Project: 2: Tour d'Algorithms: Cuda Sorting Championship

Title: Project 2
Name: Sah, Rohit Kumar 800819633 `rohsah@siue.edu`
Karna, Atulesh Chandra 800675898 `akarna@siue.edu`

1 Introduction

1.1 Project Overview

This project, titled Tour d'Algorithms: Cuda Sorting Championship, focuses on investigating parallel programming using Cuda, particularly in the context of sorting algorithms. The goal is to compare the performance of single-thread and multi-thread implementations of the most basic sorting algorithm Bubble Sort on randomly generated numbers, providing insights into how parallel programming impacts computational efficiency.

The project involves creating both single-thread and multi-thread implementations for Bubble Sort algorithm, alongside a reference implementation using sort by cuda thrust library to benchmark the results. The performance of these algorithms is evaluated by measuring the execution times across varying input sizes are studied.

1.2 Hardware Overview

The hardware setup for the systems used in this project includes a single server. The *radish.cs.siue.edu*, has 4 cores with 2 threads per core, resulting in 8 virtual CPUs (vCPUs). Most importantly it is powered by Nvidia Titan GPU that handles the cuda processing part. This system and hardware configurations are enough for testing the multi-thread efficiency of sorting algorithms with Cuda.

2 Methodology

2.1 Experimental Setup

To evaluate performance, the project generated random arrays of integers, where the size of the arrays was varied from small to large. The random arrays were generated using the command-line arguments for array size and seed value. Each program was compiled into separate executables, and the execution time was measured.

Each executable took two command-line arguments:

```
[executable name] [number of random integers to generate] [seed value for random number
    generation] [1 for printing logs and 0 for log-less execution]
```

Then, a bash script was written to build and execute all the executable and write the runtime of each into a single CSV file.

Parallelization was achieved using CUDA, with each sorting algorithm being parallelized based on its structure. Execution times were measured using the CUDA Events, ensuring accuracy. Performance was compared for each algorithm in multi-thread, single-thread, and using Thrust's std::sort as a reference.

2.2 Metrics for Comparison

The primary metric used for comparison was execution time in seconds. Tests were performed across a range of input sizes to observe how performance scales with increasing data. The results were plotted in graphs to visualize the difference between serial, parallel, and reference implementations.

3 Thrust Sort

3.1 Algorithm Description

The Thrust library uses optimized algorithms for GPU acceleration, leveraging the internal power of NVIDIA's implementation to sort arrays efficiently. This served as a baseline for comparing custom implementations.

- **Best Case:** O($n\ log\ n$)
- **Average Case:** O($n\ log\ n$)
- **Worst Case:** O($n\ log\ n$)

3.2 Implementation Details

The program utilized the Thrust library's thrust::sort function. The random input with a given seed is used to generated random numbers. Then memory is allocated for both input and output. Then, the thrust::sort is used to sort the array in-place. Then the sorted array is copied back to the host for verification and display.

3.3 Performance

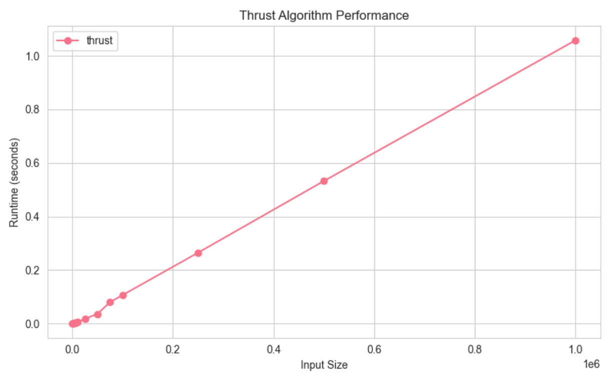

Figure 1: Thrust runtime over Input Size

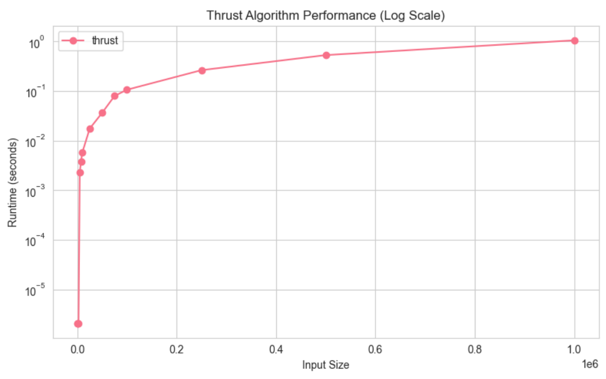

Figure 2: Thrust runtime (log) over Input Size

The running time for arrays of varying sizes (e.g., 250K, 500K elements) shows near-linear scaling relative to O(n log n). Thrust's performance is influenced by efficient memory management and GPU optimizations. Its superior results stem from well-tuned kernels and efficient use of shared memory as it is part of the standard thrust library.

4 Single-threaded GPU Sort

4.1 Algorithm Description

This solution implemented a simple Bubble Sort on the GPU using a single CUDA thread, making it equivalent to a serial implementation but executed on the GPU.

- Best Case: O(n))
- Average Case: O(n^2)
- Worst Case: O(n^2)

4.2 Implementation Details

Input was allocated in device memory. A single CUDA thread performed the Bubble Sort, iteratively comparing and swapping adjacent elements until the array was sorted. While inefficient, this implementation provided valuable practice in CUDA memory management and kernel invocation.

```
__device__ inline void swap(int * x, int * y) {
  int temp = * x;
  * x = * y;
  * y = temp;
}

__device__ void bubbleSort(int arr[], int n) {
  int i, j;
  for (i = 0; i < n - 1; i++)
    for (j = 0; j < n - i - 1; j++)
      if (arr[j] > arr[j + 1])
        swap( & arr[j], & arr[j + 1]);
}
```

4.3 Performance

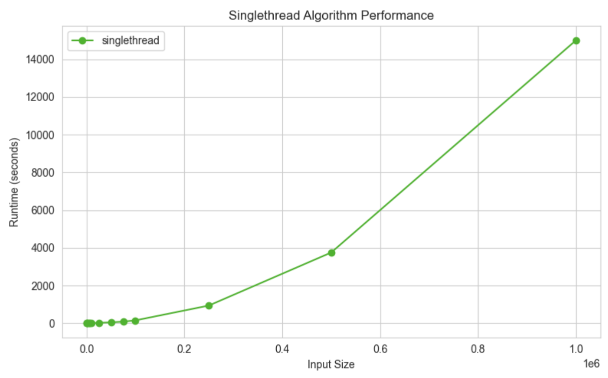

Figure 3: Single-thread runtime over Input Size

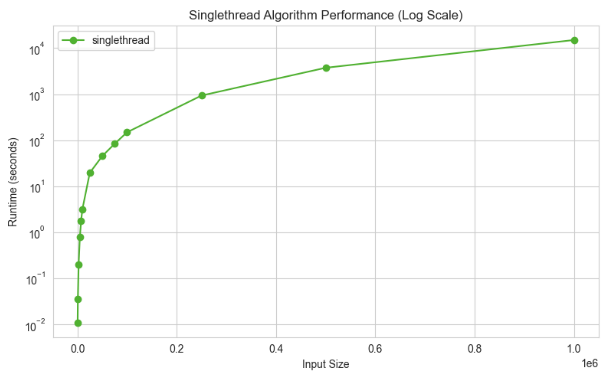

Figure 4: Single-thread runtime (log) over Input Size

Execution time increases quadratically with input size. The quadratic growth highlights the inefficiency of Bubble Sort, worsened by the overhead of GPU memory transfers. GPU's parallel capabilities were underutilized in this approach.

5 Multi-thread GPU Sort

5.1 Algorithm Description

The bubble sort algorithm is adapted for GPU execution by dividing the sorting task into independent operations on pairs of elements, leveraging the CUDA thread hierarchy to execute these operations in parallel. The algorithm utilizes even-odd sorting. During each iteration if the iteration index is even, threads sort pairs at indices (0, 1), (2, 3), ... and if the iteration index is odd, threads sort pairs at indices (1, 2), (3, 4), ... Threads independently compare and swap elements within their assigned pairs. Each thread block synchronizes after sorting its assigned pairs to ensure consistency before proceeding to the next iteration.

- **Best Case:** O(n log n)

- **Average Case:** O($n\ log\ n$)
- **Worst Case:** O($n\ log\ n$)

5.2 Implementation Details

The kernel function bubbleSort executes the bubble sort algorithm in parallel. Each thread identifies the pair of elements it will operate on using *blockIdx*, *blockDim*, and *threadIdx*. Pairs are compared, and elements are swapped if out of order. The *__syncthreads*() function ensures all threads complete their operations before the next iteration begins.

The kernel is configured with a grid and block size determined based on the input array size and device properties. This ensures efficient utilization of available GPU resources. Allocate memory for the array on the GPU using *cudaMalloc* and copy the data from the host to the device. Calculate the grid and block dimensions based on the maximum threads per block (*maxThreads*) and input size (*size*). Launch the bubbleSort kernel with the calculated dimensions and pass the device array. Copy the sorted array back to the host using *cudaMemcpy*.

```
__device__ void swap(int * a, int * b) {
  int tmp = * a;
  * a = * b;
  * b = tmp;
}

__global__ void bubbleSort(int * array, int n) {
  int id = blockIdx.x * blockDim.x + threadIdx.x;
  for (int i = 0; i < n; i++) {
    int offset = i % 2;
    int left = 2 * id + offset;
    int right = left + 1;

    if (right < n) {
      if (array[left] > array[right]) {
        swap( & array[left], & array[right]);
      }
    }
    __syncthreads();
  }
}

bubbleSort << < grdDim, blkDim >> > (d_array, size);
CudaCheckError();

CudaSafeCall(cudaMemcpy(array, d_array, size * sizeof(int), cudaMemcpyDeviceToHost));
CudaCheckError
```

5.3 Performance

The optimal configuration ensures all available threads are utilized, reducing the number of iterations required. While *__syncthreads*() ensures consistency, excessive synchronization can impact performance, especially for larger arrays. The implementation is limited by the number of threads that can be allocated on the GPU. For very large inputs, additional optimizations or hierarchical sorting methods may be necessary. Execution time exhibits linear growth for increasing input sizes, much faster than both single-threaded and Thrust. The massively parallel implementation takes advantage of CUDA cores, distributing work efficiently

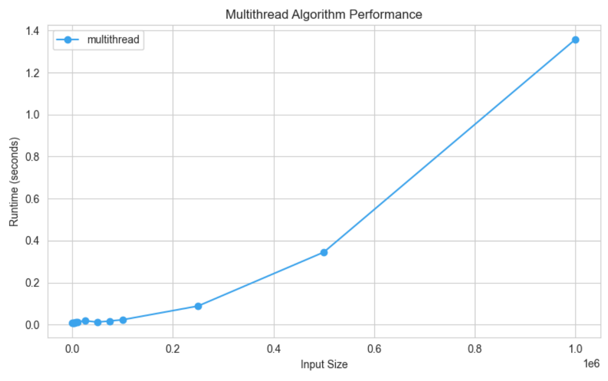

Figure 5: Multi-thread runtime over Input Size

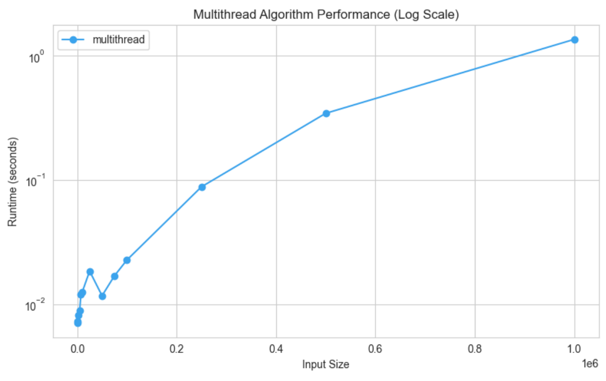

Figure 6: Multi-thread runtime (log) over Input Size

and minimizing memory access delays. The performance is bottle-necked only by global memory bandwidth and thread synchronization.

6 Conclusion

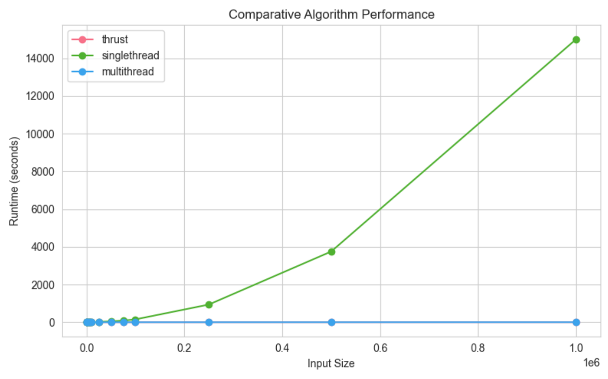

Figure 7: Comparative Algorithm Performance

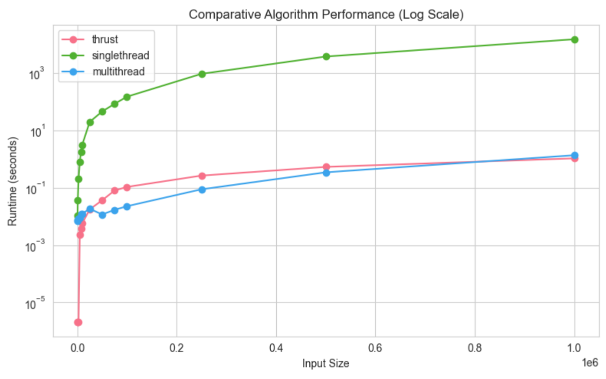

Figure 8: Comparative Algorithm Performance (log)

Thrust, Single-threaded Bubble Sort, Multi-threaded Bubble Sort runtime are plotted against input size in a graph. Single-threaded Bubble Sort performs the worst among these 3 whereas the Multi-threaded Bubble Sort and Thrust Sort have comparable performance. Although bubble sort has quadratic time complexity, when multi-threaded the result is comparable to the thrust results. This actually illustrated how powerful multi-threading can be in GPU Environments.

Appendices

Code Listings

https://github.com/sahrohit/parallel-sorting.git

Runtime Data

https://github.com/sahrohit/parallel-sorting/blob/main/benchmark_results.csv