# AN INTRODUCTION TO PARALLEL COMPUTING

## CHAPTER 14 : THRUST LIBRARY

BY FRED ANNEXSTEIN, PHD

# 14. THRUST LIBRARY

## CHAPTER OBJECTIVES

In this chapter we study the CUDA library known as Thrust. We show the features and usage of Thrust to more easily program GPUs running CUDA. We end the chapter with a simple Monte Carlo simulation code for modeling collisions of balls and bins.

## THRUST

CUDA comes with many standard libraries, algorithms, and data structures for use with accelerated GPU's. Here we will focus on the Thrust library, which provides a collection of parallel algorithms and data structures, presented with the similar syntax as C++'s Standard Template Library.

Programming with Thrust makes it easy to use device vectors in CUDA kernels. Thrust can greatly improve the readability of your code. Users can develop high-

performance applications very rapidly by using Thrust by providing programmers with the power to leverage parallel primitives. Many of the algorithms we have covered so far are included as primitives in Thrust. Thrust makes common operations concise and readable, and simplifies programming by hiding cudaMalloc, cudaMemcpy and cudaFree. Using Thrust, programmers need not worry about device memory copies and implementing standard algorithms like sort and reduce.

## THRUST VECTORS

Thrust has two template containers called vectors: with one for the host and one for the device.

```
thrust::host_vector <T>

thrust::device_vector <T>

// Here is an allocation of a host vector with two
elements thrust::host_vector <int> h_vec(2);

// Here is a copy of a host vector to device
thrust::device_vector d_vec = h_vec;

// Here is how to set device values from the host
d_vec[0] = 13;
d_vec[1] = 27;

// Here is how to print the sum of device value
std::cout << "sum: " << d_vec[0] + d_vec[1] <<
std::endl;
```

Thrust has several algorithms that use the container vector. Algorithms include sorting, reductions, and scans.

```
thrust::sort()

thrust::reduce()

thrust::inclusive_scan()
```

# THRUST ITERATORS

In Thrust, sequences are defined by pair of iterators. Iterators are like smart pointers and we can reference the begin and end positions as follows:

```
// Allocate device vector with 4 items
thrust::device_vector d_vec(4);

thrust::device_vector::iterator begin =d_vec.begin();
thrust::device_vector::iterator end = d_vec.end();

int length = end – begin;
end = d_vec.begin() + 3;
// This defines a sequence of 3 elements in the d_vec
```

Iterators allow programmers to track space usage. Here we initialize 1000 random values in a vector on the host using the generate function and applying rand. We then copy this vector onto the device using an assignment statement. Next we run a parallel reduce function to compute the sum of the random values, both on the host and the devise using Thrust iterators.

```
thrust::host_vector h_vec(1000);

thrust::generate(h_vec.begin(), h_vec.end(), rand);

thrust::device_vector d_vec = h_vec;

int h_sum = thrust::reduce(h_vec.begin(),
h_vec.end());

int d_sum = thrust::reduce(d_vec.begin(),
d_vec.end());
```

The next example shows how to use STL lists and copy them to device vectors using Thrust iterators.

```
// Declare list container on host
std::list h_list;

h_list.push_back(13);
h_list.push_back(27);

// Allocate space on device to store list
thrust::device_vector d_vec(h_list.size());

// Copy list to device using begin and end refs
thrust::copy(h_list.begin(), h_list.end(),
d_vec.begin());

// An alternative method
d thrust::device_vector d_vec(h_list.begin(),
h_list.end());
```

# SORTING RANDOM VECTORS

The following source code example generates 32 Million random numbers serially on the host, and then transfers them to the parallel (CUDA) device, which is then sorted on the device, and finally copied back to the host.

```
int main(void) {

thrust::host_vector<int> h_vec(32 << 20);
std::generate(h_vec.begin(), h_vec.end(), rand);
thrust::device_vector<int> d_vec = h_vec;
thrust::sort(d_vec.begin(), d_vec.end());
thrust::copy(d_vec.begin(), d_vec.end(),
                                h_vec.begin());
return 0;
}
```

Be aware that depending on the system, we usually have to include the library files to access the Thrust data structures and functions.

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

# FUNCTION TEMPLATES AND GENERIC ALGORITHMS

In C++ -style programming we use function templates and functors to support polymorphism, define functions that work on multiple types, and to define functors or function objects that can be passed as arguments.

Here we define a function template to add two values of an arbitrary type T.

```
template< typename T >
T add(T a, T b)
{ return a + b; }

// adding integers
int x = 10; int y = 20; int z;
z = add<inbt> (x,y);  // type of T explicit
z = add(x,y); // type of T automatic

// adding  floats
float x = 10.0f; float y = 20.0f; float z;
z = add<int>(x,y);  // type of T explicit
z = add(x,y);      // type of T automatic
```

Here we define a functor - function object - which is then invoked to execute a sum function on two examples, one with int objects and one with float objects.

```
template< typename T >
class add {
public: T operator()(T a, T b)
{ return a + b; } };

int x = 10; int y = 20; int z;
add<int> func; // create an add functor for T=int
z = func(x,y); // invoke functor on x and y

float x = 10; float y = 20; float z;
add<float> func; // create an add functor for T=float
z = func(x,y); // invoke functor on x and y
```

Next we show how to create a generic function that we will apply to arrays of arbitrary types. Here we will use a function that is a binary operation for a type T and transform a pair of arrays by applying the function component-wise to the elements in two arrays x and y thereby creating z.

```
// apply function f to seqs x,y and store in z
template <typename T, typename Function>
```

```
void transform(int N, T * x, T * y, T * z, Function
f)
{
 for (int i = 0; i < N; i++)
 z[i] = f(x[i], y[i]);
}

int N = 100;
int x[N]; int y[N]; int z[N];
add<int> func; // add functor for T=int

transform(N, x, y, z, func);    // z[i] = x[i] + y[i]
transform(N, x, y, z, add<int>()); // equivalent call
```

# THRUST REDUCE

Recall the parallel reduce function, which we programmed earlier using a tree-based approach. Reduce will take a (device) vector, starting value, and a binary function. Reduce will compute the complete operation of applying the function across the entire sequence. In these examples, we will pass the functors add<int>, add<float>, and maximum<int> to the Thrust reduce function along with a 0 starting value.

```
#include <thrust/reduce.h>
// declare storage
device_vector<int> i_vec = ...
device_vector<float> f_vec = ...

reduce(i_vec.begin(), i_vec.end(), 0, add<int>());

reduce(f_vec.begin(), f_vec.end(),0.0f,add<float>());

reduce(i_vec.begin(), i_vec.end(), 0,maximum<int>());
```

# SORTING WITH GENERIC COMPARISONS

```
// define an operator to compare the x component
//    of two float2 structures

struct compare_float2
{
 __host__ __device__
 bool operator()(float2 a, float2 b)
 {
 return a.x < b.x;
 }
};


device_vector<float2> vec = ...

compare_float2 comp;

// Here we sort elements by x component
sort(vec.begin(), vec.end(), comp);
```

# BALLS AND BINS SIMULATION USING THRUST

We now write a CUDA program using a number of the Thrust operations we have been discussing in this chapter. The simulation that we carry out is called balls_and_bins. In the balls_and_bins we want to simulate the throwing of some number of balls into some number of bins. For this example those numbers will be the same number n.

We are interested in creating a simulation to learn how many bins are hit when n balls are thrown at random into n bins. To create each of the simulations we need to generate a random sequence of numbers in the range

1 to n. To do this we will create a random number functor. In the main loop we will use a for-loop to set the values of n as powers of 2 from 2^1 to 2^20.

We create each random sequence on the host and then copy to the device. On the device we will call the sort function followed by the unique function which is a parallel filter that takes out of the sorted array all duplicate elements in the array.

We will use Thrust's *unique* function which works as follows: for each group of consecutive vector elements with the same value, *unique* removes all but the first element of the group. The return value is an iterator *new_end* such that no two consecutive elements in the range upto *new_end* are equal. The function unique is stable, meaning that the relative order of elements that are not removed is unchanged.

We finish the simulation by determining the number of unique items in the list, which is the same as the number of bins that are hit by any of the thrown balls. We output this representing the number of non-empty bins.

From the results below we see that as a fraction of the total number of bins, the number of non-empty bins appears to approach the number 0.632. Though a

mathematical argument we can show that this ratio approaches 1 - 1/e =0.632120558. Here is the full program of the balls_and_bins simulation.

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

struct rand_functor
{const int a;
rand_functor(int _a) : a(_a) {}
__host__ __device__
int  operator()() const {
return rand() % a;
}};


int main(void) {

for (int n =2; n< (2<<20); n *=2){
thrust::host_vector<int> hv(n);
thrust::generate(hv.begin(), hv.end(),
rand_functor(n));

thrust::device_vector<int> dv = hv;
thrust::sort(dv.begin(), dv.end());
thrust::device_vector<int>::iterator newend =

thrust::unique(dv.begin(),dv.end());

int numunique = newend - dv.begin();

printf("#bins %d  # distinct: %d  ratio %f\n",n,
numunique,  double

(numunique) / n);

}
return 0;
}

Output:

#bins 2  # distinct: 2  ratio 1.000000
#bins 4  # distinct: 2  ratio 0.500000
#bins 8  # distinct: 5  ratio 0.625000
#bins 16  # distinct: 12  ratio 0.750000
```

```
#bins 32  # distinct: 18  ratio 0.562500
#bins 64  # distinct: 36  ratio 0.562500
#bins 128  # distinct: 79  ratio 0.617188
#bins 256  # distinct: 163  ratio 0.636719
#bins 512  # distinct: 323  ratio 0.630859
#bins 1024  # distinct: 654  ratio 0.638672
#bins 2048  # distinct: 1291  ratio 0.630371
#bins 4096  # distinct: 2602  ratio 0.635254
#bins 8192  # distinct: 5154  ratio 0.629150
#bins 16384  # distinct: 10319  ratio 0.629822
#bins 32768  # distinct: 20801  ratio 0.634796
#bins 65536  # distinct: 41388  ratio 0.631531
#bins 131072  # distinct: 82811  ratio 0.631798
#bins 262144  # distinct: 165811  ratio 0.632519
#bins 524288  # distinct: 331535  ratio 0.632353
#bins 1048576  # distinct: 663439  ratio 0.632705
```

# MODULE 14 LAB

For this lab you will practice working with the Thrust library to run a different type of balls and bins simulation. In the previous example we ran a simulation to determine how the number of occupied (or unique) bins grows when we throw n balls into n bins at random.

For this lab you will write a simulation code to investigate how the size of the largest bin grows when we throw n balls into n bins. Your program should print an analogous table for n ranging across powers of 2 from 2^1 to 2^20, and for each n you will print out the size of the largest bin with each n.