CME213/ME339
Lecture 12

Eric Darve
Erich Elsen
Austin Gibbons

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

Spring 2012

Thrust Algorithms

- Transformations
- Reductions
- Sorts
- Searches
- Set Operations
- Compaction

Algorithm variants

- Many algorithms will have many variants on the same idea
- Often there can be either 1 or 2 input sequences with either an unary or binary function
- *_if variants take a stencil or predicate function and only perform the operation if the stencil or predicate is true
- *_by_key variants perform the operation on multiple smaller sequences
    - with the exception of sort_by_key which is poorly named

Transform Algorithms

transform

```
OutputIterator thrust::transform(InputIterator first,
                                 InputIterator last,
                                 OutputIterator result,
                                 UnaryFunction op);
```

Example use:

```
#include <thrust/transform.h>
#include <thrust/functional.h>

int data[10] = {-5, 0, 2, -3, 2, 4, 0, -1, 2, 8};
thrust::negate<int> op;
thrust::transform(data, data + 10, data, op); // in-place transformation
// data is now {5, 0, -2, 3, -2, -4, 0, 1, -2, -8};
```

Transform Algorithms

`transform`

```
OutputIterator thrust::transform(InputIterator1 first1,
                                 InputIterator1 last1,
                                 InputIterator2 first2,
                                 OutputIterator result,
                                 BinaryFunction op);
```

Example use:

```
#include <thrust/transform.h>
#include <thrust/functional.h>

thrust::host_vector<int> input1(6);
//= {-5, 0, 2, 3, 2, 4}
thrust::host_vector<int> input2(6);
// = { 3,  6, -2,  1,  2,  3};
thrust::host_vector<int> output(6);

thrust::plus<int> op;
thrust::transform(input1.begin(), input1.end(), input2.begin(),
                  output.begin(), op);
// output is now {-2,  6,  0,  4,  4,  7};
```

Transform Algorithms

`transform_if`

```
ForwardIterator thrust::transform_if (InputIterator first,
                    InputIterator last,    ForwardIterator result,
                    UnaryFunction op,      Predicate pred);
```

Example use:

```
#include <thrust/transform.h>
#include <thrust/functional.h>
int data[10]    = {-5, 0, 2, -3, 2, 4, 0, -1, 2, 8};

struct is_odd {
    __host__ __device__
    bool operator()(int x) {
        return x % 2;
    }
};

thrust::negate<int> op;
// negate odd elements; in-place transformation
thrust::transform_if(data, data + 10, data, op, is_odd());

// data is now {5, 0, 2, 3, 2, 4, 0, 1, 2, 8};
```

Transform Algorithms

`transform_if`

```
ForwardIterator thrust::transform_if(InputIterator1 first1,
        InputIterator1 last1,        InputIterator2 first2,
        InputIterator3 stencil,      ForwardIterator result,
        BinaryFunction binary_op,    Predicate pred);
```

Example use:

```
#include <thrust/transform.h>
#include <thrust/functional.h>

int input1[6]  = {-5,  0,  2,  3,  2,  4};
int input2[6]  = { 3,  6, -2,  1,  2,  3};
int stencil[8] = { 1,  0,  1,  0,  1,  0};
int output[6];

thrust::plus<int> op;
thrust::identity<int> identity;

thrust::transform_if(input1, input1 + 6, input2, stencil,
                     output, op, identity);

// output is now {-2,  0,  0,  3,  4,  4};
```

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

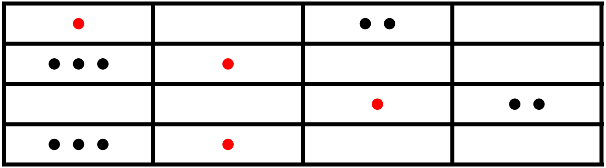
```
int num_bins = thrust::inner_product(data.begin(), data.end() - 1,
                                     data.begin() + 1,
                                     (int)1, thrust::plus<int>(),
                                     thrust::not_equal_to<int>());
```

```
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)
```

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
==================
00000010000000000

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
```