Thrust

CUDA Course

July 21-25

István Reguly

UNIVERSITY OF OXFORD
OXFORD e-Research CENTRE

Thrust

- Open High-Level Parallel Algorithms Library
- Parallel Analog of the C++ Standard Template Library (STL)
  - Vector containers
  - Algorithms
- Comes with the toolkit
- Productive way to use CUDA

Example

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

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
  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;
}
```

Productivity

- Containers
  - host_vector
  - device_vector
- Memory management
  - Allocation, deallocation
  - Transfers
- Algorithm selection
  - Location is implicit

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

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

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

// read device values from the host
int sum = d_vec[0] + d_vec[1];
// invoke algorithm on device
thrust::sort(d_vec.begin(), d_vec.end());
```

Productivity

- Large set of algorithms
  - ~100 functions
  - CPU, GPU
- Flexible
  - C++ templates
  - User-defined types
  - User-defined operators

Algorithm	Description
reduce	Sum of a sequence
find	First position of a value in a sequence
mismatch	First position where two sequences differ
count	Number of instances of a value
inner_product	Dot product of two sequences
merge	Merge two sorted sequences

Portability

- Implementations
  - CUDA C/C++
  - Threading Building Blocks
  - OpenMP
  - Interoperable with anything CUDA based
- Recompile

```
nvcc -DTHRUST_DEVICE_SYSTEM=THRUST_HOST_SYSTEM_OMP
```

- Mix backends

```
thrust::omp::vector<float> my_omp_vec(100);
thrust::cuda::vector<float> my_cuda_vec(100);
```

Interoperability

- Thrust containers and raw pointers
  - Use container in CUDA kernel

```
thrust::device_vector<int> d_vec(...);
cuda_kernel<<<N, 128>>>(some_argument_d,
                        thrust::raw_pointer_cast(&d_vec[0]));
```

  - Use a device pointer in thrust algorithms (not a vector though, no begin(), end(), resize() etc.)

```
int *dev_ptr;
cudaMalloc((void**)&dev_ptr, 100*sizeof(int));

thrust::device_ptr<int> dev_ptr_thrust(dev_ptr);
thrust::fill(dev_ptr_thrust, dev_ptr_thrust+100, 0);
```

Thrust

- Constantly evolving
- Reliable – comes with the toolkit, tested every day with unit tests
- Performance – specialised implementations for different hardware
- Extensible – allocators, backends, etc.

Rapid development

Thrust Implementation

```
thrust::
thrust::
thrust::
thrust::
```

Profiling

```
thrust::
thrust::
thrust::
thrust::
```

Specialize Components

```
thrust::
thrust::
kern<<<
```

Rapid development

- Thrust helps you think about your algorithm as a collection of parallel operations
  - But not worry about “trivial” implementation details
- Gives you a good idea of the complexity of the algorithms involved, it’s a good mental exercise to think through different possible ways of doing the same thing

Simple examples

- Initialisation

```
thrust::fill(array.begin(), array.end(), 7); //7,7,7,7,...
thrust::sequence(array.begin(), array.end());//0,1,2,3,...
```

- Reduction

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

- Transformations

```
// initialize X to 0,1,2,3, ....
thrust::sequence(X.begin(), X.end());
// compute Y = -X
thrust::transform(X.begin(), X.end(), Y.begin(), thrust::negate<int>());
// fill Z with twos
thrust::fill(Z.begin(), Z.end(), 2);
// compute Y = X mod 2
thrust::transform(X.begin(), X.end(), Z.begin(), Y.begin(), thrust::modulus<int>());
// replace all the ones in Y with tens
thrust::replace(Y.begin(), Y.end(), 1, 10);
// print Y
thrust::copy(Y.begin(), Y.end(), std::ostream_iterator<int>(std::cout, "\n"));
```

Iterators

- Generalisation of pointers:
  - Objects that give access to other objects
  - Often used to iterate over a range of objects
    - Incremented, it points to the next object
- Interface between containers and algorithms
  - Algorithms take iterators as arguments, they provide access to elements of a container

```
thrust::copy (InputIterator first, InputIterator last,
              OutputIterator result)
```

Iterators

```
template<class InputIterator, class OutputIterator>
OutputIterator copy (InputIterator first,
                     InputIterator last, OutputIterator result)
{
  while (first!=last) {
    *result = *first;
    ++result; ++first;
  }
  return result;
}
```

Iterators

```
template<class InputIterator>
InputIterator::type reduce(InputIterator first,
                           InputIterator last)
{
  InputIterator::type sum = 0;
  while (first!=last) {
    sum += *first;
  ++first;
  }
  return sum;
}
```

Iterators

- .begin(), .end()
- constant_iterator<T>(T value)
  - always returns value
- counting_iterator<T>(T init)
  - the ith element returns init+i
- permutation_iterator<T,Idx>(values, indices)
  - the ith element returns values[indices[i]]
- transform_iterator<..>(values, function())
  - Applies function() to the elements

Best practices

- Many kernels (especially small ones) are limited by memory bandwidth
- Fusion
  - Combining operations, less memory movement
- Data layout
  - Structure of arrays – memory coalescing
- Implicit sequences
  - Save on memory accesses and storage

Fusion

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

float sum_of_squares(const thrust::device_vector<float> &x) {
  size_t N = x.size();
  thrust::device_vector<float> x_squared(N); // Temporary storage: N elements.

  // Compute x^2: N reads + N writes.
I. thrust::transform(x.begin(), x.end(), x_squared.begin(), square());

  // Compute the sum of x^2s: N + k reads + k+1 writes (k is a small constant).
II. return thrust::reduce(x_squared.begin(), x_squared.end());
}
```

I. – Compute square, read and write the whole array
II. – Do reduction, read whole array, write small partial sums array

Fusion

```
float fused_sum_of_squares(const thrust::device_vector<float> &x) {
  // Compute the x^2s and their sum:
  //  N + k reads + k+1 writes (k is a small constant).
  return thrust::reduce(
    thrust::make_transform_iterator(x.begin(), square()),
    thrust::make_transform_iterator(x.end(), square()));
}
```

- Eliminated temporary storage, N reads, N writes
- Makes use of iterators

Scan

- Parallel prefix sum
- One of the most versatile primitives
  - exclusive scan
  - inclusive scan
  - segmented inclusive/exclusive scan
- In-place operation
- Compact, sort, string matching, tree operations, recurrences, histograms, etc.

Scan - compaction

```
input data    [3   2   1   2   2   7   6   9   7   8   …
needed flags  [0   1   1   0   0   1   0   1   0   1   …
```

Exclusive scan the needed flags array:

```
scan flags    [0   0   1   2   2   2   3   3   4   4   …
```

Gives the indices where to put them

```
thrust::exclusive_scan(flags.begin(), flags.end(), indices.begin());
thrust::copy_if(input.begin(), input.end(),
                thrust::make_permutation_iterator(output.begin(), indices.begin()),
                flags.begin(), _1 == 1);
```

Functor

Exercise

- Given N elements (vertices) we compute a group index for each (aggregation) in the range of 0...N

```
vertex idx    0   1   2   3   4   5   6   7   8   9    …
group idx    [0   2   2   6   4   0   6   4   0   9     …
```

- Come up with 0...M group indices for each, M is the number of groups

Example

- Processing rainfall data
  - Which day – where – how much
  - Sorted by day
  - No rain (measurement of 0) excluded

```
day           [0   0   1   2   5   5   6   6   7   8   …
site          [2   3   0   1   1   2   0   1   2   1    …
measurement   [9   5   6   3   3   8   2   6   5   10  …
```

Storage options

- Array of structures

```
struct Sample
{
  int day;
  int site;
  int measurement;
};
thrust::device_vector<Sample> data;
```

- Structure of arrays (Best Practice)

```
struct Data
{
  thrust::device_vector<int> day;
  thrust::device_vector<int> site;
  thrust::device_vector<int> measurement;
};
Data data;
```

Number of days with any rainfall

Easy way: the unique function, but that deletes duplicates…

```
int days = thrust::unique(data.day.begin(), data.day.end()) -
                                data.day.begin();
```

```
inner_product(x,y) = x[0]*y[0]+x[1]*y[1]+x[2]*y[2]+…
```

```
thrust::inner_product(data.day.begin(), data.day.end()-1,
                      data.day.begin()+1,
                      1,                      //initial value
                      thrust::plus<int>(), //+functor
                      thrust::not_equal_to<int>()); //*functor
```

day [0 0 1 2 5 5 6 6 7 8 …

= ≠ ≠ ≠ = ≠ = ≠ ≠

day [0 0 1 2 5 5 6 6 7 8 …

1 + [0 + 1 + 1 + 1 + 0 + 1 + 0 + 1 + 1 …

Total rainfall at each site

```
template <typename Vector>
void compute_total_rainfall_per_site(const Data &data, Vector &site,
                                     Vector &measurement) {
  // Copy data to keep the original data as it is.
  Vector  tmp_site(data.site), tmp_measurement(data.measurement);
  // Sort the "pairs" (site, measurement) by value of site.
  thrust::sort_by_key(tmp_site.begin(), tmp_site.end(),
                      tmp_measurement.begin());
  // Reduce measurements by
  thrust::reduce_by_key(tmp_site.begin(), tmp_site.end(),
                        tmp_measurement.begin(), site.begin(),
                        measurement.begin());
}
```

```
tmp_site        [0 = 0   1 = 1 = 1 = 1   2 = 2 = 2   3    …
tmp_measurement [6 + 2   3 + 3 + 6 + 10  9 + 8 + 5   5    …

    site        [0   1   2   3   …
    measurement [8   22  22  5   …
```

Number of days where rainfall exceeded 5

```
using namespace thrust::placeholders;
int count_days_where_rainfall_exceeded_5(const Data &data) {
  size_t N = compute_number_of_days_with_rainfall(data);

  thrust::device_vector<int> day(N);
  thrust::device_vector<int> measurement(N);

  thrust::reduce_by_key( data.day.begin(), data.day.end(),
                         data.measurement.begin(), day.begin(),
                         measurement.begin());

  return thrust::count_if(measurement.begin(),
                          measurement.end(), _1 > 5);
}
```

```
struct greater_than {
int threshold;
greater_than( int threshold ) : threshold( threshold ) {}
__device__ __host__ bool operator()( int i ) { return i > threshold; }
};
```

First day where total rainfall exceeded 32

```
int find_first_day_where_total_rainfall_exceeded_32(const Data &data) {
  // Allocate memory to store the prefix sums of measurement.
  thrust::device_vector<int> sums(data.measurement.size());

  // Compute prefix sums.
  thrust::inclusive_scan(data.measurement.begin(),
                         data.measurement.end(), sums.begin());

  // Find the 1st day using a binary search
  int day = thrust::lower_bound(sums.begin(), sums.end(), 33) -
                                sums.begin();
  // Get the day.
  return data.day[day];
}
```

day	[0	0	1	2	5	5	6	6	7	8	…
measurement	[9	5	6	3	3	8	2	6	5	10	…
sums	[9	14	20	23	26	34	36	42	47	57	…

Sort Unsorted Input

day	[0	5	1	6	5	7	2	0	8	6	…
site	[2	2	0	0	1	2	1	3	1	1	…
measurement	[9	8	6	2	3	5	3	5	10	6	…

Sort by day and site

day	[0	0	1	2	5	5	6	6	7	8	…
site	[2	3	0	1	1	2	0	1	2	1	…
measurement	[9	5	6	3	3	8	2	6	5	10	…

Sort Unsorted Input

```
struct day_site_cmp
 {
  template <typename Tuple0, typename Tuple1>
  __device__ __host__ bool operator()(const Tuple0 &t0,
                                      const Tuple1 &t1) {
     int day0  = thrust::get<0>(t0);
     int day1  = thrust::get<0>(t1);
     int site0 = thrust::get<1>(t0);
     int site1 = thrust::get<1>(t1);
     return day0 < day1 || (day0 == day1 && site0 < site1); }
};
void sort_data(Data &data)
{
  thrust::sort_by_key(
    thrust::make_zip_iterator(thrust::make_tuple(data.day.begin(),
                                                 data.site.begin())),
    thrust::make_zip_iterator(thrust::make_tuple(data.day.end(),
                                                 data.site.end())),
    data.measurements.begin(),
    day_site_cmp());
}
```

Sort unsorted input (Faster)

```
void sort_data(Data &data)
{
  thrust::device_vector<int64> tmp(data.day.size());
  // Pack (day, site) pairs into 64-bit integers.
  thrust::transform(
    thrust::make_zip_iterator(thrust::make_tuple(data.day.begin(),
                                                 data.site.begin())),
    thrust::make_zip_iterator(thrust::make_tuple(data.day.end(),
                                                 data.site.end())),
    tmp.begin(),
    pack());

  // Sort using the 64-bit integers as keys.
  thrust::sort_by_key(tmp.begin(),tmp.end(),data.measurement.begin());
  // Unpack (day, site) pairs from 64-bit integers.
  thrust::transform(tmp.begin(),tmp.end(),
  thrust::make_zip_iterator(thrust::make_tuple(data.day.begin(),
                                               data.site.begin())),
                                               unpack());
}
```