Project 2: Cuda Programming
By Scott Fraundorf

Table of Contents

Thrust

Algorithm Description

The thrust sorting algorithm is a variation of the radix sorting method. [1]

Time Complexity

The Time complexity of the Thrust algorithm is O(n), see the description under Performance.

Implementation Details

Implementing thrust sort simply involved using the command from the thrust library as shown in Figure 10.

Performance

The performance of the Thrust algorithm is O(n), based on the linear regression performed in Excel which yielded an equation of $y = 9.9152x + 354.05$ and an R^2 of 0.9978. The other types of regressions whether polynomial or exponential had a lower R^2.

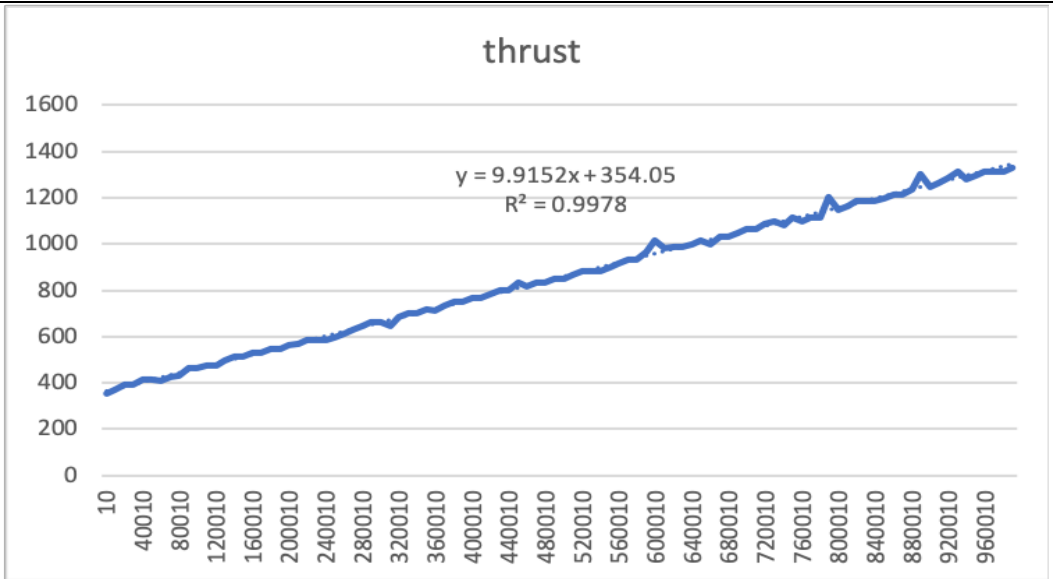


Figure 1, thrust algorithm result showing time in milliseconds (y-axis) vs. number of integers (x-axis)

Singlethread

Algorithm Description

I simply implemented bubble sort using single threaded CUDA programming.

Time Complexity

Bubble sort has a time complexity of $O(n^2)$. If implemented with a single thread it should still have a time complexity of $O(n^2)$.

Implementation Details

Bubble sort is implemented in the kernel, see Figure 11. In the main function of the program, the array of integers is copied into a CUDA array which is then passed into a 1 block and 1 thread on the side of a block CUDA grid, and executed as shown in Figure 12.

Performance

According to the linear regression, the algorithm performed in polynomial time, specifically O(n^2). The polynomial regression returned an equation of $1.1643x^2 - 0.1752x + 357.06$, with an R^2 of 0.9999. See Figure 2.

The single-threaded program is performing as expected since bubble sort has a O(n^2) time complexity.

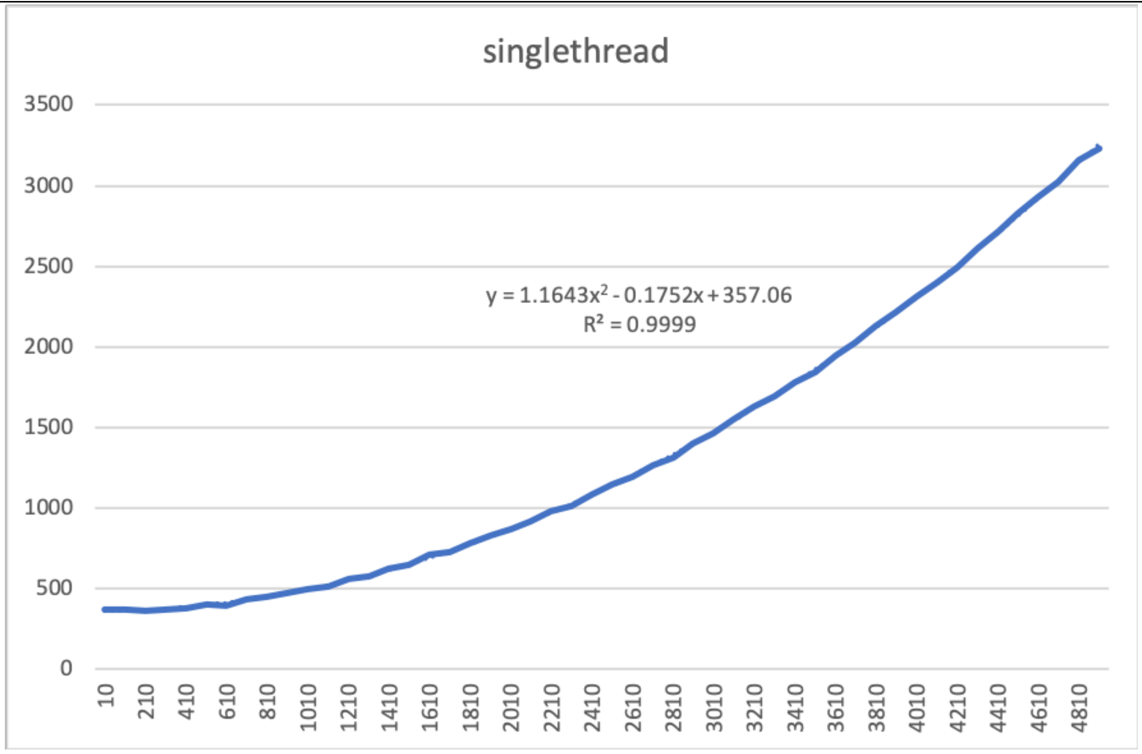


Figure 2, single-threaded algorithm result showing time in milliseconds (y-axis) vs. number of integers (x-axis)

Multithread

Algorithm Description

I used a bucket bubble sort, where the number of buckets is calculated based on the CUDA grid dimensions. The buckets are distributed over the grid of blocks and threads within blocks, and the work carried out by the thread is bubble sort on that specific bucket's integers.

Time Complexity

According to [2], bucket sort assumes that the numbers to be sorted were created by a random process. Because of that bucket sorts running time in the average-case is O(n). It will also be $\theta(n)$ if the sum of squares regarding the size of buckets is linear. [2]

Implementation Details

First the CUDA grid is determined. This is done by creating a variable for the estimated threads equal to the size of the array divided by 100. The array size was divided by 100 to ensure a sizeable work load for each thread. This variable is an estimate, because once the grid proportions are calculated, the actual number of threads will most likely change from the estimate.

The side of the square grid in threads is determine by taking the square-root of the estimated threads plus 1. The square-root is used to find the number of threads that there should be on a side of the square. It probably would have worked better to take the ceiling rather than adding 1, the purpose of which is make sure there are more not less threads on a side to produce the estimated threads.

In the first implementation, rhe side of a block is calculated by dividing the side of the grid in threads by 100. The number of blocks is calculated as the threads on the side of the grid divided by the threads on the side of a block plus 1. Again, to ensure that there are more than enough to produce the estimated number of threads. The actual number of threads, and therefore the actual number of buckets used in the sort is the number of blocks on a side times the number of threads on the side of a block squared. See Figure 3.

In the first implementation, at least with up to 1,000,000 integers being sorted, the number of threads per block always ended up being 1. This is because even with 1 million integers, 10^6 / 100 = 10^4. The square-root of 10^4 is 100. 100 / 100 = 1. This implementation worked, however another strategy is to use the logarithm to determine the number of threads on the side of a block. When this was used, the performance was slightly better when up to 1,000,000 integers was tested. In that case, there were 6 threads on the side of a block, when one million integers was sorted.

$$E = \frac{s}{100}, if\ E = 0\ \{E = 1\}$$
$$sG = \sqrt{E} + 1$$
$$Implementation\ 1: sB = \frac{sG}{100}, if\ sB = 0\ \{sB = 1\}$$
$$Implementation\ 2: sB = \log s, if\ sB = 0\ \{sB = 1\}$$

$$x = \frac{sG}{sB} + 1$$
$$T = (x * \ sB)^2$$
$$\beta = T$$

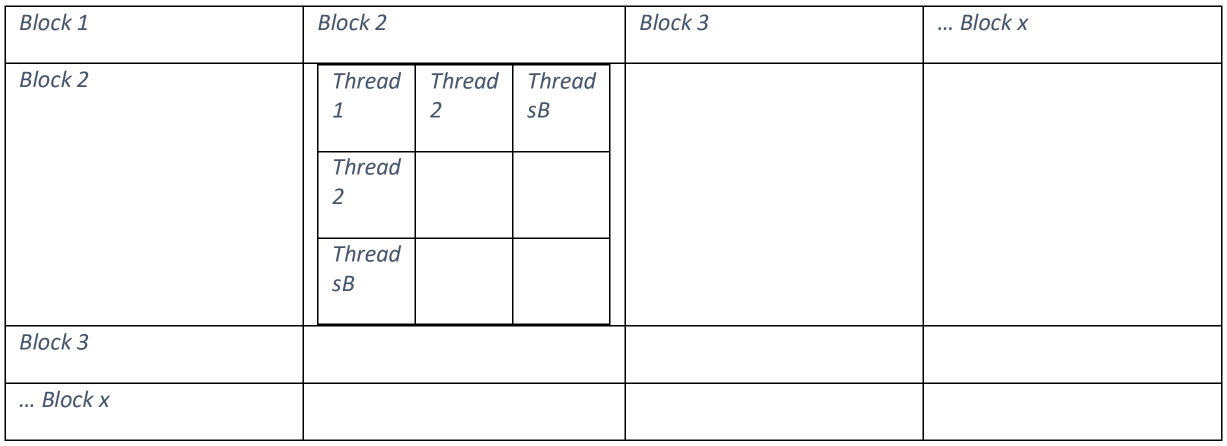


Figure 3, equations for number of buckets (top) where E = estimated threads, s = array size, sG = threads on grid side, sB = threads on block side, x = blocks on grid side, T = number of threads, β = number of buckets. Diagram of grid (bottom).

Since a 2D array could not be passed into the kernel, instead a 1D array was used where the individual buckets are separated by -1 (since the random arrays do not include negative numbers). Also passed into the function were two arrays, one for bucket start indexes and the other for bucket finish indexes, the purpose being so that determining bucket starts and finishes only had to be done once, and not in each thread.

There were issues with having too large of number of threads in a block. This was solved in the first implementation by dividing by 100 twice in course of creating the CUDA grid, or in implementation 2 using the $\log_{10}$ of the array size to determine the threads on a side of each block. As seen in Figure 13, within the kernel the current bucket is determined from the x and y location in the grid. (i.e. current bucket = x location + (y location * threads on the side of the grid))

```
estimated threads: 10000
estimated diameter: 101
threads on side of block: 6
blocks on side of grid: 17
threads (buckets): 10404
```

Figure 4, grid dimensions and number of buckets for sorting 1 million integers and $\log_{10}$ block sides.

Array elements were assigned to buckets by dividing the current value by the max value + 1 and multiplying by the number of buckets as seen in Figure 14.

Performance

From the graph, the performance of the algorithm appears to be linear, even though the regression with the highest R^2 was exponential. This may be a result of the trials that resulted in a segmentation fault throwing off the regression. Up to 1,000,000 integers, the multithreaded program outperformed the thrust algorithm, both of which outperformed by many orders of magnitude the single threaded version. See Figure 8 and Figure 9.

Figure 5 shows the multithreaded results when the sides of the blocks are the log base 10 of the initial array size. For instance, if the array size is 1,000,000 than the side in threads of a block is 6 threads. This implementation had slightly better performance than the one shown in Figure 6, that used the original implementation.

The simplest reason why the performance of the algorithm was O(n) is that the algorithm is performing bubble sort on 100 integers in each thread, which is running concurrently. If the size of the array that is to be sorted increases, the number of threads sorting 100 integers also increases. This causes there to be a much more gradual increase in execution time.

There was a limit of ~32 threads per block. The array of buckets separated by -1 was copied to CUDA memory. Each thread operated on its own portion of the array. This would have been occurring in consecutive locations of memory. According to Satish, et. al., transactions with memory are grouped together in such a way that memory throughput is greatly increased in a GPU multithreaded system.

It is also important to make sure the algorithm is parallelized enough because GPUs do not use a cache, and instead just use the multithreading. In the implementation of bucket bubble sort, this was accomplished by giving each thread roughly 100 integers to sort. Satish, et. al. mention that 5,000 threads is required to make good use of modern GPUs. When sorting 1,000,000 numbers the multithreaded program created 10,404 threads. See Figure 4.

Another issue within GPUs is differences in execution within a warp. Within a block, all threads were performing bubble sort, so there was not really much in the way of a different execution sequence, besides operating on a different set of numbers. This was the same between different blocks as well. [3]

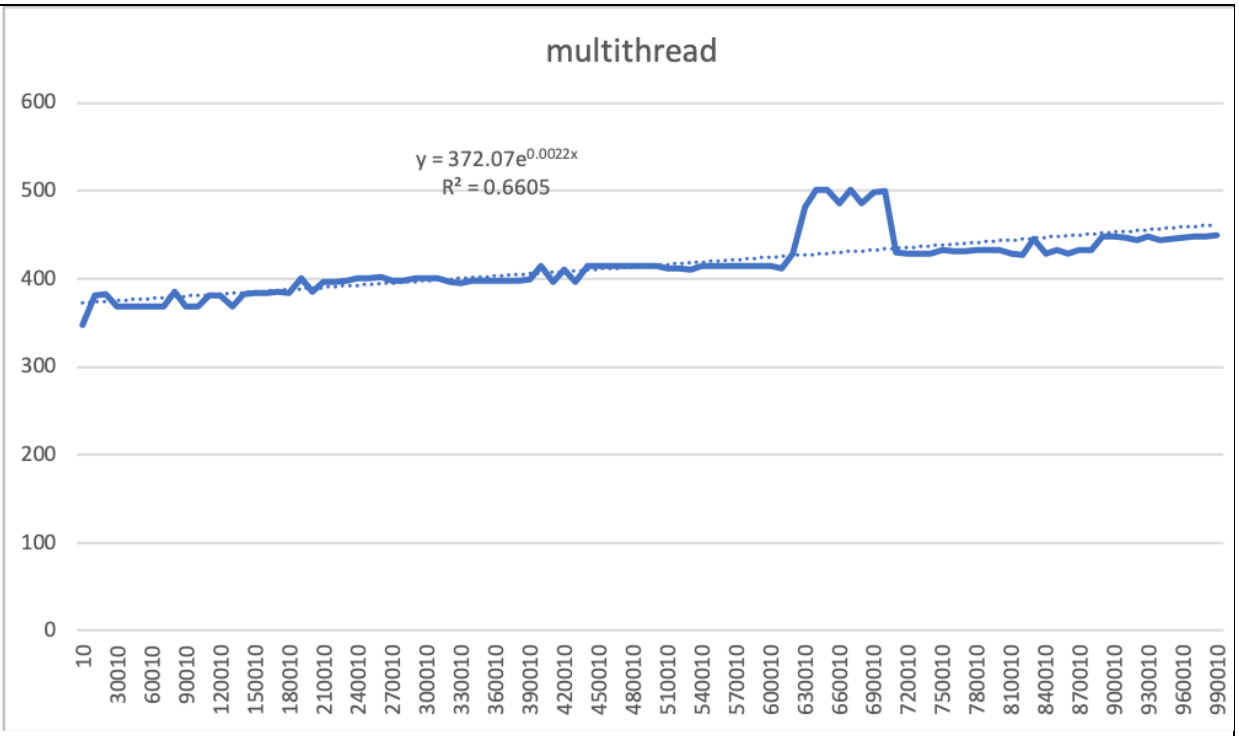


Figure 5, multithreaded algorithm with block sides that are $\log_{10}$ of the array size. Time in milliseconds (y-axis) vs. number of integers (x-axis)

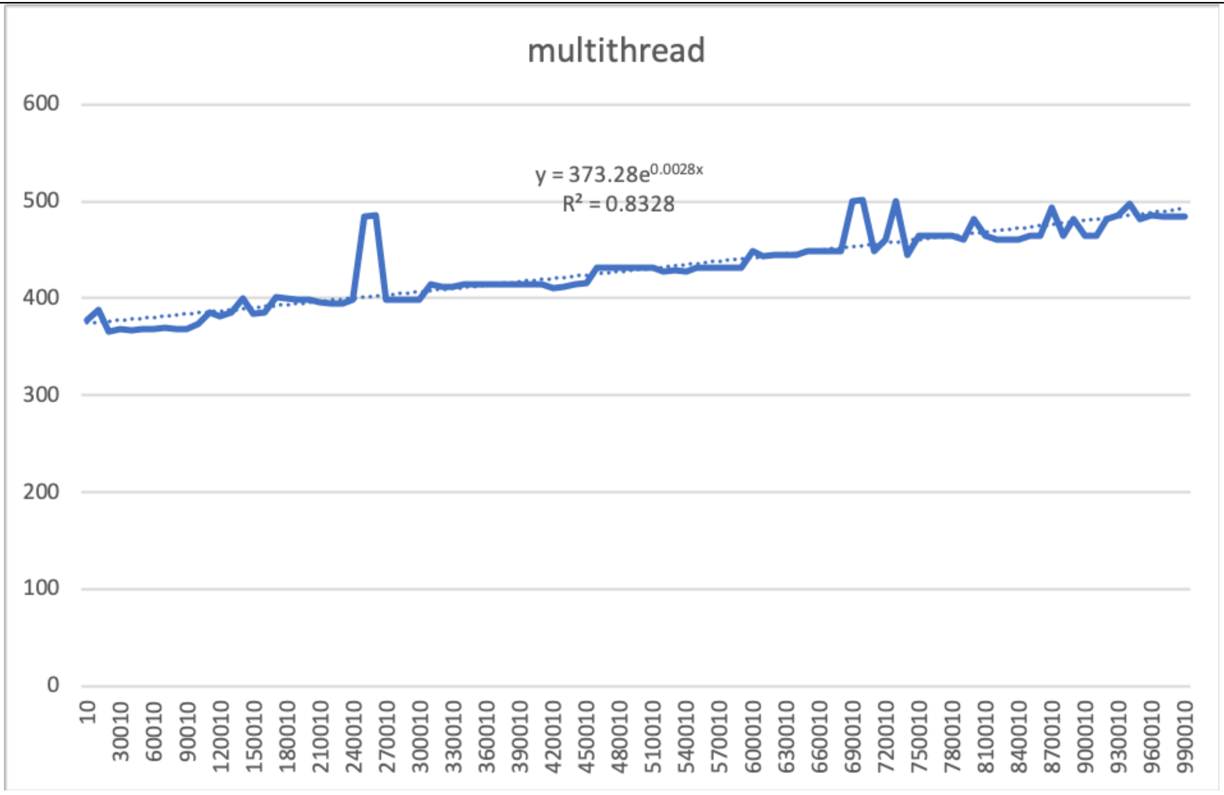
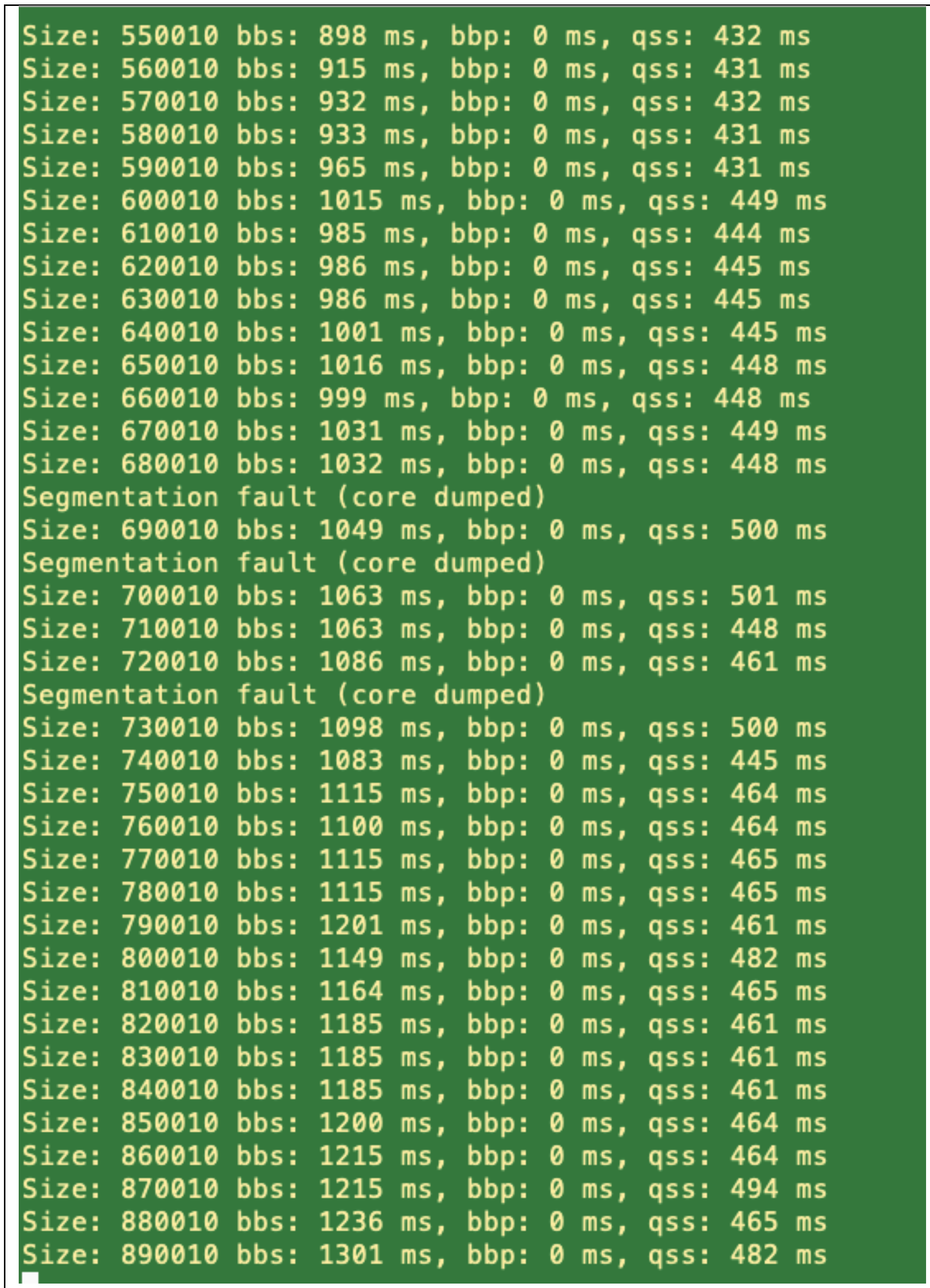


Figure 6, Multithread algorithm result showing time in milliseconds (y-axis) vs. number of integers (x-axis)



```
Size: 550010 bbs: 898 ms, bbp: 0 ms, qss: 432 ms
Size: 560010 bbs: 915 ms, bbp: 0 ms, qss: 431 ms
Size: 570010 bbs: 932 ms, bbp: 0 ms, qss: 432 ms
Size: 580010 bbs: 933 ms, bbp: 0 ms, qss: 431 ms
Size: 590010 bbs: 965 ms, bbp: 0 ms, qss: 431 ms
Size: 600010 bbs: 1015 ms, bbp: 0 ms, qss: 449 ms
Size: 610010 bbs: 985 ms, bbp: 0 ms, qss: 444 ms
Size: 620010 bbs: 986 ms, bbp: 0 ms, qss: 445 ms
Size: 630010 bbs: 986 ms, bbp: 0 ms, qss: 445 ms
Size: 640010 bbs: 1001 ms, bbp: 0 ms, qss: 445 ms
Size: 650010 bbs: 1016 ms, bbp: 0 ms, qss: 448 ms
Size: 660010 bbs: 999 ms, bbp: 0 ms, qss: 448 ms
Size: 670010 bbs: 1031 ms, bbp: 0 ms, qss: 449 ms
Size: 680010 bbs: 1032 ms, bbp: 0 ms, qss: 448 ms
Segmentation fault (core dumped)
Size: 690010 bbs: 1049 ms, bbp: 0 ms, qss: 500 ms
Segmentation fault (core dumped)
Size: 700010 bbs: 1063 ms, bbp: 0 ms, qss: 501 ms
Size: 710010 bbs: 1063 ms, bbp: 0 ms, qss: 448 ms
Size: 720010 bbs: 1086 ms, bbp: 0 ms, qss: 461 ms
Segmentation fault (core dumped)
Size: 730010 bbs: 1098 ms, bbp: 0 ms, qss: 500 ms
Size: 740010 bbs: 1083 ms, bbp: 0 ms, qss: 445 ms
Size: 750010 bbs: 1115 ms, bbp: 0 ms, qss: 464 ms
Size: 760010 bbs: 1100 ms, bbp: 0 ms, qss: 464 ms
Size: 770010 bbs: 1115 ms, bbp: 0 ms, qss: 465 ms
Size: 780010 bbs: 1115 ms, bbp: 0 ms, qss: 465 ms
Size: 790010 bbs: 1201 ms, bbp: 0 ms, qss: 461 ms
Size: 800010 bbs: 1149 ms, bbp: 0 ms, qss: 482 ms
Size: 810010 bbs: 1164 ms, bbp: 0 ms, qss: 465 ms
Size: 820010 bbs: 1185 ms, bbp: 0 ms, qss: 461 ms
Size: 830010 bbs: 1185 ms, bbp: 0 ms, qss: 461 ms
Size: 840010 bbs: 1185 ms, bbp: 0 ms, qss: 461 ms
Size: 850010 bbs: 1200 ms, bbp: 0 ms, qss: 464 ms
Size: 860010 bbs: 1215 ms, bbp: 0 ms, qss: 464 ms
Size: 870010 bbs: 1215 ms, bbp: 0 ms, qss: 494 ms
Size: 880010 bbs: 1236 ms, bbp: 0 ms, qss: 465 ms
Size: 890010 bbs: 1301 ms, bbp: 0 ms, qss: 482 ms
```

Figure 7, segmentation faults caused by multithread. bbs is the thrust result, and qss is the multithreaded result. Bbp is single-threaded result and takes 0 ms because it is disabled when running up to 1,000,000 tuples. The reason for this is the n^2 performance of bubble sort.

Both implementations resulted in a small number of segmentation faults when testing between 10 and a million integers incremented by ten thousand. The cause of the segmentation faults were unknown, occurred only a small fraction of the trials,, and did not occur for array sizes less than 10,000, though not many seeds were tested.

Appendix A

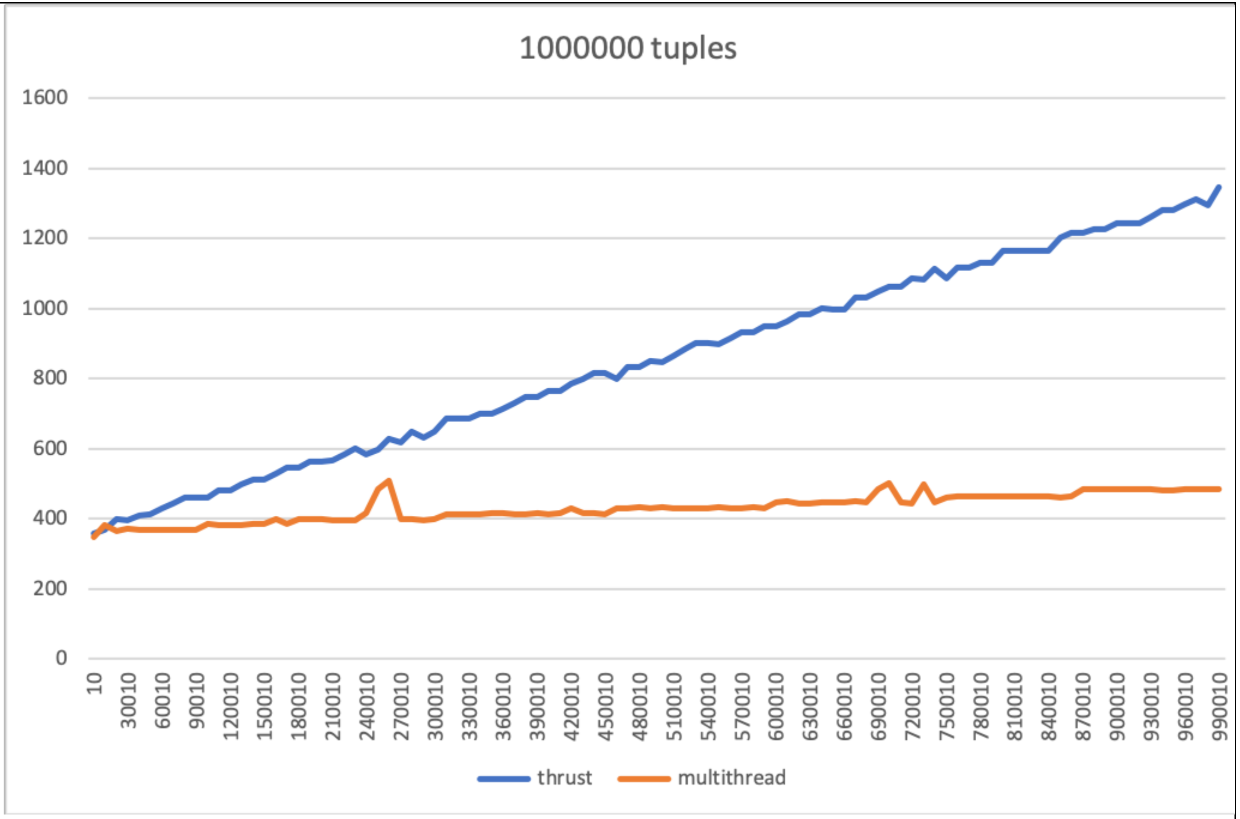


Figure 8, comparison between the thrust algorithm and the multithreaded algorithm, performed on up to 1,000,000 integers. Time in milliseconds (y-axis) vs number of integers (x-axis).

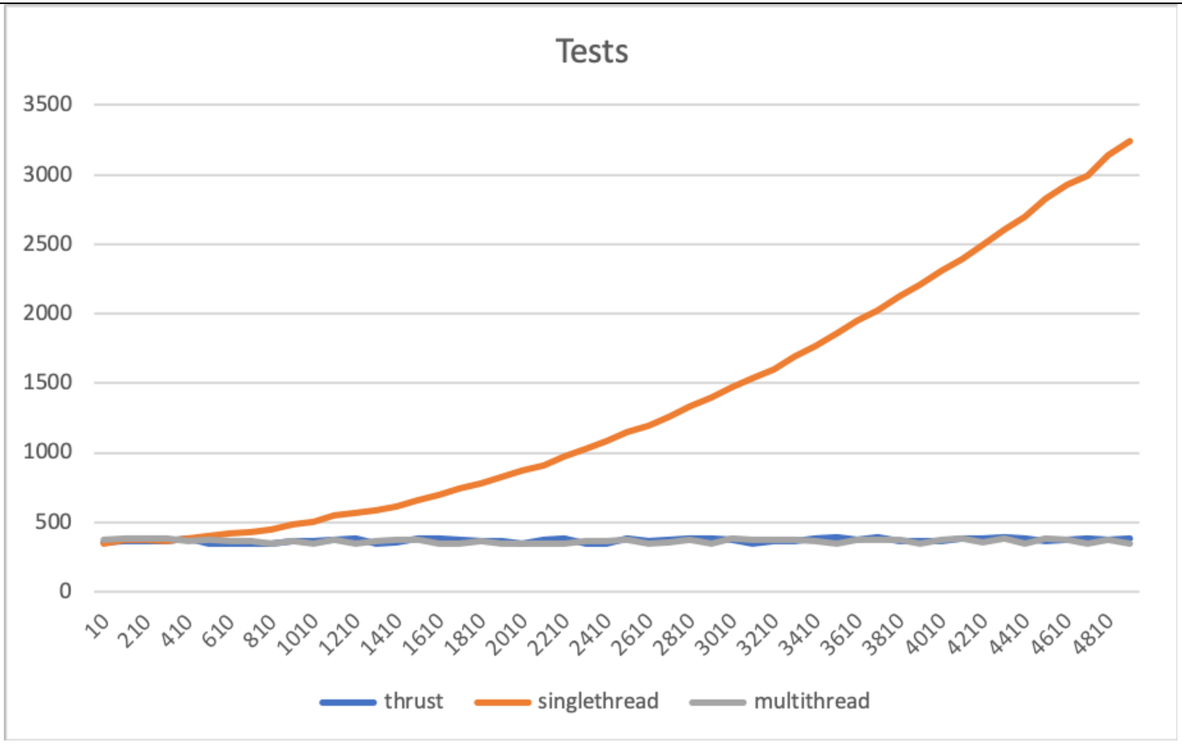


Figure 9, comparison of all 3 algorithms. Time in milliseconds (y-axis) vs number of integers (x-axis).

Appendix B

```
// get the random numbers
        array = makeRandArray( size, seed );

        cudaEvent_t startTotal, stopTotal; float timeTotal; cudaEventCreate(&startTotal);
cudaEventCreate(&stopTotal); cudaEventRecord( startTotal, 0 );

        ///////////////////////////////////////////////////////////////////
        /////////////////////// YOUR CODE HERE      ///////////////////////
        ///////////////////////////////////////////////////////////////////

        thrust::sort(array, array + size);

        /*********************************
         *
         Stop and destroy the cuda timer
         *********************************/
        cudaEventRecord( stopTotal, 0 );
        cudaEventSynchronize( stopTotal );
        cudaEventElapsedTime( &timeTotal, startTotal, stopTotal );
        cudaEventDestroy( startTotal );
        cudaEventDestroy( stopTotal );
        /**********************************
         end of cuda timer destruction
         **********************************/
```
Figure 10, implementation of Thrust sort

```
__global__ void matavgKernel(int * array, int size ) {

                for(int i = 0; i <= size - 1; i ++)
                {

                        for(int j = 1; j <= size - 1; j ++)
                        {


                                if(array[j] <  array[j - 1])
                                {

                                        int c = array[j - 1];

                                        array[j - 1] = array[j];

                                        array[j] = c;


                                }//end if

                        }//end for j

                }//end for i

        }//end function
```

Figure 11, bubble sort.

```
Main()
{

        array = makeRandArray( size, seed );

        int * host_array = (int*)malloc(size * 4);
```

```
	for(int i =0; i <= size - 1; i ++)
	{

		host_array[i] = array[i];

	}//end for i

	cudaEvent_t startTotal, stopTotal; float timeTotal; cudaEventCreate(&startTotal);
cudaEventCreate(&stopTotal); cudaEventRecord( startTotal, 0 );

	/////////////////////////////////////////////////////////////////////
	//////////////////////  YOUR CODE HERE      //////////////////////
	/////////////////////////////////////////////////////////////////////


	int * cuda_array;

	cudaMalloc(&cuda_array, size * 4);

	cudaMemcpy(cuda_array, host_array, size * 4, cudaMemcpyHostToDevice);

	matavgKernel <<< 1, 1 >>> (cuda_array, size);

	cudaMemcpy(host_array, cuda_array, size * 4, cudaMemcpyDeviceToHost);

	cudaFree(cuda_array);

	/***********************************
	 *
	 Stop and destroy the cuda timer
	 **********************************/
	cudaEventRecord( stopTotal, 0 );
	cudaEventSynchronize( stopTotal );
	cudaEventElapsedTime( &timeTotal, startTotal, stopTotal );
	cudaEventDestroy( startTotal );
	cudaEventDestroy( stopTotal );
	/***********************************
	  end of cuda timer destruction
	 **********************************/
}
```

Figure 12, Implementation of singlethread.

```
__global__ void matavgKernel()
{

                  int i = threadIdx.x + blockDim.x * blockIdx.x;
                  int j = threadIdx.y + blockDim.y * blockIdx.y;

                  int threads_on_a_side = (blockDim.x * blocks_on_a_side);

                  int current = i + (j * threads_on_a_side);


                  if(bucket_starts[current] != -1)
                  {

                           bubble_sort(array_of_buckets, size, bucket_starts[current],
bucket_finishes[current]);

                  }//end if


         }//end function
```

Figure 13, kernel in multithreaded program.

```
for(int i = 0; i <= size - 1; i ++)
        {
                 int bucket = ((double)array[i] / (double)(max_value
+ 1)) * number_of_buckets;

array_of_buckets[bucket][bucket_counts[bucket]] = array[i];

                 bucket_counts[bucket] ++;

        }//end for i
```

Figure 14, bucket assignment.

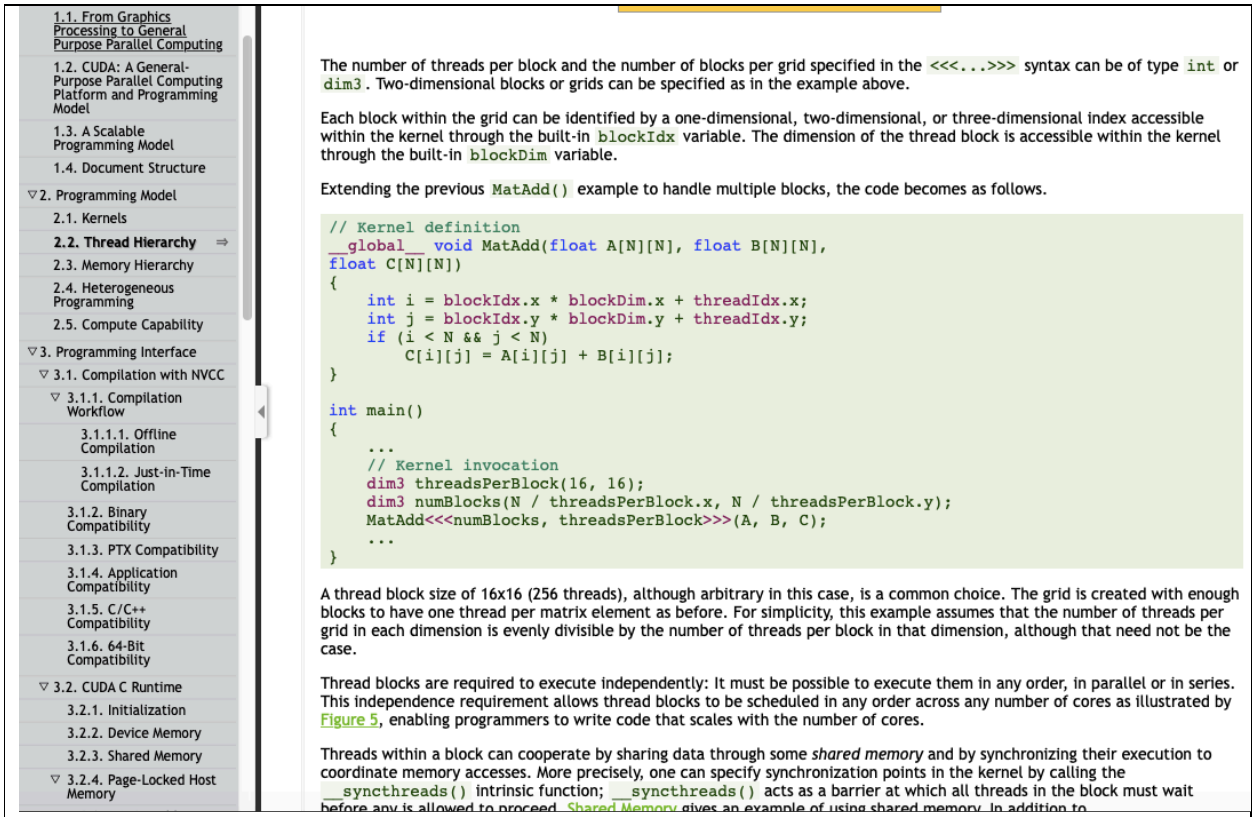
Code Reuse

I’m not sure I used this, but I had written down this link in the code that I had looked at:

https://stackoverflow.com/questions/6419700/way-to-verify-kernel-was-executed-in-cuda

Code was reused from the Nvidia tutorial on use of CUDA, specifically the use of i, j, numBlocks and threadsPerBlock as well as the declaration of dim3 variables. See Figure 15.

https://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html



The number of threads per block and the number of blocks per grid specified in the `<<<...>>>` syntax can be of type `int` or `dim3`. Two-dimensional blocks or grids can be specified as in the example above.

Each block within the grid can be identified by a one-dimensional, two-dimensional, or three-dimensional index accessible within the kernel through the built-in `blockIdx` variable. The dimension of the thread block is accessible within the kernel through the built-in `blockDim` variable.

Extending the previous `MatAdd()` example to handle multiple blocks, the code becomes as follows.

```
// Kernel definition
__global__ void MatAdd(float A[N][N], float B[N][N],
float C[N][N])
{
    int i = blockIdx.x * blockDim.x + threadIdx.x;
    int j = blockIdx.y * blockDim.y + threadIdx.y;
    if (i < N && j < N)
        C[i][j] = A[i][j] + B[i][j];
}

int main()
{
    ...
    // Kernel invocation
    dim3 threadsPerBlock(16, 16);
    dim3 numBlocks(N / threadsPerBlock.x, N / threadsPerBlock.y);
    MatAdd<<<numBlocks, threadsPerBlock>>>(A, B, C);
    ...
}
```

A thread block size of 16x16 (256 threads), although arbitrary in this case, is a common choice. The grid is created with enough blocks to have one thread per matrix element as before. For simplicity, this example assumes that the number of threads per grid in each dimension is evenly divisible by the number of threads per block in that dimension, although that need not be the case.

Thread blocks are required to execute independently: It must be possible to execute them in any order, in parallel or in series. This independence requirement allows thread blocks to be scheduled in any order across any number of cores as illustrated by Figure 5, enabling programmers to write code that scales with the number of cores.

Threads within a block can cooperate by sharing data through some *shared memory* and by synchronizing their execution to coordinate memory accesses. More precisely, one can specify synchronization points in the kernel by calling the `__syncthreads()` intrinsic function; `__syncthreads()` acts as a barrier at which all threads in the block must wait before any is allowed to proceed. Shared Memory gives an example of using shared memory. In addition to

Figure 15, code that was reused from CUDA tutorial.

Bibliography

[1] N. Bell and J. Hoberock, "Thrust: A Productivity-Oriented Library for CUDA," in *GPU Computing Gems Jade Edition*, Morgan Kaufmann Publishers, 2011.

[2] T. H. Cormen, C. E. Leiserson, R. L. Rivest and C. Stein, Introduction to Algorithms Third Edition, Cambridge: Massachusetts Instittue of Technology, 2009.

[3] N. Satish, M. Harris and M. Garland, "Designing Efficient Sorting Algorithms for Manycore GPUs," NVIDIA Corporation, 2008.