CSE 746 - Parallel and High Performance Computing Lecture 9 - Thrust template library

Pawel Pomorski, *HPC Software Analyst*
SHARCNET, University of Waterloo
ppomorsk@sharcnet.ca
http://ppomorsk.sharcnet.ca/

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 TYPE>

void printtwice(TYPE data)
{
std::cout << "Twice: " << data * 2 << std::endl;
}

int main(){

printtwice(2.0);
printtwice(2);

return 0;
}
```

Thrust library documentation

- Thrust main site:
  http://thrust.github.io/
- Thrust library - complete reference of calls:
  http://thrust.github.io/doc/namespacethrust.html
- NVIDIA Thrust documentation
  http://docs.nvidia.com/cuda/thrust/index.html
- Thrust at Google Code (not maintained)
  https://code.google.com/p/thrust/
- Example reference on standard C++ STL library
  http://www.sgi.com/tech/stl/

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);
```

Exercise 0

- write some simple test programs using Thrust vectors

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);
```

```
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;
}
```

Functor in C++

- a data structure which can be called as a function
- permits having a function with a stored state

```
#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;
}
```

Exercise 1

- revisit SAXPY problem, do it again using Thrust
- first try a slow way, using multiplication followed by addition. This will require a temporary storage vector.

Slow SAXPY

- cost: 4N reads, 3N writes

```
void saxpy_slow(float A, thrust::device_vector<float>& X,
thrust::device_vector<float>& Y)
{
    thrust::device_vector<float> temp(X.size());

    // temp <- A
    thrust::fill(temp.begin(), temp.end(), A);

    // temp <- A * X
    thrust::transform(X.begin(), X.end(), temp.begin(), temp.begin(),
thrust::multiplies<float>());

    // Y <- A * X + Y
    thrust::transform(temp.begin(), temp.end(), Y.begin(), Y.begin(),
thrust::plus<float>());
}
```

Faster SAXPY

- this will require writing your own functor to do the operation in one step
- amount to kernel fusion: will have one kernel under the hood instead of two

Faster SAXPY

- cost: 2N reads, N writes

```
struct saxpy_functor
{
    const float a;

    saxpy_functor(float _a) : a(_a) {}

    __host__ __device__
        float operator()(const float& x, const float& y) const {
            return a * x + y;
        }
};

void saxpy_fast(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));
}
```

Reduction

- provide range of sequence, initial value and operation

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

- above options 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

- use transform_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'}
```

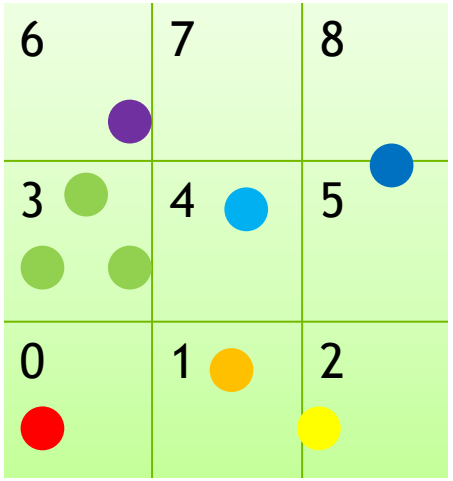
Exercise 2

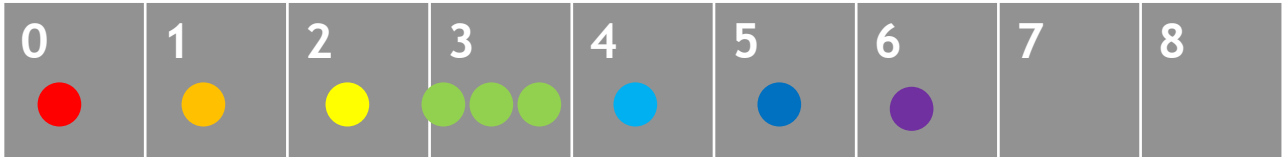
- write code which sorts a 1D array of integers, using Thrust library, both on host and on gpu
- write another code using the STL library (compile with g++ -O2)
- compare performance between STL, Thrust on host, and Thrust on GPU
- test for vector size 32 << 20 (which is 32 times 2 to power 20)
- use timing utility from previous lectures (saxpy_timed has it)

Exercise 3

- 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

Exercise 3





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

Stage 2 - compute bucket index

- write functor which takes float2 and computes
- this functor should store width and height in its structure (as w,h), obtaining the values upon initialization
- functor should take width2 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