Lecture 10

Thrust template library

Thrust library

- Another approach which aims to make coding in CUDA easier
- A higher level interface to the CUDA library which aims to make the programmer more productive, and bug free code easier to write
- Uses an approach relying on C++ constructs, so usefulness will depend on how proficient the programmer is with that style of programming
- Included with CUDA by default hence easy to use
- It is a template library, so all its code resides in header files, and no libraries need to be linked to make the code work
- Advantage of templates: you don’t need to write separate code for each datatype

Thrust library

- Under very active development, new features being added constantly
- the version included with CUDA may not be the latest, install the latest if you want the latest features
- code used in this lecture may not use the latest Thrust features, which may make it easier to code some of the things shown in this lecture (eg. SAXPY)
- more recent Thrust versions include support for OpenMP (for algorithms on host) and OpenACC

Quick reminder on C++ templates

- available in C++ but not in C
- templates permit function to take different datatypes as arguments, so no need to define a different function for each argument
- actually, compiler will generate separate object code for each datatype

```
#include <iostream>

template<class myTYPE>
void printtwice(myTYPE data)
{
std::cout << "Twice: " << data * 2 << std::endl;
}

int main(){
printtwice(2.0);
printtwice(2);
return 0;
}
```

Thrust library documentation

- Thrust main site:
  https://nvidia.github.io/cccl/thrust/

- NVIDIA Thrust documentation
  http://docs.nvidia.com/cuda/thrust/index.html

Thrust library

- C++ template library for CUDA based on C++ Standard Template Library (STL)
- Provides a large number of parallel algorithms. Many of these have direct analogs in the STL, and when the equivalent function exists, Thrust uses the same name. Eg.

  thrust::sort
  std::sort

  So need to be careful if using them both at the same time.
- Simplifies data movement between hosts and device, performs allocation/deallocation of memory for the programmer (so no need to explicitly free data structures created with Thrust)

Overloading is standard

- Thrust calls will do different things depending on arguments supplied
- Number of arguments may vary
- Debugging somewhat difficult, a programming bug will make compiler produce many lines of error output

Quick look at C++ STL library

- Provides containers: *vector*, *list*, *map* etc. and because it’s a template library, these can hold any data type
- STL provides a large collection of algorithms to manipulate data in these containers

```
#include <iostream>     // std::cout
#include <algorithm>    // std::reverse
#include <vector>       // std::vector
using namespace std;

int main(){
    vector<int> v(3);           // Declare a vector of 3 elements.
    v[0] = 7;
    v[1] = v[0] + 3;
    v[2] = v[0] + v[1];         // v[0] == 7, v[1] == 10, v[2] == 17
    reverse(v.begin(), v.end()); // v[0] == 17, v[1] == 10, v[2] == 7
    sort(v.begin(), v.end());    // v[0] == 7, v[1] == 10, v[2] == 17
    return 0;
}
```

Iterators

- What exactly are v.begin() and v.end() ?

```
reverse(v.begin(), v.end()); // v[0] == 17, v[1] == 10, v[2] == 7
```

- They are **iterators**, which are in turn generalization of pointers.
- *v.begin()* points at first element in vector v, *v.end()* points one element beyond the last element in vector v
- *v.begin()+1* points at second element in vector v etc.
- Iterators make it possible to decouple containers and algorithms
- Can be used to apply algorithms to subarrays, hence two arguments

“Hello World” in Thrust

- Simple program illustrating memory allocation, handled by Thrust both on host and device, with no need to free memory
- No explicit memory allocation/deallocation
- It’s possible to access containers stored on device directly

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

int main(void){

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

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

// manipulate device values from the host
d_vec[0] = 13;
d_vec[1] = 27;
std::cout << "sum: " << d_vec[0]+d_vec[1] << std:: endl;
return 0;
}
```

Vectors in Thrust

- Thrust vector has useful methods associated, for instance, using some vector H as example:
  H.size()   - number of objects stored in vector
  H.resize(N) - resize vector so it stores N objects
- To print contents can use simple loop

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

- Can access both device and host vectors, but of course access to device vector will require a slow CUDA memory copy under the hood, so should be used sparingly

```
//all needed include files here
int main(void)
{
   // initialize all ten integers of a device_vector to 1
   thrust::device_vector<int> D(10, 1);

   // 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());

   // copy all of H back to the beginning of D
   thrust::copy(H.begin(), H.end(), D.begin());

   // print D
   for(int i = 0; i < D.size(); i++)
       std::cout << "D[" << i << "] = " << D[i] << std::endl;
   return 0;
}
```

Compiling Thrust

- Thrust is part of CUDA, so Thrust code should be placed in .cu files and compiled with **nvcc**
- No special flags are required, nvcc will find the thrust headers automatically, and no libraries need linking as it’s a template library

CUDA and Thrust can be mixed

```
size_t N = 10;
// 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);
```

```
size_t N = 10;
// 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);
```

Thrust algorithms

- All algorithms in Thrust have implementation for both host and device
- When invoked with host iterator, then dispatched on host
- When invoked with device iterator, dispatched to device
- Except for **thrust::copy**, all arguments to a Thrust algorithm should live in the same place: either all on the host or all on the device
- If this is not satisfied, compiler will produce an error message

Transformation algorithms

- Apply operation to each element in some input range, stores result in destination range
- thrust::sequence, thrust::replace are example
- thrust::transform allows some function to be applied to one or more vectors
- The function supplied via a functor
- Standard operations come predefined

```
// vector addition  X+Y=Z
thrust::transform(X.begin(), X.end(), Y.begin(), Z.begin(),
     thrust::plus<float>());
```

- More complicated operations need to be written by programmer

Generating vector

- thrust::generate can be used to fill out values of a vector

```
thrust::host_vector<int> h_points(N);
thrust::generate(h_points.begin(), h_points.end(), somefunction);
```

```cpp
int main(void)
{
   // allocate three device_vectors with 10 elements
   thrust::device_vector<int> X(10);
   thrust::device_vector<int> Y(10);
   thrust::device_vector<int> Z(10);

   // 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"));

   return 0;
}
```

Functors in C++

- A C++ class/structure that acts as a function

```
#include <iostream>

struct absValue
{
     float operator()(float f) {
          return f > 0 ? f : -f;
     }
};

int main( )
{
     using namespace std;

     float f = -123.45;
     absValue aObj;
     float abs_f = aObj(f);
     cout << "f = " << f << " abs_f = " << abs_f << endl;
     return 0;
}
```

Functors in C++ (cont)

- Now we can pass additional parameters inside the “function” at the object initialization (constructor) stage:

```cpp
#include <iostream>
class myFunctorClass
{
    public:
         myFunctorClass (int x) : _x( x ) {}
         int operator() (int y) { return _x + y; }
    private:
         int _x;
};
int main()
{
    myFunctorClass addFive( 5 );
    std::cout << addFive( 6 );
    return 0;
}
```

Saxpy example

- We’ll revisit SAXPY problem, now using Thrust
- Folder: /home/syam/CSE746/thrust/
- We’ll first try a slow way, using multiplication followed by addition. This will require a temporary storage vector. (No functors!)

Include statements to use:

```
#include <thrust/transform.h>
#include <thrust/device_vector.h>
#include <thrust/host_vector.h>
#include <thrust/functional.h>
#include <iostream>
#include <iterator>
#include <algorithm>
```

- We’ll use the following Thrust templates:

```
device_vector (x3)
fill
transform  (x2: “multiplies” and “plus”)
```

1) Initialize x, y host vectors to some values, copy them to GPU with “device_vector” templates.
2) Use “device_vector” to create the “temp” vector of the required length on GPU
3) Use “fill” template to fill it with the constant A value
4) Use “transform(…, thrust::multiplies<float>())” to compute “A * X → temp”
5) Use “transform(…, thrust::plus<float>())” to compute “A * X + Y → Y”

Faster SAXPY

- This will require writing a functor to do the operation in one step
- In the functor definition, we’ll use the constructor to pass the “A” constant value to the class.
- The overloaded “operator()” will have two “functional” arguments – “x” and “y”. Prepend it with “__host__ __device__”.
- We’ll use a single thrust::transform template to compute A*X+Y->Y

Reduction

- Provide range of the sequence, initial value and operation

```
int sum = thrust::reduce(D.begin(), D.end(), (int) 0, thrust::plus<int>());
```

- Above options are so common that they are default, so three lines of code below all do the same thing

```
int sum = thrust::reduce(D.begin(), D.end(), (int) 0, thrust::plus<int>());
int sum = thrust::reduce(D.begin(), D.end(), (int) 0);
int sum = thrust::reduce(D.begin(), D.end())
```

Kernel fusion for reduction kernels

- Compute norm of vector in one kernel. First define functor for the squaring operation

```
// square<T> computes the square of a number f(x) -> x*x
template <typename T>
struct square
{
   __host__ __device__
       T operator()(const T& x) const {
           return x * x;
       }
};
```

Kernel fusion for reduction kernels (cont.)

- Use transform_reduce (fusion of transform and reduce)

```
int main(void)
{
   // initialize host array
   float x[4] = {1.0, 2.0, 3.0, 4.0};

   // transfer to device
   thrust::device_vector<float> d_x(x, x + 4);

   // setup arguments
   square<float>        unary_op;
   thrust::plus<float> binary_op;
   float init = 0;

   // compute norm
   float norm = std::sqrt( thrust::transform_reduce(d_x.begin(), d_x.end(),
unary_op, init, binary_op) );
   std::cout << norm << std::endl;
   return 0;
}
```

Counting iterator

- Acts as array but does not require any memory storage (i.e. it’s computed on the fly as required)

```
#include <thrust/iterator/counting_iterator.h>
…

// create iterators
thrust::counting_iterator<int> first(10);
thrust::counting_iterator<int> last = first + 3;

first[0]   // returns 10
first[1]   // returns 11
first[100] // returns 110

// sum of [first, last)
thrust::reduce(first, last);   // returns 33 (i.e. 10 + 11 + 12)
```

Sorting in Thrust

```
#include <thrust/sort.h>
…

const int N = 6;
int A[N] = {1, 4, 2, 8, 5, 7};

thrust::sort(A, A + N);
// A is now {1, 2, 4, 5, 7, 8}
```

Sorting by key in Thrust

```
#include <thrust/sort.h>
…

const int N = 6;
int    keys[N] = {  1,   4,   2,   8,   5,   7};
char values[N] = {'a', 'b', 'c', 'd', 'e', 'f'};

thrust::sort_by_key(keys, keys + N, values);
// keys is now   {  1,   2,   4,   5,   7,   8}
// values is now {'a', 'c', 'b', 'e', 'f', 'd'}
```

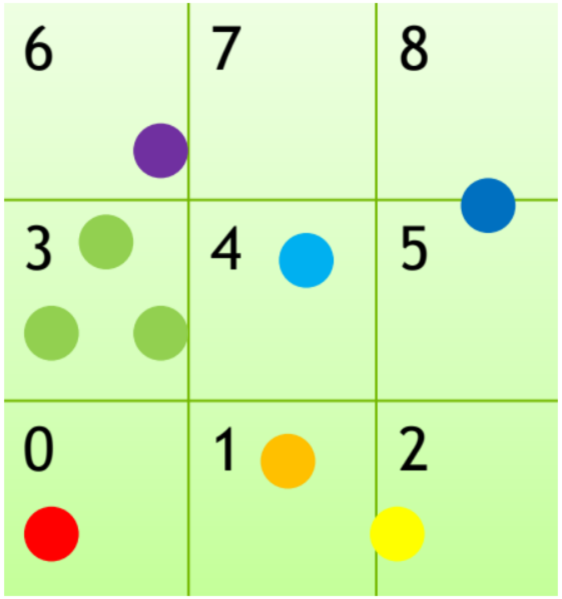
Hands on exercise

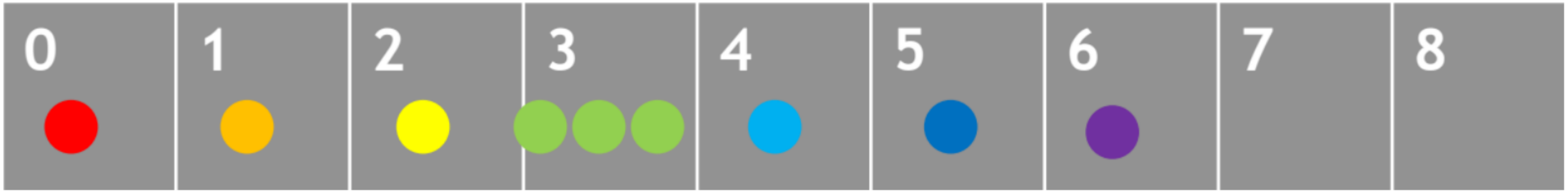
- Write code which sorts a 1D array of integers using Thrust library on gpu
- Compare to the existing STL (cpu) version, sort.cc (compile with g++ -O2): /home/syam/CSE746/thrust/sort.cc
- Test for vector size 32 << 20 (which is 32 times 2 to power 20)

Include statements:

```
#include <thrust/transform_reduce.h>
#include <thrust/functional.h>
#include <thrust/device_vector.h>
#include <thrust/host_vector.h>
#include <thrust/sort.h>
#include <cmath>
#include <iostream>
#include <iterator>
#include <algorithm>
#include <sys/time.h>
#include <time.h>
```

Bucket sorting





Bucket sorting example

- Write a program to solve a 2D bucket sorting problem
- Generate N points inside a 2D unit square (0,1)x(0,1)
- Divide square into 2D grid of buckets, dimension width x height, assign 1D numbering to buckets
- For each point, determine the bucket it belongs to
- Sort the points, using the bucket index as key
- Count how many points are in each bucket

Stages

1. Create random points
2. Compute bucket index for each point
3. Sort points by bucket index
4. Count size of each bucket

Stage 1 - create random points

- Generate them on host, then copy to device
- Use **thrust::generate** to fill out host vector
- Use CUDA datatype float2 to store x,y coordinates of points
- Can use:
  rand() / (RAND_MAX + 1.0f)
  to generate random values in (0,1) range
- Employ **make_float2** function from CUDA:

  float2 myVar = make_float2(float x, float y)

Stage 2 - compute bucket index

- Write functor which takes float2 and returns the 1D cell index
- This functor should store width and height in its structure (as w,h) via its constructor, obtaining the values upon initialization
- Functor should take float2 input, and return integer index
  y*w+x
  where
  x=”x_coordinate” * w
  y=”y_coordinate” *h
- Once have the functor, use thrust::transform to compute bucket index

Stage 3 - sort points by bucket index

- Use thrust::sort_by_key

Stage 4 - count number of points in each bucket

Allocate
bucket_begin,bucket_end

vectors to indicate where first/last element of each bucket is positioned

use

thrust::counting_iterator
thrust::lower_bound
thrust::upper_bound