Project - Tour d'Algorithms: Cuda Sorting Championship

CS-516

Computer Architecture

SIUE

Submitted by:

Krishna Khanal-800771289

Sushrit Kafle-800792522

Venkatesh Bollineni-800792618

Table of Contents

1. Thrust

Thrust sort algorithm description:

Thrust is a library in C++ programming language that consists of various parallel algorithms and data structures. Thrust sort is a sorting algorithm within the library that improves efficiency in sorting large datasets using the power of GPUs. In this program, we sorted a randomly generated array of numbers using the Thrust library for GPU parallel computing. **thrust :: sort** function is used to efficiently sort the array in ascending order. This sorting algorithm takes advantage of parallel processing on the GPU, which makes it suitable for handling large datasets.

Time complexity:

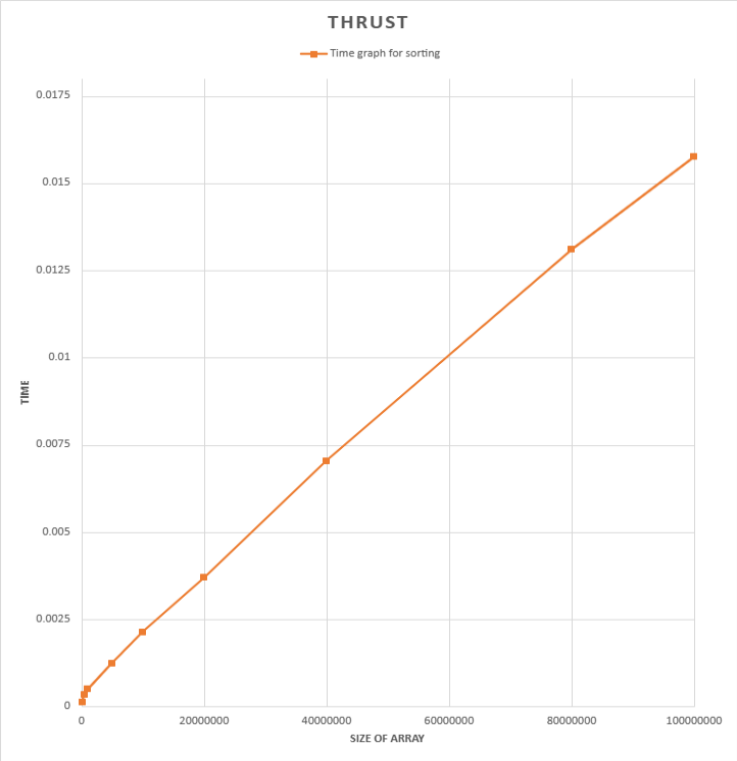
Although it is not specified in NVIDIA documentation, the time complexity of the sorting operation is assumed to be O(Nlog(N)) based on its behavior in the graph, where N represents the number of elements in the array.

Implementation details:

Thrust sort sorts by using the GPUs with many cores that work simultaneously. It divides the sorting work among these cores, making the computation faster. Thrust sort has ability to adapt to different array sizes during runtime by managing the varying lengths through dynamic sizing and memory allocation. Thrust simplifies GPU programming by providing a high-level path to write parallel algorithms without necessarily understanding the details of CUDA. It can handle different data types and array sizes for various sorting tasks. As integrated into CUDA, Thrust simplifies GPU programming by producing CUDA code which fits to the GPU's architecture.

Performance:

Thrust	
Array Size	Time for sorting
100000	0.000141312
500000	0.000361504
1000000	0.00050288
5000000	0.00125542
10000000	0.00214016
20000000	0.00370995
40000000	0.00704624
80000000	0.0131062
100000000	0.0157696



The result shows the time took to sort various sizes of arrays using the CUDA Thrust library. Larger arrays took more time because of computer's memory process and data movement in between the CPU and GPU. It can be observed that the thrust sort can work very fast and is able to sort larger array in very short time. From the figure it can be observed that it sorts array of size 100 million under 0.0175 seconds, which is very fast. Thus, it can be clearly observed that. **thrust :: sort** uses the GPU and its parallel computing abilities very efficiently and is able to handle large and complex set of array and sort them rapidly.

2. Single-Thread

Quick sort algorithm description:

The code implements the Quick-sort algorithm in a serial fashion using CUDA, a parallel computing platform developed by NVIDIA. In general, Quicksort is a divide-and-conquer sorting algorithm that works by selecting a pivot element and partitioning the array into two subarrays such that elements less than the pivot are on the left, and elements greater than the pivot are on the right. In this case we have applied quicksort algorithm using an explicit stack and iteration to sort the array in the CUDA kernel itself.

The Quicksort algorithm using a stack is started by creating an empty stack and pushing the initial boundaries of the entire array, which are represented as the left and right indices. In the main loop of the algorithm, it continues its iterative operation as long as the stack remains non-empty. During each iteration, the algorithm retrieves the left and right boundaries of the current subarray by popping the top two elements from the stack. These boundaries guide the further steps of selecting a pivot, partitioning the subarray, and identifying the new boundaries for the resulting partitions. A pivotal element is then selected, and the subarray is partitioned, arranging elements such that which are less than the pivot is on the left and which are greater on the right. The boundaries of the resulting right and left partitions are pushed onto the stack, ensuring the correct order of pushing – the right partition precedes the left. This process repeats until the stack is empty, showing the completion of sorting for all subarrays. Finally, at this point, the original array is fully sorted.

Time complexity:

The time complexity of quicksort algorithm using a stack is generally the same as the recursive quicksort algorithm, which is O(nlogn) on average. This complexity comes from the fact that on average each element is compared to the pivot only logn times, and there are total n elements in the array.

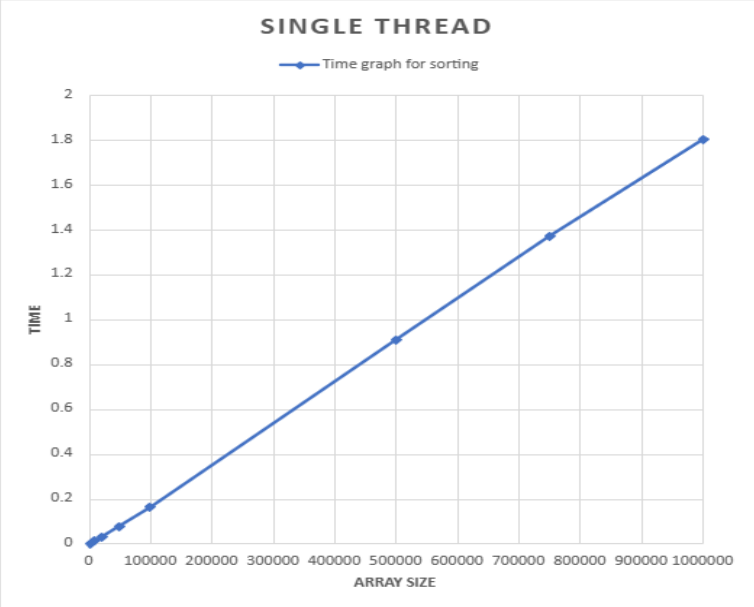
Implementation details:

The CUDA kernel for quicksort uses only one block and kernel processing on the GPU for sorting the subarrays. As this is a single thread sorting, this code does not take advantage of the performance enhancing capabilities CUDA and the GPU.Also, in our case, the implementation used an explicit stack data structure, int stack[32], to keep track of subarrays that are need to be

sorted as size of the input array is not known at compile time.. The size of this stack is fixed which is 32 in our case, which provides a limit on the depth of recursion or the number of subarrays that can be sorted parallely. The fixed stack ensures that the GPU's limited memory is used efficiently, avoiding the need for dynamic memory allocation and deallocating the memory during sorting.

Performance:

Single Thread	
Array Size	Time for sorting
1000	0.00126554
10000	0.0140503
20000	0.0296162
50000	0.0790614
100000	0.162315
500000	0.908144
750000	1.36941
1000000	1.80127



The graph shows that sorting times are increasing when the array size increases which is probably due to memory access becoming less efficient as more cache misses happen, consequently increasing latency. As GPU architecture is designed for parallelism, this single-threaded implementation has not fully shown its potential and has contributed very little improvement in

the compiling time as the array size increases. The inefficiency of this algorithm is clear when compared to the thrust:: sort as this algorithm take 1.8 sec for 1million sized array while thrust::sort took 0.0005 sec only.

3. Multi-Thread

Merge sort algorithm description:

Inorder to implement a multithread sorting algorithm, we have chosen to implement a merge sort. Merge sort is a very efficient sorting algorithm which also uses a divide and conquer principle. In simple term, merge sort separates the array to two parts, sorts it and merges it . We have used CUDA programming with **mergeSort** kernel where we use a unique global index (tid) and allocate threads so that the merging can be done parallelly.

The **mergeSort** kernel operates on the GPU to sort a specific section of the array assigned to each thread by calculating a unique global index (tid) for every thread which involves considering its index within the block and multiplying it by the block size. The algorithm is an iterative merge sort where it repeatedly merges adjacent subarrays of increasing size. The **mergeSort** kernel is launched on the GPU with the appropriate block and thread configurations, allowing for parallel sorting of the array. After the completion of GPU processing, the sorted array is copied back from the device to the host, ensuring a seamless integration of GPU-accelerated merge sorting into the overall algorithmic flow.

Time complexity:

The time complexity of the merge sort algorithm is O(n log n), where n represents the number of elements in the array. This is separated as recursive divide of array in O(log n) time and O(n) comparisons of elements.

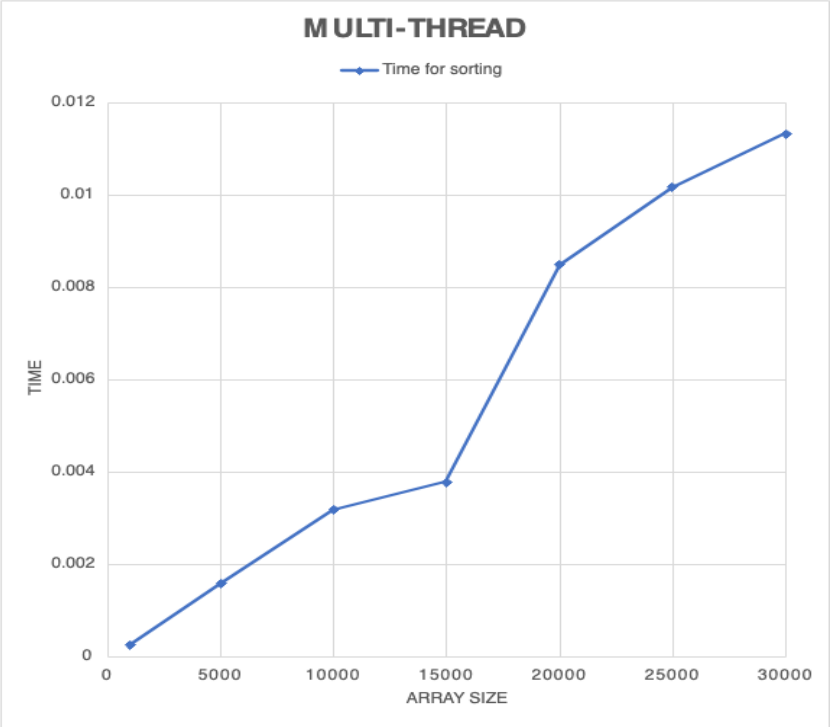
Implementation details:

The main part of the algorithm is in the mergeSort kernel, which operates on individual elements of the input array. where each thread calculates its unique global index, and the algorithm performs an iterative merge sort for various subarray sizes. The merging process involves comparing elements from left and right subarrays and stores them in a temporary array. The main function

initializes the input array, calls the GPU-based merge sort, measures the execution time, and prints the sorted array. It also checks for errors in CUDA function calls and offers information about the size of the GPU memory that has been allocated.

Performance:

Array size	Time for sorting
1000	0.000257024
5000	0.00159334
10000	0.0031857
15000	0.0037929
20000	0.00849318
25000	0.0101767
30000	0.0113371



The figure clearly shows the power of CUDA programming as the arrays are sorted very quickly. There can be seen a slight bent in the graph and this could be due to the parallelism overhead. The graph depicts a nlogn time complexity and sorts array of 30000 in less than 0.012 seconds. The

performance of multithread is even evident in the graph between multithread and single thread performance for the same size of array. Single thread's time increases rapidly while multithread looks slightly linear compared to single thread. This is due to the fact that CUDA efficiently divide the array to kernels and blocks allowing process to run concurrently.