but also HOW it is done (with a for loop).

Procedural vs. Functional Programming

Say we want to double every number in a list. The functional way is:

```
//declare a functor
struct doubler {
  void operator()(int& x) {
    x *= 2;
  }
};
std::for_each(mylist.begin(), mylist.end(), doubler());
```

We declare only what we want done (doubling). How this happens is not specified.

Functors

Can have internal state...

```
template<typename T>
struct xer {
  T multiple;
  //constructor to initialize multiple
  xer(const T& m) : multiple(m) {}

  void operator()(T& x) {
    x *= multiple;
  }
};

//declare our functors
xer<float> mf(5.4f);
xer<int>   mi(3);

std::for_each(floatlist.begin(), floatlist.end(), mf);
std::for_each(intlist.begin(), intlist.end(), mi);
```

Iterators

- A powerful concept that allows for applying algorithms such as `for_each` to a variety of containers
  - vectors, lists, sets, maps, etc...
- On the GPU the only container that makes much sense is the vector
- Vector iterators are nice in that they are essentially just pointers
- `.begin()` points to the first element
- `.end()` points to one past the last element
- Algorithms also work with raw pointers

```
int a[10];
std::for_each(a, a + 10, doubler());
```

Iterators and Thrust

To use a device vector in a normal kernel, you need to do:

```
myKernel<<<blocks, threads>>>(
  thrust::raw_pointer_cast(&d_vector[0]));
```

To use a raw device pointer with a thrust algorithm, you need to let thrust know this pointer exists on the device:

```
thrust::device_ptr<float> my_thrust_ptr(my_device_ptr);
thrust::sort(my_thrust_ptr, my_thrust_ptr + N);
```

Now with Thrust

```
template<typename T>
struct xer {
  T multiple;
  //constructor to initialize multiple
  __host__ __device__
  xer(const T& m) : multiple(m) {}

  __host__ __device__
  void operator()(T& x) {
    x *= multiple;
  }
};

//declare our functors
xer<int>   mi(3);

thrust::host_vector<int> h_intlist(10);
thrust::device_vector<int> d_intlist(10);

//runs on the CPU
thrust::for_each(h_intlist.begin(), h_intlist.end(), mi);
//runs on the GPU
thrust::for_each(d_intlist.begin(), d_intlist.end(), mi);
```

Radix Sort with Thrust

Sort this sequence: `0 1 1 0 1 0 1`

- Idea: Each 0 needs to know how many 0s are before it
- Equivalent to knowing the current position and how many 1s before us
- Each 1 needs to know many 1s are before it and also how many 0s
- We need Scan, Transform and Scatter

Scan it: `0 0 1 2 2 3 3 4`
Now each position "knows" how many ones come before it and also how many total ones there are (the last entry)

Radix Sorting with Thrust

Transform

- If we are a 0 at position 5, then we need to know how many 0s are before us, but we know how many 1s are before us.
- That is equal to our position (5) minus the number of 1s before us. This is our output position.
- If we are a 1, our final position is total number of 0s plus the number of 1s before us.

```
    if (digit == 0)
      outputPos = ourPosition - ScanVal;
    else //digit == 1
      outputPos = numZeros + ScanVal;
```

Transform: 0 3 4 1 5 2 6

Radix Sort With Thrust

Scatter

- output[outputLoc[i]] = input[i]
- outputLoc = transformed values
- input = original input sequence

```
Input:      0 1 1 0 1 0 1
OutputLocs: 0 3 4 1 5 2 6
Output:     0 0 0 1 1 1 1
```

Radix Sort With Thrust

- We would perform this process for total number of bits that we need to sort
- Thrust has a better way of sorting that we've seen earlier
- The built-in radix sort is *extremely* fast
- This example is for pedagogical purposes, don't actually write your own sort

```
thrust::sort(input.begin(), input.end());
```

Done.

Kernel Fusion

Image we want to compute $\sum_{i=0}^{N} x_i^2$

Approach 1:

```
struct square {
  __host__ __device__
  float operator()(float x) {
    return x * x;
  }
};

thrust::device_vector<float> input(100);
thrust::device_vector<float> squares(100);

//transform(InputStart, InputEnd, OutputStart, Function);
thrust::transform(input.begin(), input.end(),
                  squares.begin(), square());
float sum = thrust::reduce(squares.begin(), squares.end());
```

- Slow - we write the squares back to memory and then read them in again
- Wastes space - we don't need to store the temporary values

Transform Iterators and Kernel Fusion

Transform Iterators Apply a transform in-place allowing the new value to used locally.

```
struct square {
  __host__ __device__
  float operator()(float x) {
    return x * x;
  }
};

thrust::device_vector<float> input(100);

float sum = thrust::reduce(
               thrust::make_transform_iterator(
                   input.begin(), square()),
               thrust::make_transform_iterator(
                   input.end(), square()));
```

Constant and Counting Iterators

We often need to represent a constant number or an increasing sequence. Explicitly representing them would waste both storage space and memory resources moving them around.

```
//sum = 100
int sum = thrust::reduce(thrust::make_constant_iterator(1),
                         thrust::make_constant_iterator(1) + 100);

//sum = 5050
sum = thrust::reduce(thrust::make_counting_iterator(1),
                     thrust::make_counting_iterator(101));
```

Histogram Example

```
Given the sequence:
   [2 1 0 0 2 2 1 1 1 1 4]
the dense histogram would be:
   [2 5 3 0 1]
the sparse histogram would be:
   [(0,2), (1,5), (2,3), (4,1)]
```

- How might we implement these operations with thrust algorithms?
- The first step - sort!
- Even when a serial algorithm might not involve sorting it is often a useful primitive when using thrust
- Let's examine the sparse case - we can do it with one thrust call!

Segmented Reduction

```
thrust::pair<OutputIterator1,OutputIterator2>
thrust::reduce_by_key(InputIterator1 keys_first,
                      InputIterator1 keys_last,
                      InputIterator2 values_first,
                      OutputIterator1 keys_output,
                      OutputIterator2 values_output);
```

We perform multiple reductions using adjacent identical keys to determine which reductions to perform

```
Keys      : 1 3 3 3 2 2 1
Vals      : 9 8 7 6 5 4 3

Out Keys : 1 3  2 1
Out Vals : 9 21 9 3
```

So how do we use this to do the histogram?

Sparse Histogram

After sorting...

```
Sequence:        0 0 1 1 1 1 1 2 2 2 4 (keys)
Const. Iterator: 1 1 1 1 1 1 1 1 1 1 1 (vals)

Out Keys: 0 1 2 4 E X X X X X X
Out Vals: 2 5 3 1 E X X X X X X
```

```
thrust::device_vector<int> data(11); //= [2 1 0 0 2 2 1 1 1 1 4]
thrust::device_vector<int> histogram_values(11);
thrust::device_vector<int> histogram_counts(11);

thrust::sort(data.begin(), data.end());

typedef thrust::device_vector<int>::iterator devIt;
thrust::pair<devIt, devIt> endIterators =
   thrust::reduce_by_key(data.begin(), data.end(),
                         thrust::make_constant_iterator(1),
                         histogram_values.begin(),
                         histogram_counts.begin());

int num_values = endIterators.first - histogram_values.begin();
```

Sparse Histogram

- What if we don't want to over-allocate space for `histogram_values` and `histogram_counts`?
- Must count how many distinct values in keys
- Can do this with a clever use of `inner_product`
- Inner product: $(a_0 \otimes b_0) \oplus (a_1 \otimes b_1) \oplus \dots$
- Dot product is $\otimes = \times$ and $\oplus = +$

```
Sequence A:         0 0 1 1 1 1 1 2 2 2 4 (keys)
Sequence B:           0 0 1 1 1 1 1 2 2 2 4 (keys)
```

What should our operations be?

Sparse Histogram

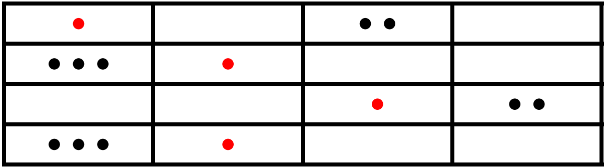
```
int num_bins = thrust::inner_product(data.begin(), data.end() - 1,
                                     data.begin() + 1,
                                     (int)1, thrust::plus<int>(),
                                     thrust::not_equal_to<int>());
```

```
Sequence A:        0 0 1 1 1 1 1 2 2 2 4 (keys)
Sequence B:          0 0 1 1 1 1 1 2 2 2 4 (keys)

A != B     :         0 1 0 0 0 0 1 0 0 1
num_bins   :     1 + 0+1+0+0+0+0+1+0+0+1 = 4
```

Second Example - Point Binning

- First generate a random collection of 2d points
- Then bin these points into a 2d grid of cells
- Finally extract which cells have only one point in them



Random Number Generation

With Thrust

- Can do RNG on both host and device
- We will focus on host generation
- We need a both a generator AND a distribution
- A Generator produces random bits
  - `thrust::default_random_engine`
- The Distribution turns these bits into something useful - uniformly distributed floats between $[-3, 10]$ for example
  - `thrust::uniform_real_distribution`
  - `thrust::uniform_int_distribution`

Random Point Generation

```
// return a random vec2 in [0,1)^2
vec2 make_random_float2(void)
{
  //The static is important!
  static thrust::default_random_engine rng;
  static thrust::uniform_real_distribution<float> u01(0.0f, 1.0f);
  float x = u01(rng);
  float y = u01(rng);
  return vec2(x,y);
}

thrust::host_vector<float2> h_points(N);
thrust::generate(h_points.begin(), h_points.end(), make_random_float2);

thrust::device_vector<float2> points = h_points;
```

Grid Structure

```
// allocate storage for a 2D grid
// of dimensions w x h
unsigned int w = 200, h = 100;

// the grid data structure keeps a range per grid bucket:
// each bucket_begin[i] indexes the first element of
// bucket i's list of points
thrust::device_vector<unsigned int> bucket_begin(w*h);

// each bucket_end[i] indexes one past the last element of
// bucket i's list of points
thrust::device_vector<unsigned int> bucket_end(w*h);

// allocate storage for each point's bucket index
thrust::device_vector<unsigned int> bucket_indices(N);
```

Point to Bucket Functor

```
// hash a point in the unit square to the index of
// the grid bucket that contains it
struct point_to_bucket_index :
          thrust::unary_function<float2,unsigned int>
{
  float width;  // buckets in the x dimension (grid spacing = 1/width)
  float height; // buckets in the y dimension (grid spacing = 1/height)

  __host__ __device__
  point_to_bucket_index(unsigned int width, unsigned int height)
    : width(width), height(height) {}

  __host__ __device__
  unsigned int operator()(const float2& v) const
  {
    // find the raster indices of p's bucket
    unsigned int x = static_cast<unsigned int>(v.x * width);
    unsigned int y = static_cast<unsigned int>(v.y * height);

    // return the bucket's linear index
    return y * width + x;
  }

};
```

Points → Cells

```
// transform the points to their bucket indices
thrust::transform(points.begin(),
                  points.end(),
                  bucket_indices.begin(),
                  point_to_bucket_index(w,h));

// sort the points by their bucket index
thrust::sort_by_key(bucket_indices.begin(),
                    bucket_indices.end(),
                    points.begin());
```

Transform (assume `w=10 h=10` here):

```
Points: (.7,.8) (.4,.1) (.6,.4) (.2,.3) (.62, .43)
Index :    87      14      64      32       64
```

Sort:

```
Index :    14      32      64      64         87
Points: (.4,.1) (.2,.3) (.6,.4) (.62, .43) (.7, .8)
```

Determine Contents of Each Cell

- For each cell `[0, w*h)` we need to figure out its bounds
- Hmmm...makes me think of a `counting_iterator`
- The algorithms we need are `lower_bound` and `upper_bound`
- `lower_bound` takes a sequence and a list of search values
- For each search value, it finds the first place in the sequence it could be inserted without changing the ordering

```
Seq: 0 0 1 2 4 4 5 6 6 6 7
             3     6

lower_bound: 3 6
Output:      4 7
```

Determine Contents of Each Cell

- `upper_bound` is similar except it finds the *last* place in the sequence the value can be inserted without changing the ordering

```
Seq: 0 0 1 2 4 4 5 6 6 6 7
             3           6

upper_bound: 3 6
Output:      4 10
```

Now we can determine the number of points in each cell by subtracting the output of `lower_bound` from `upper_bound`

Code for Cell Determination

- The sequence we're searching *in* comes first
- The values we're searching *for* come next
- The output goes last

```
// find the beginning of each bucket's list of points
thrust::counting_iterator<unsigned int> search_begin(0);

thrust::lower_bound(bucket_indices.begin(),
                    bucket_indices.end(),
                    search_begin,
                    search_begin + w*h,
                    bucket_begin.begin());

// find the end of each bucket's list of points
thrust::upper_bound(bucket_indices.begin(),
                    bucket_indices.end(),
                    search_begin,
                    search_begin + w*h,
                    bucket_end.begin());
```

Extracting Lonely Points

The function we need for this is called `remove_copy_if`

```
Cell:      0 1 2 3 4 5 6 7 8 9
# Points: 0 1 3 2 1 4 0 4 1 0
```

```
Out: 1 4 8
```

We need a predicate to determine which cells to remove based on the value of points

```
struct is_equal_to_one : thrust::unary_function<int, int>
{
    __host__ __device__
    int operator()(const int& v)
    {
        return v == 1;
    }
};
```

Extract Cells

```
thrust::device_vector<int> bucket_sizes(N);
thrust::transform(bucket_end.begin(), bucket_end.end(),
                  bucket_begin.begin(), bucket_sizes.begin(),
                  thrust::minus<int>());

int num_lonely_cells = thrust::count_if(bucket_sizes.begin(),
                                        bucket_sizes.end(),
                                        is_equal_to_one());

thrust::device_vector<int> lonely_cells(num_lonely_cells);
thrust::remove_copy_if(make_counting_iterator(0),
                       make_counting_iterator(w*h),
                       bucket_sizes.begin(),
                       lonely_cells.begin(),
                       is_equal_to_one() );
```

Maximum Number of Points

To determine the cell with the most points, we could use `max_element` which returns an *iterator* to the largest element

```
thrust::device_vector<int>::iterator maxIt;
maxIt = thrust::max_element(bucket_sizes.begin(), bucket_sizes.end());

int maxNum = *maxIt;
int maxPos = maxIt - bucket_sizes.begin();
```

```
# Points: 0 1 3 2 1 4 0 4 1 0
```

```
maxNum: 4
maxPos: 5 (returns the first if there are duplicates)
```

Additional Thrust Information

https://github.com/thrust/thrust/wiki/Quick-Start-Guide

https://github.com/thrust/thrust/wiki/Documentation

http://developer.download.nvidia.com/CUDA/training/GTCthrust.mp4

Vigenère Cipher

```
Plain text:  ILIKEMYTEACHER
KEY:         NOTNOTNOTNOTNO
==========================
Cipher text: VZBXSFLHXNQARF
```

- By using multiple shifts (or permutations) instead of just 1 as in a substitution cipher the frequency distribution of the cipher text is obscured
- If we knew the key length, then we could solve multiple substitution ciphers

```
Plain text:  ILIKEMYTEACHER
KEY:         NOTNOTNOTNOTNO
==========================
Cipher text: VZBXSFLHXNQARF
```

Determine Key Length

First define an Index of Coincidence (IOC) between two texts as:

$$\frac{26 * \sum_{i=0}^{N} A_i == B_i}{N}$$

```
MYTEACHERISAWESOME
ILOVETHRUSTCODING
==================
000000100000000000

IOC = 26 * 1 / 17 = 1.53
```

- Defined so that the IOC between two *randomly* chosen texts is 1
- In English the IOC between two different texts (Moby Dick and The Great Gatsby) is ~ 1.73
- Which is because letters are not chosen randomly, but have a non-uniform frequency distribution

Breaking the Vigenère

Shifts don't line up and the IOC is ~ 1, the texts appear random

```
VZBXSFLHXNQARF
 VZBXSFLHXNQARF
```

```
VZBXSFLHXNQARF
  VZBXSFLHXNQARF
```

Shifts line up and suddenly the IOC jumps to ~ 1.73, because now at each position the "alphabet" though jumbled, is the same

```
VZBXSFLHXNQARF
   VZBXSFLHXNQARF
```