The Thrust library

Will Landau

Iowa State University

November 11, 2013

Outline

Outline

Background

- Thrust is the CUDA analog of the Standard Template Library (STL) of C++. It comes with any installation of CUDA 4.2 and above and features:
  - Dynamic data structures
  - An encapsulation of GPU/CPU communication, memory management, and other low-level tasks.
  - High-performance GPU-accelerated algorithms such as sorting and reduction
- Brief history:
  - Emerged from Komrade (deprecated) in 2009
  - Maintained primarily by Jared Hoberock and Nathan Bell of NVIDIA.

vector1.cu

```
1  #include <thrust/host_vector.h>
2  #include <thrust/device_vector.h>
3  #include <iostream>
4
5  int main(void){
6
7    // H has storage for 4 integers
8    thrust::host_vector<int> H(4);
9
10   // initialize individual elements
11   H[0] = 14;
12   H[1] = 20;
13   H[2] = 38;
14   H[3] = 46;
15
16   // H.size() returns the size of vector H
17   std::cout << "H has size " << H.size() << std::endl;
18
19   // print contents of H
20   for(int i = 0; i < H.size(); i++)
21     std::cout << "H[" << i << "] = " << H[i] << std::endl;
22
23   // resize H
24   H.resize(2);
25   std::cout << "H now has size " << H.size() << std::endl;
26
27   // Copy host_vector H to device_vector D
28   thrust::device_vector<int> D = H;
```

vector1.cu

```
29    // elements of D can be modified
30    D[0] = 99;
31    D[1] = 88;
32
33    // print contents of D
34    for(int i = 0; i < D.size(); i++)
35      std::cout << "D[" << i << "] = " << D[i] << std::endl;
36
37    // H and D are automatically deleted when the function returns
38    return 0;
39  }
```

```
1  > nvcc vector1.cu -o vector1
2  > ./vector1
3  H has size 4
4  H[0] = 14
5  H[1] = 20
6  H[2] = 38
7  H[3] = 46
8  H now has size 2
9  D[0] = 99
10 D[1] = 88
```

Notes

- Thrust takes care of `malloc()`, `cudaMalloc()`, `free()`, and `cudaFree()` for you without sacrificing performance.
- The "=" operator does a cudaMemcpy() if one vector is on the host and one is on the device.
- `thrust::` and `std::` clarify the *namespace* of the function after the double colon. For example, we need to distinguish between `thrust::copy()` and `std::copy()`.
- The "<<" operator sends a value to an output stream, the C++ alternative to `printf()`.

vector2.cu

```
1  #include <thrust/host_vector.h>
2  #include <thrust/device_vector.h>
3  #include <thrust/copy.h>
4  #include <thrust/fill.h>
5  #include <thrust/sequence.h>
6  #include <iostream>
7
8  int main(void){
9    // initialize all ten integers of a device_vector to 1
10   thrust::device_vector <int> D(10, 1);
11
12   // set the first seven elements of a vector to 9
13   thrust::fill(D.begin(), D.begin() + 7, 9);
14
15   // initialize a host_vector with the first five elements of D
16   thrust::host_vector<int> H(D.begin(), D.begin() + 5);
17
18   // set the elements of H to 0, 1, 2, 3, ...
19   thrust::sequence(H.begin(), H.end());
20
21   // copy all of H back to the beginning of D
22   thrust::copy(H.begin(), H.end(), D.begin());
23
24   // print D
25   for(int i = 0; i < D.size(); i++)
26     std::cout << "D[" << i << "] = " << D[i] << std::endl;
27
28   return 0;
29 }
```

vector2.cu

```
30 [landau@impact1 vector2]$ make
31 nvcc vector2.cu -o vector2
32 [landau@impact1 vector2]$ ./vector2
33 D[0] = 0
34 D[1] = 1
35 D[2] = 2
36 D[3] = 3
37 D[4] = 4
38 D[5] = 9
39 D[6] = 9
40 D[7] = 1
41 D[8] = 1
42 D[9] = 1
43 [landau@impact1 vector2]$
```

Assignment

- `thrust::copy()` copies a section of one vector into a section of another.
- `thrust::fill()` sets a range of elements to some fixed value.
- `thrust::sequence()` assigns equally-spaced values to a section of a vector.

The vector template classes

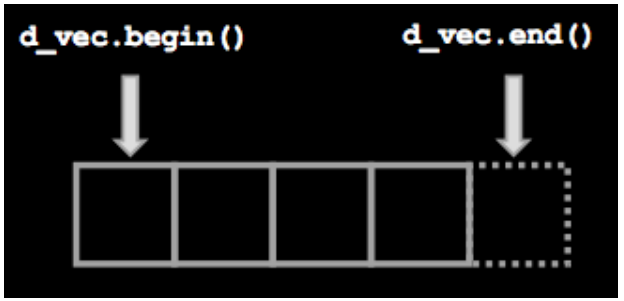
- Declaring vectors:
  - `thrust::device_vector<T> D;` creates a vector `D` with entries of data type `T` on the device.
  - The analogous declaration for host vectors is `thrust::host_vector<T> H;`.
- An object `D` of the vector template class includes the following features:
  - A dynamic linear array of elements of type `T`.
  - Two *iterators*:
    - `D.begin()`
    - `D.end()`

Outline

Basic iterators

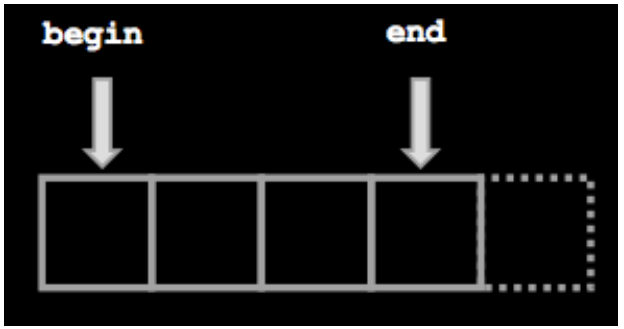
- An iterator is a pointer with a C++ wrapper around it. The wrapper contains additional information, such as whether the vector is stored on the host or the device.

```
1 // allocate device vector
2 thrust::device_vector<int> d_vec(4);
3
4 d_vec.begin(); // returns iterator at first element of d_vec
5 d_vec.end(); // returns iterator one past the last element of d_vec
6
7 // [begin, end) pair defines a sequence of 4 elements
```



Iterators act like pointers.

```
1 // allocate device vector
2 thrust::device_vector<int> d_vec(4);
3
4 thrust::device_vector<int>::iterator begin = d_vec.begin();
5 thrust::device_vector<int>::iterator end = d_vec.end();
6
7 int length = end - begin; // compute the length of the vector
8
9 end = d_vec.begin() + 3; // define a sequence of 3 elements
```



Using iterators

```
1  // allocate device vector
2  thrust::device_vector<int> d_vec(4);
3
4  thrust::device_vector<int>::iterator begin = d_vec.begin();
5
6  *begin = 13; // same as d_vec[0] = 13;
7  int temp = *begin; // same as temp = d_vec[0];
8
9  begin++; // advance iterator one position
10
11 *begin = 25; // same as d_vec[1] = 25;
```

Wrap pointers to make iterators.

```
1  int N = 10;
2
3  // raw pointer to device memory
4  int * raw_ptr;
5  cudaMalloc((void **) &raw_ptr, N * sizeof(int));
6
7  // wrap raw pointer with a device_ptr
8  thrust::device_ptr<int> dev_iter(raw_ptr); // dev_iter is now an
       iterator pointing to device memory
9  thrust::fill(dev_iter, dev_iter + N, (int) 0); // access device memory
       through device_ptr
10
11 dev_iter[0] = 1;
12
13 // free memory
14 cudaFree(raw_ptr);
```

Unwrap iterators to extract pointers.

```
1 // allocate device vector
2 thrust::device_vector<int> d_vec(4);
3
4 // obtain raw pointer to device vectors memory
5 int * ptr = thrust::raw_pointer_cast(&d_vec[0]); // use ptr in a CUDA C
      kernel
6 my_kernel<<<N/256, 256>>>(N, ptr);
7
8 // Note: ptr cannot be dereferenced on the host!
```

constant_iterator

- A constant_iterator is a pointer with some constant value associated with it.

```
#include <thrust/iterator/constant_iterator.h>
...
// create iterators
thrust::constant_iterator<int> first(10);
thrust::constant_iterator<int> last = first + 3;

first[0]; // returns 10
first[1]; // returns 10
first[100]; // returns 10

// sum of [first, last)
thrust::reduce(first, last); // returns 30 (i.e. 3 * 10)
```

counting_iterator

- A counting_iterator is a pointer with the value some_constant + offset associated with it.

```
1  #include <thrust/iterator/counting_iterator.h>
2  ...
3
4  // create iterators
5  thrust::counting_iterator<int> first(10);
6  thrust::counting_iterator<int> last = first + 3;
7
8  first[0]; // returns 10
9  first[1]; // returns 11
10 first[100]; // returns 110
11
12 // sum of [first, last)
13 thrust::reduce(first, last); // returns 33 (i.e. 10 + 11 + 12)
```

transform_iterator

- A transform_iterator is a pointer with the value some_function(vector_entry) associated with it.

```
#include <thrust/iterator/transform_iterator.h>
...
thrust::device_vector <int> vec(3);
vec[0] = 10;
vec[1] = 20;
vec[2] = 30;

// create iterator
thrust::transform_iterator <int> first =
  thrust::make_transform_iterator(vec.begin(), negate<int>());

thrust::transform_iterator <int> last =
  thrust::make_transform_iterator(vec.end(), negate<int>());

first[0] // returns -10
first[1] // returns -20
first[2] // returns -30

thrust::reduce(first, last); // returns -60 (i.e. -10 + -20 + -30)

//same thing:
thrust::reduce(
  thrust::make_transform_iterator(
    vec.begin(), negate<int>()),
  thrust::make_transform_iterator(
    vec.end(), negate<int>()));
```

permutation_iterator

- A permutation_iterator is a pointer associated with a permuted vector.

```
#include <thrust/iterator/permutation_iterator.h>
...
thrust::device_vector <int> map(4);
map[0] = 3;
map[1] = 1;
map[2] = 0;
map[3] = 5;

thrust::device_vector <int> source(6);
source[0] = 10;
source[1] = 20;
source[2] = 30;
source[3] = 40;
source[4] = 50;
source[5] = 60;

typedef thrust::device_vector<int>::iterator indexIter;

thrust::permutation_iterator<indexIter, indexIter> pbegin =
  thrust::make_permutation_iterator(
    source.begin(), map.begin());

thrust::permutation_iterator<indexIter, indexIter> pend =
  thrust::make_permutation_iterator(
    source.end(), map.end());
```

permutation_iterator

```
26 int p0 = pbegin[0]; // source[map[0]] = 40
27 int p1 = pbegin[1]; // source[map[1]] = 20
28
29 int sum = thrust::reduce(pbegin, pend);
30
31 /* sum =
32  * source[map[0]] + source[map[1]] + source[map[2]] + source[map[3]] =
33  * source[3] + source[1] + source[0] + source[5] =
34  * 40 + 20 + 10 + 60 =
35  * 130 */
```

zip_iterator

- A zip_iterator is a pointer associated with a vector of tuples.

```
1  #include <thrust/device_vector.h>
2  #include <thrust/tuple.h>
3  #include <thrust/iterator/zip_iterator.h>
4  #include <iostream>
5
6  #include <thrust/iterator/zip_iterator.h>
7
8  int main(){
9    thrust::device_vector<int> int_v(3);
10   int_v[0] = 0; int_v[1] = 1; int_v[2] = 2;
11
12   thrust::device_vector<float> float_v(3);
13   float_v[0] = 0.0; float_v[1] = 1.0; float_v[2] = 2.0;
14
15   thrust::device_vector<char> char_v(3);
16   char_v[0] = 'a'; char_v[1] = 'b'; char_v[2] = 'c';
17
18   // typedef these iterators for shorthand
19   typedef thrust::device_vector<int>::iterator   IntIterator;
20   typedef thrust::device_vector<float>::iterator FloatIterator;
21   typedef thrust::device_vector<char>::iterator  CharIterator;
```

zip_iterator

```
22    // typedef a tuple of these iterators
23    typedef thrust::tuple<IntIterator, FloatIterator, CharIterator>
          IteratorTuple;
24
25    // typedef the zip_iterator of this tuple
26    typedef thrust::zip_iterator<IteratorTuple> ZipIterator;
27
28    // finally, create the zip_iterator
29    ZipIterator iter(thrust::make_tuple(int_v.begin(), float_v.begin(),
          char_v.begin()));
30
31    *iter;   // returns (0, 0.0, 'a')
32    iter[0]; // returns (0, 0.0, 'a')
33    iter[1]; // returns (1, 1.0, 'b')
34    iter[2]; // returns (2, 2.0, 'c')
35
36    thrust::get<0>(iter[2]); // returns 2
37    thrust::get<1>(iter[0]); // returns 0.0
38    thrust::get<2>(iter[1]); // returns 'b'
39
40    // iter[3] is an out-of-bounds error
41 }
```

Outline

- Containers are fancy data storage classes used in the Standard Template Library (STL), the CPU C++ analog of Thrust.
- Examples of containers include:
  - `vector`
  - `deque`
  - `list`
  - `tack`
  - `queue`
  - `priority_queue`
  - `set`
  - `multiset`
  - `map`
  - `multimap`
  - `biset`

container.cu

- Thrust only implements vectors, but it's still compatible with the rest of STL's template classes.

```
1  #include <thrust/device_vector.h>
2  #include <thrust/copy.h>
3  #include <list>
4  #include <vector>
5
6  int main(void){
7    // create an STL list with 4 values
8    std::list<int> stl_list;
9    stl_list.push_back(10);
10   stl_list.push_back(20);
11   stl_list.push_back(30);
12   stl_list.push_back(40);
13
14   // initialize a device_vector with the list
15   thrust::device_vector <int> D(stl_list.begin(), stl_list.end());
16
17   // copy a device_vector into an STL vector
18   std::vector<int> stl_vector(D.size());
19   thrust::copy(D.begin(), D.end(), stl_vector.begin());
20
21   return 0;
22 }
```

Outline

Transformations

- A transformation is the application of a function to each element within a range of elements in a vector. The results are stored as a range of elements in another vector.
- Examples:
  - `thrust::fill()`
  - `thrust::sequence()`
  - `thrust::replace()`
  - `thrust::transform()`

transformations.cu

```
1  #include <thrust/device_vector.h>
2  #include <thrust/transform.h>
3  #include <thrust/sequence.h>
4  #include <thrust/copy.h>
5  #include <thrust/fill.h>
6  #include <thrust/replace.h>
7  #include <thrust/functional.h>
8  #include <iostream>
9
10 int main(void) {
11   // allocate three device_vectors with 10 elements
12   thrust::device_vector <int> X(10);
13   thrust::device_vector <int> Y(10);
14   thrust::device_vector <int> Z(10);
15
16   // initialize X to 0,1,2,3, ....
17   thrust::sequence(X.begin(), X.end());
18
19   // compute Y = -X
20   thrust::transform(X.begin(), X.end(), Y.begin(), thrust::negate<int>()
         );
21
22   // fill Z with twos
23   thrust::fill(Z.begin(), Z.end(), 2);
24
25   // compute Y = X mod 2
26   thrust::transform(X.begin(), X.end(), Z.begin(),
27                     Y.begin(), thrust::modulus<int>());
```

transformations.cu

```
28    // replace all the ones in Y with tens
29    thrust::replace(Y.begin(), Y.end(), 1, 10);
30
31    // print Y
32    thrust::copy(Y.begin(), Y.end(), std::ostream_iterator<int>(std::cout,
            "\n"));
33    return 0;
34  }
```

```
1  > nvcc transformations.cu -o transformations
2  > ./transformations
3  0
4  10
5  0
6  10
7  0
8  10
9  0
10 10
11 0
12 10
13 [landau@impact1 transformations]$
```

Reductions

- A reduction algorithm uses a binary operation to reduce an input vector to a single value. For example, here are equivalent ways to code the pairwise sum:

```
1 int sum = thrust::reduce(D.begin(), D.end(),
2   (int) 0, thrust::plus<int>());
3
4 int sum = thrust::reduce(D.begin(), D.end(),
5   (int) 0);
6
7 int sum = thrust::reduce(D.begin(), D.end())
```

- The third argument is the starting value of the reduction.
- The fourth argument is the binary operation that defines the kind of reduction.

Counting

- Another reduction: use `thrust::count()` to count the number of times a value appears in a vector.

```
1  #include <thrust/count.h>
2  #include <thrust/device_vector.h>
3  ...
4
5  // put three 1s in a device_vector
6  thrust::device_vector <int> vec(5,0);
7
8  vec [1] = 1;
9  vec [3] = 1;
10 vec [4] = 1;
11
12 // count the 1s
13 int result = thrust::count(vec.begin(), vec.end(), 1);
14 // result is three
```

Scans

- A scan, also called a prefix-sum, applies a function to multiple sub-ranges of a vector and returns the result in a vector of the same size. The default function is addition.

Text

```
1  #include <thrust/scan.h>
2  #include <thrust/device_vector.h>
3  #include <iostream>
4
5  int main(){
6    thrust::device_vector<int> data(6, 0);
7    data[0] = 1;
8    data[1] = 0;
9    data[2] = 2;
10   data[3] = 2;
11   data[4] = 1;
12   data[5] = 3;
13
14   thrust::inclusive_scan(data.begin(), data.end(), data.begin()); // in-
         place scan
15   // data is now {1, 1, 3, 5, 6, 9}
16
17  /* data[0] = data[0]
18   * data[1] = data[0] + data[1]
19   * data[2] = data[0] + data[1] + data[2]
20   * ...
21   * data[5] = data[0] + data[1] + ... + data[5]
22   */
23 }
```

Scans

- There are *exclusive scans* in addition to *inclusive scans*.

```
1  #include <thrust/scan.h>
2  #include <thrust/device_vector.h>
3  #include <iostream>
4
5  int main(){
6    thrust::device_vector<int> data(6, 0);
7    data[0] = 1;
8    data[1] = 0;
9    data[2] = 2;
10   data[3] = 2;
11   data[4] = 1;
12   data[5] = 3;
13   thrust::exclusive_scan(data.begin(), data.end(), data.end()); // in-
         place scan
14
15   // data is now {0, 1, 1, 3, 5, 6}
16
17  /* data[0] = 0
18   * data[1] = data[0]
19   * data[2] = data[0] + data[1]
20   * ...
21   * data[5] = data[0] + data[1] + ... + data[4]
22   */
23 }
```

Reordering

- The "Reordering" utilities provides subsetting and partitioning tools:
  - `thrust::copy_if()`: copy the elements that make some logical function return true.
  - `thrust::partition()`; reorder a vector such that values returning true precede values returning false.
  - `thrust::remove()` and `remove_if()`: remove elements that return false.
  - `thrust::unique()`: remove duplicates in a vector.

Partitions

```
1  #include <thrust/partition.h>
2
3  struct is_even{
4    __host__ __device__ bool operator()(const int x){
5      return (x % 2) == 0;
6    }
7  };
8
9  int main(){
10   int A[] = {1, 2, 3, 4, 5, 6, 7, 8, 9, 10};
11   const int N = sizeof(A)/sizeof(int);
12   thrust::partition(A, A + N,
13                     is_even());
14
15   // A is now {2, 4, 6, 8, 10, 3, 7, 1, 9, 5}
16   int i;
17   for(i = 0; i < N; ++i){
18     std::cout << "A[" << i << "] = " << A[i] << std::endl;
19   }
20   return 0;
21 }
```

- Notice: I can use host arrays directly.
- However, arrays stored on the GPU must be converted into device vectors or iterators before usage in Thrust algorithms.

Sorting

- `thrust::sort()`

```
1 #include <thrust/sort.h>
2 ...
3 const int N = 6;
4 int A[N] = {1, 4, 2, 8, 5, 7};
5 thrust::sort(A, A + N);
6 // A is now {1, 2, 4, 5, 7, 8}
```

- `thrust::sort_by_key()`

```
1 #include <thrust/sort.h>
2 ...
3 const int N = 6;
4 int keys[N] = { 1, 4, 2, 8, 5, 7};
5 char values[N] = {'a', 'b', 'c', 'd', 'e', 'f'};
6 thrust::sort_by_key(keys, keys + N, values);
7 // keys is now { 1, 2, 4, 5, 7, 8}
8 // values is now {'a', 'c', 'b', 'e', 'f', 'd'}
```

Sorting

- `thrust::stable_sort()`

```
1 #include <thrust/sort.h>
2 #include <thrust/functional.h>
3 ...
4 const int N = 6;
5 int A[N] = {1, 4, 2, 8, 5, 7};
6 thrust::stable_sort(A, A + N, thrust::greater<int>());
7 // A is now {8, 7, 5, 4, 2, 1}
```

Outline

Resources

- Guides:
  1. Bell N. and Hoberock J. Thrust. http://developer.download.nvidia.com/CUDA/training/webinarthrust1.mp4
  2. Savitch W. Absolute C++. Ed. Hirsch M. 3rd Ed. Pearson, 2008.
  3. CUDA Toolkit 4.2 Thrust Quick Start Guide. March 2012. http://docs.nvidia.com/cuda/thrust/index.html
- Code from today is posted at http://will-landau.com/gpu/thrust.html.

That's all for today.

- Series materials are available at http://will-landau.com/gpu.