

**CME 213**

**SPRING 2017**

**Eric Darve**

**1891**

# Previous lectures

**How to compute a reduction efficiently on a GPU:**

- **kernel 1:** minimize branching inside a warp
- **kernel 2:** remove bank conflicts
- **kernel 3:** assign work to all threads (first step of reduction)
- **kernel 4:** unroll and remove **syncthreads**
- **kernel 5:** main loop unroll using **#pragma**
- **kernel 6:** local reduction; fixed grid size independent of vector size

**Atomics**

# Thrust

**Writing CUDA code is not that easy!**

- Difficulty of writing code with good performance
- Portability problem: what if my computer does not have an NVIDIA GPU?

**Thrust offers a clever solution to some of these problems!**



# What is Thrust?

**It's the STL (standard template library) of CUDA!**

**It's a C++ template library that does many things:**

- **Implements several standard algorithms like sort, reduction, scan**
- **Allows the user to easily specify "custom" operations if they are not in the library.**
- **Provides a high-performance implementation. All the hard work has been done for you already!**

**That's magic.**

**Works beautifully but requires good knowledge of templates.**

**"Think STL"**

# Multiple backends

Provides multiple back-ends: CUDA, OpenMP, TBB (Threading Building Blocks by Intel), and regular C++.

Compilation option examples:

## OpenMP

```
-Xcompiler -fopenmp \
-DTHRUST_DEVICE_SYSTEM=THRUST_DEVICE_SYSTEM_OMP -lgomp
```

## TBB

```
-DTHRUST_DEVICE_SYSTEM=THRUST_DEVICE_SYSTEM_TBB -ltbb
```

You can compile with nvcc or just g++.

# Backend preprocessing macro options

**THRUST\_DEVICE\_SYSTEM\_OMP**

**THRUST\_DEVICE\_SYSTEM\_TBB**

**THRUST\_DEVICE\_SYSTEM\_CUDA**

**THRUST\_DEVICE\_SYSTEM\_CPP**

# You can use shared memory with Thrust

Yes

No

**Start the presentation to activate live content**

If you see this message in presentation mode, install the add-in or get help at [PollEv.com/app](http://PollEv.com/app)

0

Total Results: 0

# Vectors

- **Data is stored as vectors.**
- **Two types of vectors: host and device.**
- **Device vector lives on the GPU but we can be used like a regular STL vector!**

# Host vector

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

#include <iostream>

int main(void) {
    // H has storage for 4 integers
    thrust::host_vector<int> H(4);

    // initialize individual elements
    H[0] = 14;
    H[1] = 20;
    H[2] = 38;
    H[3] = 46;

    // H.size() returns the size of vector H
    std::cout << "H has size " << H.size() << std::endl;

    // print contents of H
    for(size_t i = 0; i < H.size(); i++) {
        std::cout << "H[" << i << "] = " << H[i] << std::endl;
    }
}
```

# Device vector

- Same syntax.
- Lives on the GPU but no difference for the programmer.

```
// Copy host_vector H to device_vector D
thrust::device_vector<int> D = H;

// elements of D can be modified
D[0] = 99;
D[1] = 88;

// print contents of D
for(size_t i = 0; i < D.size(); i++) {
    std::cout << "D[" << i << "] = " << D[i] << std::endl;
}
```

# Containers

- STL has many containers!  
`vector, list, deque, set, multiset, map, multimap,  
hash_set, hash_multiset, hash_map, hash_multimap`
- Thrust only allows vector.
- This is because this is the only container that fits well the GPU computing model.
- Thrust is still compatible with the STL.

# Static dispatching

Thrust is a clever system. It will figure out based on the type of the arguments whether an algorithm should be run on the host or device.

```
// initialize all ten integers of a device_vector to 1
thrust::device_vector<int> D(10, 1);
```

Runs on the GPU

```
// set the first seven elements of a vector to 9
thrust::fill(D.begin(), D.begin() + 7, 9);
```

```
// initialize a host_vector with the first five elements of D
thrust::host_vector<int> H(D.begin(), D.begin() + 5);
```

```
// set the elements of H to 0, 1, 2, 3, ...
thrust::sequence(H.begin(), H.end()));
```

Runs on the CPU

Decision is based on type: host or device

# Pointer types

- Casting between pointer types is often needed.
- Thrust assumes that a regular pointer **lives on the host**. But this is not always true!

```
{  
    // 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);  
}
```

Without the cast,  
the function would  
run on the host

- This allows using Thrust in the middle of other CUDA code.

# Raw pointers

Similarly, there are cases where Thrust produces some output result that we want to use in CUDA code.

```
{  
    // create a device_ptr  
    thrust::device_ptr<int> dev_ptr = thrust::device_malloc<int>(N);  
  
    // extract raw pointer from device_ptr  
    int* raw_ptr = thrust::raw_pointer_cast(dev_ptr);  
}
```

Convert a device pointer back to a regular pointer.

This means you can freely go back and forth between CUDA and Thrust.

# Let's get to work!

- Let's calculate a saxpy:

$$Y \leftarrow aX + Y$$

- We take as input `(a, x[i], y[i])` and return `a*x[i]+y[i]`
- For this, STL-like transform functions are required.
- What is a transform?

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

Input → Output

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

(Input1,Input2) → Output

# Operators

- How do we define an operator?
- Generally speaking, this is simply an object  $f$  for which we can define  $f(x, y)$ .
- See `saxpy.cu` example.

## What's a functor?

Answer: a class that defines an operator ()

A binary function takes two inputs, and returns one output

```
struct saxpy_functor : public thrust::binary_function<float, float, float> {  
    const float a;  
  
    saxpy_functor(float _a) : a(_a) {}  
  
    __host__ __device__  
    float operator()(const float& x, const float& y) const {  
        return a * x + y;  
    }  
};
```

That's the () operator! The type of the variables must match **binary\_function**

# Applying the transform

```
void saxpy(float A, thrust::device_vector<float>& X,
           thrust::device_vector<float>& Y) {
    // Y <- A * X + Y
    thrust::transform(X.begin(), X.end(), Y.begin(), Y.begin(), saxpy_functor(A));
}
```

Input 1

Input 2

Output

Functor

# Algorithms

- That's good but I already know how to do that.
- Remember that the backend can be GPU or CPU. That should count!
- More importantly, Thrust provides many STL algorithms:
  - Reduce
  - Scan
  - Sort
  - Reorder/copy
  - Search

# Example: sort

- Many sorts possible using the same framework.
- `sort.cu`

# Basic sort

```
std::cout << "sorting integers\n";
{
    thrust::device_vector<int> keys(N);
    initialize(keys);
    print(keys);
    thrust::sort(keys.begin(), keys.end());
    print(keys);
}
```

79 78 62 78 94 40 86 57 40 16 28 54 77 87 93 98  
16 28 40 40 54 57 62 77 78 78 79 86 87 93 94 98

# User-defined

```
std::cout << "\nsorting integers (user-defined comparison)\n";
{
    thrust::device_vector<int> keys(N);
    initialize(keys);
    print(keys);
    thrust::sort(keys.begin(), keys.end(), evens_before_odds());
    print(keys);
}

// user-defined comparison operator that acts like less<int>,
// except even numbers are considered to be smaller than odd numbers
struct evens_before_odds {
    __host__ __device__
    bool operator()(int x, int y) {
        if(x % 2 == y % 2) {
            return x < y;
        } else if(x % 2) {
            return false;
        } else {
            return true;
        }
    }
};

};


```

sorting integers (user-defined comparison)

79	78	62	78	94	40	86	57	40	16	28	54	77	87	93	98
16	28	40	40	54	62	78	78	86	94	98	57	77	79	87	93

# Key-value sorting

```
std::cout << "\nkey-value sorting\n";
{
    thrust::device_vector<int> keys(N);
    thrust::device_vector<int> values(N);
    initialize(keys, values);
    print(keys, values);
    thrust::sort_by_key(keys.begin(), keys.end(), values.begin());
    print(keys, values);
}
```

```
key-value sorting
(79, 0) (78, 1) (62, 2) (78, 3) (94, 4) (40, 5) (86, 6) (57, 7) (4
(16, 9) (28,10) (40, 5) (40, 8) (54,11) (57, 7) (62, 2) (77,12) (7
```

More examples in `sort.cu`

# Kernel fusion or composability

- More general calculations are allowed by chaining kernels together.
- Imagine for example that we want to compute the norm of a vector.
- This involves two steps:
  - Computing the square of each entry
  - Summing up all the values (reduce)
- Each step is supported by Thrust, but there is no function in Thrust that does both.
- Is it still possible to make it work?

# Transform iterators

- A clever trick. How does it work?

- Take a vector:

```
thrust::device_vector<int> input(N);
```

- Here is a simple reduce:

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

- We need to take the square of the entries.

- `input.begin()` is an iterator. It can be dereferenced to get a value

`5 ← (*it)`

- We want an iterator which when dereferenced returns the square

`5*5 ← (*...)`

- This works by writing this as

`5*5 ← square(5) ← 5 ← (*it)`

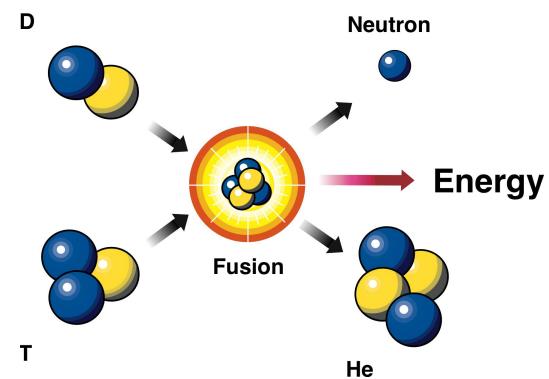
- You have to chain or fuse operations.

Input iterator

Function applied  
to iterator (\*it)

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

Create a new iterator = fusion!



```
/* Define a functor that takes the square of each entry.
 */
struct square : public thrust::unary_function<int,int> {
    __host__ __device__
    int operator()(const int x) const {
        return x * x;
    }
};
```

- A **unary functor** is required
- **transform\_iterator** allows composing kernels together.
- This greatly expands the number of things you can do using Thrust.

## A transform iterator is useful because

It allows to write  
more concise  
code

It is more efficient  
because it reduces  
the number of  
kernel calls

**Start the presentation to activate live content**

If you see this message in presentation mode, install the add-in or get help at [PollEv.com/app](https://PollEv.com/app)

0

Total Results: 0

# Histogram

- Thrust can be used in creative ways to do a lot of things.
- Say for example that you want to calculate a histogram.
- Input data:

3 4 3 5 8 5 6 6 4 4 5 3 2 5 6 3 1 3 2 3 6 5 3 3 3 2 4 2 3 3 2 5 5 5 8 2 5 6 6 3

- Bins

0 1 2 3 4 5 6 7 8

- Count

0 1 6 12 4 9 6 0 2

# Implementation

- Let's break it down step by step.
- First, sort!

```
// sort data to bring equal elements together
thrust::sort(data.begin(), data.end());
```

initial data	3	4	3	5	8	5	6	6	4	4	5	3	2	5	6	3	1	3	2	3	6	5	3	3	3	2	4	2	3	3	2	5	5	5	8	2	5	6	6	3	
sorted data	1	2	2	2	2	2	2	3	3	3	3	3	3	3	3	3	3	3	3	3	4	4	4	5	5	5	5	5	5	5	5	5	5	5	5	6	6	6	6	8	8

initial data	3	4	3	5	8	5	6	6	4	4	5	3	2	5	6	3	1	3	2	3	6	5	3	3	3	2	4	2	3	3	2	5	5	5	5	8	2	5	6	6	3
sorted data	1	2	2	2	2	2	2	3	3	3	3	3	3	3	3	3	3	3	3	3	4	4	4	4	5	5	5	5	5	5	5	6	6	6	6	6	6	8	8		

That's how many bins we need

We need to count how many repeats we have in this sequence.



# Use `upper_bound`

- We are getting deeper into Thrust voodoo territory.
- `upper_bound` searches the position where a value can be inserted without violating the ordering.

Input	1	2	2	2	2	2	2	3	3	3	3	3	3	3	3	3	3	3	3	4	...
Position	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	...
Values								2											3		

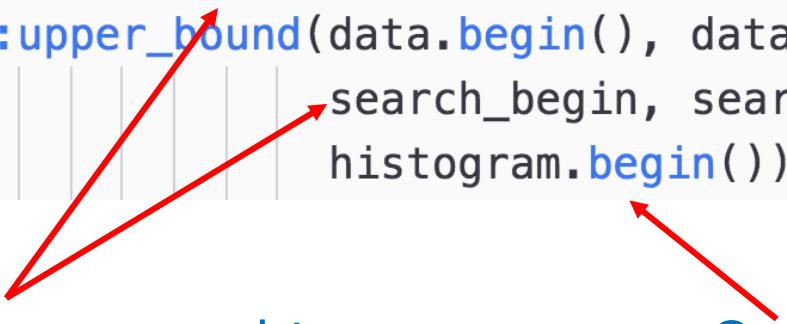
Upper bound of 2 is: 7

Upper bound of 3 is: 19

# Cumulative sum

**upper\_bound** basically gives a cumulative sum of the histogram we are looking for.

```
// find the end of each bin of values
thrust::counting_iterator<IndexType> search_begin(0);
thrust::upper_bound(data.begin(), data.end(),
                    search_begin, search_begin + num_bins,
                    histogram.begin());
```



Counts from 0 to num\_bins

0 1 2 3 4 5 6 7 8

Output goes there

Dense Histogram

	initial data	sorted data	cumulative histogram
initial data	3 4 3 5 8 5 6 6 4 4 5 3 2 5 6 3 1 3 2 3 6 5 3 3 3 2 4 2 3 3 2 5 5 8 2 5 6 6 3	1 2 2 2 2 2 2 3 3 3 3 3 3 3 3 3 3 3 3 4 4 4 4 5 5 5 5 5 5 5 6 6 6 6 8 8	0 1 7 19 23 32 38 38 40

# Dense histogram

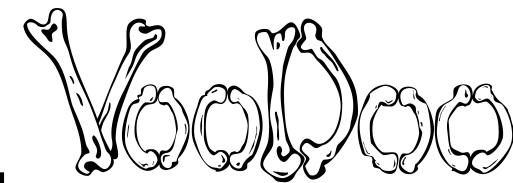
Take differences to get the histogram

```
// compute the histogram by taking differences of the cumulative histogram
thrust::adjacent_difference(histogram.begin(), histogram.end(),
                           histogram.begin());
```

```
Dense Histogram
  initial data  3 4 3 5 8 5 6 6 4 4 5 3 2 5 6 3 1 3 2 3 6 5 3 3 3 2 4 2 3 3 2 5 5 5 8 2 5 6 6 3
  sorted data   1 2 2 2 2 2 2 3 3 3 3 3 3 3 3 3 3 3 3 3 3 4 4 4 4 5 5 5 5 5 5 5 6 6 6 6 8 8
cumulative histogram 0 1 7 19 23 32 38 38 40
  histogram values 0 1 2 3 4 5 6 7 8
  histogram counts 0 1 6 12 4 9 6 0 2
```

# Sparse histogram

- Actually simpler.
- Two steps:
  1. Count how many unique bins are required (histogram bins for which we have at least one value)
  2. Calculate the number of values in each bin



# Count the number of bins needed

- Clever use of `inner_product`
- Voodoo<sup>2</sup>

```
initial data  3 4 3 5 8 5 6 6 4 4 5 3 2 5 6 3 1 3 2 3 6 5 3 3 3 2 4 2 3 3 2 5 5 5 8 2 5 6 6 3  
sorted data   1 2 2 2 2 2 2 3 3 3 3 3 3 3 3 3 3 3 3 4 4 4 4 4 5 5 5 5 5 5 5 5 5 5 5 5 5 5 5 5 5 6 6 6 6 8 8  
shifted data  2 2 2 2 2 2 2 3 3 3 3 3 3 3 3 3 3 3 3 4 4 4 4 5 5 5 5 5 5 5 5 5 5 5 5 5 5 5 5 5 6 6 6 6 8 8  
not equal data 1 0 0 0 0 0 1 0 0 0 0 0 0 0 0 0 0 0 0 1 0 0 0 1 0 0 0 0 0 0 0 0 0 1 0 0 0 0 0 0 1 0 0 0 0 0 1 0 0 0 0 0 1 0
```

This is what we are after!

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

Starting value

The addition

The multiplication!

# One call to Thrust is enough!

1 1 1 1 1 ...

```
thrust::reduce_by_key(data.begin(), data.end(),
                      thrust::constant_iterator<int>(1),
                      histogram_values.begin(),
                      histogram_counts.begin());
```

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

For each of group of consecutive keys that are equal, perform a reduction. Write the key to **keys\_output**, and the result of the reduction to **values\_output**.

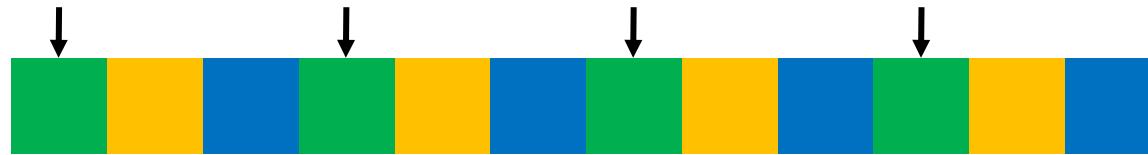
No 7 because it is a sparse histogram

initial data	3 4 3 5 8 5 6 6 4 4 5 3 2 5 0 3 1 3 2 3 6 5 3 3 3 2 4 2 3 3 2 5 5 5 8 2 5 6 6 3
sorted data	1 2 2 2 2 2 2 3 3 3 3 3 3 3 3 3 3 3 3 3 3 4 4 4 4 4 5 5 5 5 5 5 5 5 6 6 6 6 6 8 8
shifted data	2 2 2 2 2 2 3 3 3 3 3 3 3 3 3 3 3 3 3 3 4 4 4 4 4 5 5 5 5 5 5 5 6 6 6 6 6 6 8 8
not equal data	1 0 0 0 0 0 1 0 0 0 0 0 0 0 0 0 0 0 0 1 0 0 0 1 0 0 0 0 0 0 0 1 0 0 0 0 0 0 1 0 0 0 0 0 1 0
histogram values	1 2 3 4 5 6 8
histogram counts	1 6 12 4 9 6 2

# Zippers

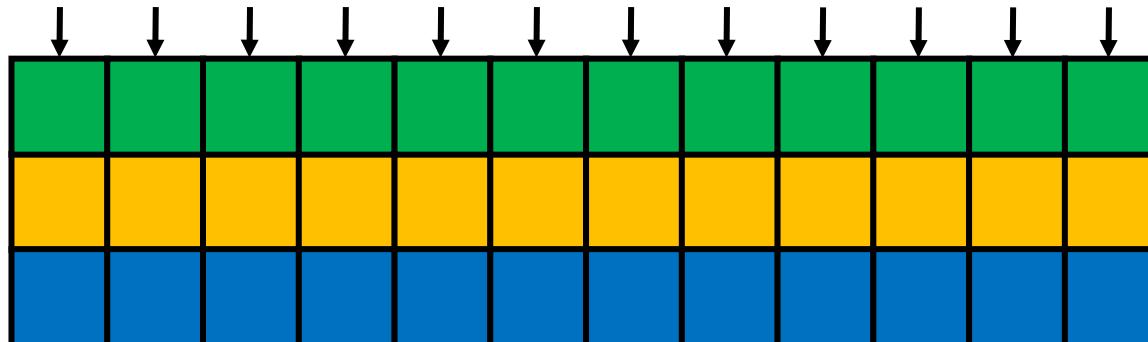


- There are many cases where a function expects one argument but we need many.
- What should we do then?
- This can be quite common in fact.
- Usually, for convenience, we define structs and put all our variables in them.
- This is the **Array of Structs** approach.
- However, on a GPU, this leads to uncoalesced memory access.



Stride 3

- It is better to use a **Struct of Arrays**.



Stride 1

# Zippers are the way to go

- Say we have two arrays of 3D vectors.
- We want to calculate the dot product of all these vectors.
- $(A_0, A_1, A_2)$  is array A of 3D vectors.
- $(B_0, B_1, B_2)$  is array B.

```
thrust::device_vector<float> A0 =
| random_vector(N, rng); // x components of the 'A' vectors
thrust::device_vector<float> A1 =
| random_vector(N, rng); // y components of the 'A' vectors
thrust::device_vector<float> A2 =
| random_vector(N, rng); // z components of the 'A' vectors

thrust::device_vector<float> B0 =
| random_vector(N, rng); // x components of the 'B' vectors
thrust::device_vector<float> B1 =
| random_vector(N, rng); // y components of the 'B' vectors
thrust::device_vector<float> B2 =
| random_vector(N, rng); // z components of the 'B' vectors
```

# Zip it!

**Zip it to create a single iterator for A, and another one for B.**

```
// Now we create some zip_iterators for A and B
Float3Iterator A_first = thrust::make_zip_iterator(
    make_tuple(A0.begin(), A1.begin(), A2.begin()));
Float3Iterator A_last = thrust::make_zip_iterator(
    make_tuple(A0.end(), A1.end(), A2.end()));
Float3Iterator B_first = thrust::make_zip_iterator(
    make_tuple(B0.begin(), B1.begin(), B2.begin()));
```

# Then run the kernel

The kernel is run as usual using the zip iterators.

```
// Finally, we pass the zip_iterators into transform() as if they
// were 'normal' iterators for a device_vector<Float3>.
thrust::transform(A_first, A_last, B_first, result.begin(), DotProduct());
```

```
// We use a 3-tuple to store the 3 components of our vector
typedef thrust::tuple<float, float, float> Float3;
```

```
// This functor implements the dot product between 3d vectors
struct DotProduct : public thrust::binary_function<Float3, Float3, float> {
    __host__ __device__
    float operator()(const Float3& a, const Float3& b) const {
        return thrust::get<0>(a) * thrust::get<0>(b) +      // x components
               thrust::get<1>(a) * thrust::get<1>(b) +      // y components
               thrust::get<2>(a) * thrust::get<2>(b);      // z components
    }
};
```

lead to better  
memory access  
and efficiency

are similar to  
arrays of  
structures

**Start the presentation to activate live content**

If you see this message in presentation mode, install the add-in or get help at [PollEv.com/app](https://PollEv.com/app)

0

Total Results: 0

# We have only scratched the surface

- <https://thrust.github.io> Thrust homepage
- <https://github.com/thrust/thrust/wiki/Quick-Start-Guide>
- The complete documentation
- <https://github.com/thrust/thrust/wiki/Documentation>
- Use Google to search
- [Check the code examples to get ideas](#)
- Nothing on TV tonight? Check out some videos on Thrust:
  - [GTC 2011 video on Thrust by Nathan Bell](#)
  - [Introduction to Thrust slides by Hoberock and Bell](#)
  - [GTC 2011 video on Thrust by Hoberock and Bell](#)