Project – Tour d'Algorithms: Cuda Sorting Championship

CS 516, Department of Computer Science, SIUE
Course Name: Computer Architecture
Instructor: Dr.Mark McKenney

Group Members:
Prashanna Raj Pandit
*
Asha Shah
†
Simran Basnet
‡

November 30, 2024

*ppandit@siue.edu
†ashshah@siue.edu
‡sibasne@siue.edu

Contents

1 Introduction

1.1 Overview of GPU Computing and Project

CUDA or "Compute Unified Device Architecture" is a parallel computing platform and programming model developed by NVIDIA, targeting general-purpose computing on GPUs. With CUDA's flexibility, programs that will execute over hundreds, thousands, or millions of threads in parallel could potentially speed sorting tasks significantly by parallelizing workloads across thousands of GPU cores- making sorting, for instance, particularly faster. Parallelism in sorting algorithms extremely reduces execution time in contrast to the ordinary single-threaded CPU implementations of such algorithms, thus rendering GPUs immensely powerful in managing the sorting of large-scale datasets.
We have been experimentally investigating three sorting algorithms using massively parallel computing under the Cuda programming model. First is the sort algorithm implemented by the Cuda Thrust Library and has the source code in $thrust.cu$. This demonstrates how good the GPU can perform operations. The second one is a sorting algorithm that runs on the GPU, which has the source code in $singlethread.cu$. Here we have used very quick sorting algorithm to implement the single thread version. The third one is a multi-threaded sorting algorithm, which essentially demonstrates parallel sorting with merge sort and has source file in $multithread.cu$. These three can be run with the following commands.

Login to radish server and run the Makefile by make command. *$make*.It automates the process of compiling and linking source files into executable programs or libraries by reading a Makefile, which contains rules and dependencies for building the project.

./executablename [array size] [seed value] [1 to print the sorted array, 0 otherwise]

For example, to execute multi-threading;

./multithreading 180000 42 1

It will execute the program multithreading with three command-line arguments: 180000, 42, and 1. The data are collected in similar fashion with different array size with fixed seed value of 42 and graphs are plotted using $plotly.express$ in Google Colab.

1.2 Objective

The idea behind this assignment is to construct and evaluate three sorting algorithms, namely a baseline sort using CUDA Thrust, a single-threaded sorting algorithm to practice CUDA syntax, and massively parallel sorting to take advantage of their complete parallel computing capabilities concerning the GPU. The basic idea involves comparing these algorithms against each other so that one might see the affordances as well as challenges of sorting using the GPU concerning performance, memory access, and parallel processing.

2 Sorting Algorithm Implementations

2.1 Thrust Sort (Baseline)

2.1.1 Algorithm Description

The CUDA Thrust library's Thrust Sort algorithm is a lot of optimized GPU-based sorting implementations that work much like C++ Standard Template Library (STL). This enormously allows parallelism by essentially dividing the sorting task into multiple GPU thread processes for very beneficial results in terms of productivity on huge datasets. Similar to quicksort or mergesort depending on the internal optimizations of the library, the thrust::sort function is based on the comparison-based sorting algorithm. Thus, in this implementation, the algorithm sorts an array made of random integers generated on the device, creating a baseline by which the performance of the GPU can be evaluated.

2.1.2 Time Complexity

The theoretical time complexity of Thrust Sort is $O(n \log n)$.

2.1.3 Implementation Details

The $thrust :: device\_vector < int >$ makes a random array of integers, which is then sorted with the $thrust :: sort$. The memory is then allocated and managed directly on the GPU so that no manual memory management has to be done using the CUDA APIs. So, the program accepts command-line arguments, such as the size of the array, the seed value to be used for random number generation, and an option to print the sorted array. The actual sort operation takes place while it is being timed with CUDA events that record and compute the total execution time for the entire sort operation so that precise measurements can be made.

2.1.4 Performance

Array Size (x)	Execution Time (y)
1000	0.0000624
10000	0.0002478
20000	0.0002498
40000	0.0002509
60000	0.0002601
80000	0.0002605
100000	0.0002785
120000	0.0002887
140000	0.0003062
160000	0.000297
180000	0.00028365

Table 1: Thrust Execution time for Different Array Size

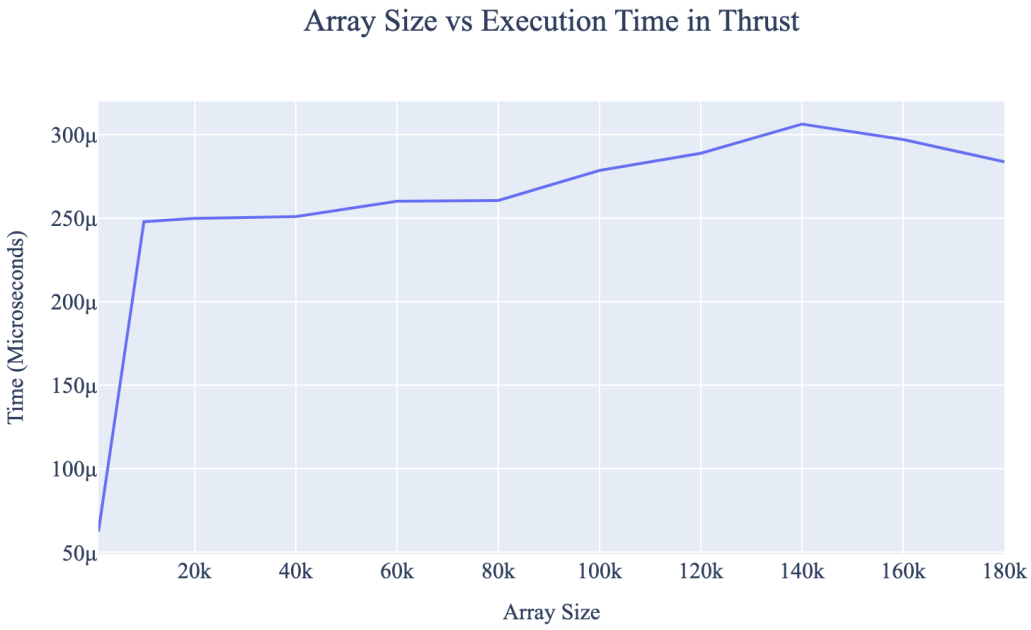

Figure 1: Figure illustrating the relationship between array size and execution time (microseconds) in Thrust

The performance data of the Thrust sorting algorithm demonstrates its high efficiency in handling large datasets on a GPU. As the input size increases from 1,000 to 180,000 elements, the execution time initially scales linearly, with a noticeable increase between 1,000 and 10,000 elements, where the execution time jumps from 0.0000624 seconds to 0.0002478 seconds. Beyond 10,000 elements, the execution time stabilizes, indicating efficient parallelization, with only marginal increases up to 180,000 elements. Notably, the execution time plateaus around 0.00025–0.00030 seconds, showcasing Thrust's ability to handle larger inputs with minimal overhead. The slight fluctuations, particularly after 100,000 elements, can be attributed to memory access patterns, GPU synchronization overhead, or hardware limitations, yet the overall performance remains stable, reflecting the algorithm's scalability and optimized parallel execution on GPU hardware.

2.2 Single-Threaded Sorting Algorithm

2.2.1 Algorithm Description

In sorting algorithms, this QuickSort is a divide-and-conquer recursive technique that partitions the input array into increasingly smaller subarrays around one of its elements called a pivot in every case, placing smaller elements than the pivot on its left and larger ones on its right. This partitioning process would repeat recursively on each previously defined partition until all related pieces would be sorted in a single stage. The algorithm can carry out such implementation iteratively with the help of a stack. As a stack stores the references of the subarrays to be sorted like the recursive function does not need to call recursively, memory resources can be effectively controlled without recursion.

2.2.2 Time Complexity

The time complexity of QuickSort depends on the selection of the pivot and the resulting partitioning. The average-case time complexity is $O(nlogn)$.

2.2.3 Implementation Details

The specified implementation is written in C++ and memory is managed by CUDA but is single-threaded. A random integer population is first followed by transferring the array to the GPU device memory to launch the iterative QuickSort kernel with a single thread and block $(<<< 1, 1 >>>)$. The kernel uses a stack to partition the subarrays and perform partitioning by comparison and swapping elements with respect to a pivot. After sorting, the array is transferred back from the host while measuring the execution time using CUDA events for more accurate timing.

2.2.4 Performance

The execution time of this single-threaded QuickSort implementation displays a monotonically increasing function as input size increases. The execution time, at its lowest for an array size of 1000 elements, is 0.001325 seconds. To test the maximum array size of 180,000 elements, it is increased to 0.296904 seconds. The linear increase corresponds to low resource utilization on the GPU since the fast or slow algorithm runs in a single-threaded mode. While small input sizes seem to work fine, it does not utilize parallel processing power from the GPU, which results in performance not exceeding limits for large input sizes. Organizing a multi-threaded or multi-parallel version of QuickSort will make the load distributed over the multiple threads, and hence execution time will be less while increasing the scalability.

Array Size (x)	**Execution Time (y)**
1000	0.001325
10000	0.013899
20000	0.029355
40000	0.062702
60000	0.097758
80000	0.132630
100000	0.161812
120000	0.202545
140000	0.232674
160000	0.257092
180000	0.296904

Table 2: Single-Thread Execution Time for Different Array Sizes

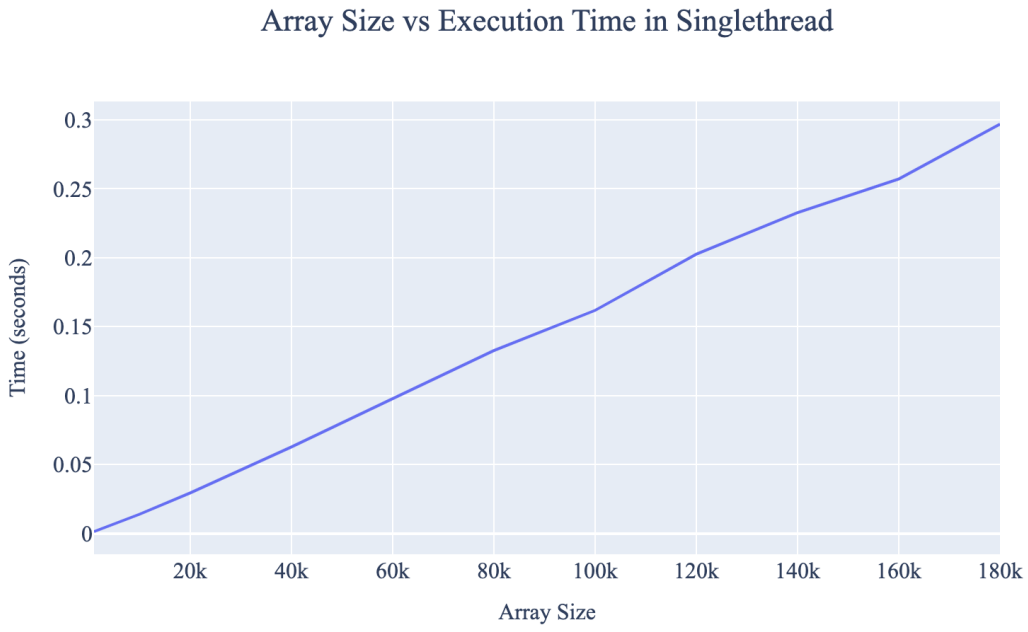

Figure 2: Figure illustrating the relationship between array size and execution time (microseconds) in Singlethread

2.3 Multi-Threaded Sorting Algorithm

2.3.1 Algorithm Description

The algorithm that is instantiated in this code is a parallelized Merge Sort algorithm using CUDA in sorting an array of integers. The actual algorithm of merge sort goes on dividing the same array into halves recursively until each segment is one single element. After that, it merges all of them into one sorted array. The CUDA version parallelizes the merge sort such that each thread performs the merge operation on a subarray of the original array. The kernel will do the merging for increasingly larger subarrays, simulating the divide-and-conquer method. Memory is allocated on both the host (CPU) and device (GPU). Data is copied to the device, processed in parallel, and copied back to the host for the final sorted array.

2.3.2 Time Complexity

The time complexity of the merge sort algorithm is $O(nlogn)$.

2.3.3 Implementation Details

Above everything, the implementation uses CUDA as a means of parallelization for the merge sort algorithm. A kernel function is invoked to merge the arbitrarily sized subarrays in parallel, with each thread finding itself associated with parts of the complete array. The merging is done in an iteration loop over the whole array using the divide-and-conquer method. First, the array

is divided into several smaller chunks, and then progressively in parallel, the merging occurs. Memory management is critical in this implementation; the two arrays are allocated on the host and device. Data moves from host to device before computation and once again after sorting, back to the host. There are also CUDA-specific checks that facilitate the smooth execution of operations on the devices.

2.3.4 Performance

Array Size (x)	Multi-Thread Time (y2)
1000	0.00054476
5000	0.0019128
10000	0.0034713
15000	0.0041022
20000	0.0087966
25000	0.010496
30000	0.0116512

Table 3: Execution Time for Multi-Thread for Different Array Sizes

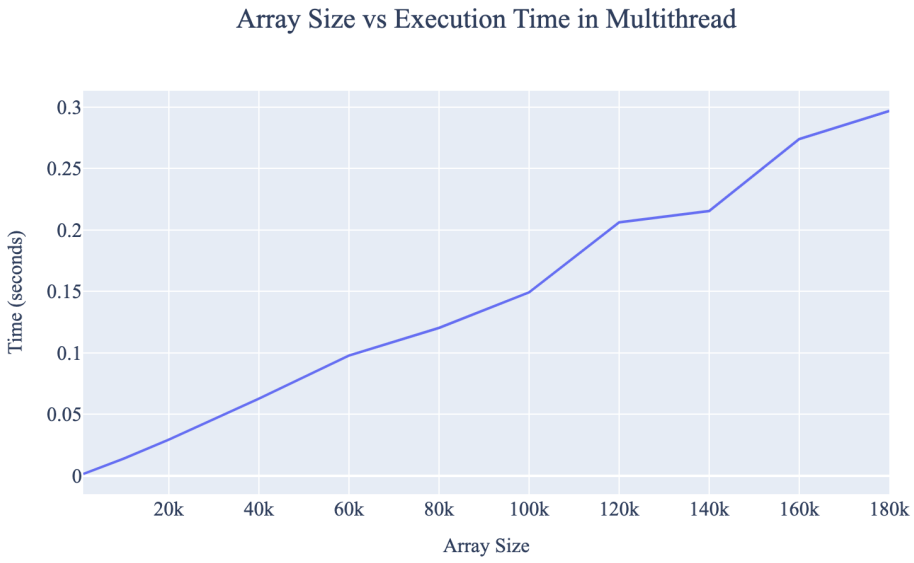

Figure 3: Figure illustrating the relationship between array size and execution time (microseconds) in Multithread

The performance results in terms of execution time (in seconds) for increasing array sizes reflected that the developed algorithm really applies to the parallelized merge sort. The execution time varies with the increase in the size of the array, although significant improvement is recorded when compared to a single-threaded implementation due to parallelization. The time complexity is $O(nlogn)$ in theory but can be affected by what is mentioned above in terms of memory access

speed, block and thread configurations, and GPU architecture configurations. Thus, results show an improvement in performance with time execution being lower for an increase in array size, which is typical when CUDA achieves parallelization working multiple threads at the same time on different portions of the data to allow much faster sorting of large datasets.

3 Overall Performance Evaluation

Array Size (x)	Multi-thread (y1)	Single-thread (y2)	Thrust (y3)
1000	0.00054476	0.0014336	0.0000624
5000	0.0019128	0.00670515	0.0002529
10000	0.0034713	0.013929	0.0002478
15000	0.0041022	0.021423	0.0002508
20000	0.0087966	0.029346	0.0002498
25000	0.010496	0.039978	0.0002457
30000	0.0116512	0.045236	0.00025699

Table 4: Execution Time for Different Array Sizes and Algorithms

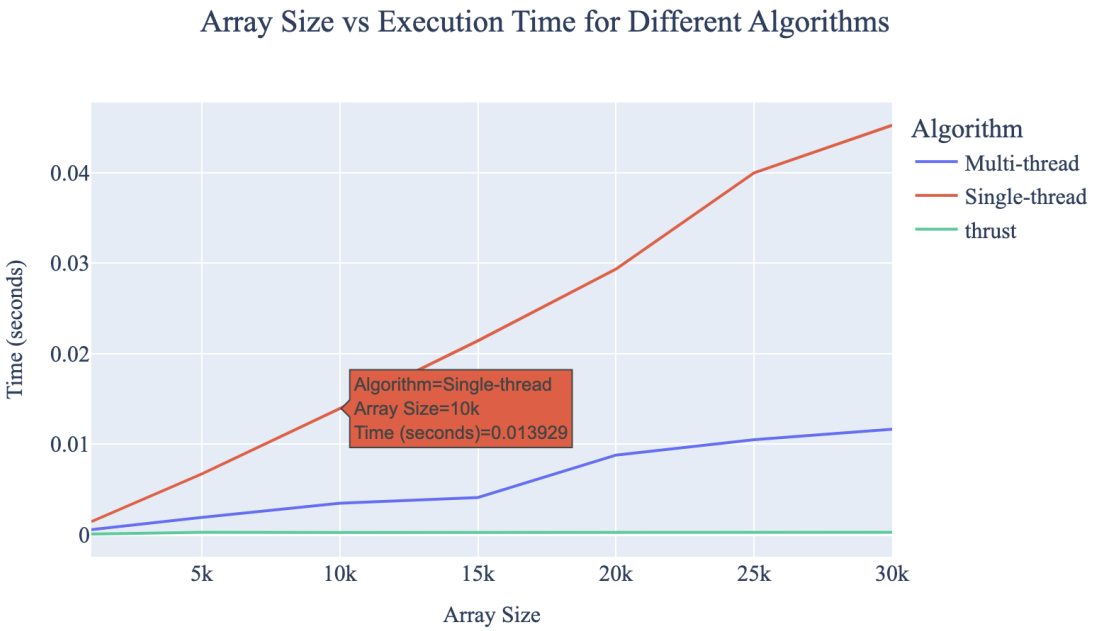

Figure 4: Figure illustrating the overall performance comparison of thrust, single-thread, and multi-thread

Performance comparison reveals that Thrust absolutely dominates both multi-thread and single-thread approaches within performing execution in microseconds which create a similar performance across multiple array sizes. Multi-threading is significantly better than single-threading but

still cannot keep up to the level reached by Thrust, particularly in larger array sizes. While single-thread execution becomes progressively slower as the array size grows, multi-thread execution increases in time at a slower rate, but never reaches the same efficiency as Thrust. As a whole, the most efficient method among them is Thrust, followed by multi-threading, and single-threading exposed the least scalability.

4 Conclusion

With all this said, the parallelized CUDA merge sort dramatically enhances sorting performance, more pronouncedly for larger data sizes. This algorithm based on GPU parallelism is able to significantly decrease the busy scatter of time by direct execution using the traditional, serial merge sort. The time savings become even more impressive as the input size increases. This showcases the efficiency of parallel computing regarding what always seems to be very computation-heavy tasks, thereby fundamentally advancing the way huge-scale data can be sorted effectively.

5 References

1. The CUDA C Programming Guide, *NVIDIA*, `http://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html`

2. The Thrust Documentation, *NVIDIA*, `http://docs.nvidia.com/cuda/thrust/`