COMS4040A & COMS7045A Assignment 1 – Report

Akshar Nana
2442321
Coms Hons

28 March 2024

1 K-nearest neighbors problem

The K-nearest neighbor algorithm provides a simple approach to classify unlabeled data using the class of the most similarly labeled data [1]. In this assignment, we consider the following question: Given a set of reference and query points in n-dimensions, which reference points are nearest to which query points? To answer this question, we first look at the brute force approach. A double for-loop is employed. We calculate the distance from every reference point to a query point. We then sort the distance list and take the k-nearest as the result. Repeat the process for every query point. The result of this algorithm is the k-nearest reference points to every query point. In brute force, two components are computationally intensive, i.e., the quick-sort and distance calculation. We can improve the speed by parallelizing both components. The following sections will compare the serial and parallel implementations.

1.1 Test methodology

The test methodology involves a bash script to run the program on a set of parameters. The program leverages the built-in OpenMP timer to measure the execution of the compute-intensive components. The script performs this benchmark for the serial, parallel with task construct, and parallel with section construct. For the reliability of results, the benchmark is performed three times for each configuration. The results are piped to an output file and are then tabulated. These include the breakdown of the total time into the two components. The experiment aims to investigate the difference in performance of the two parallel implementations over the serial implementation. It accomplishes this through a controlled environment, i.e., the hardware used to run the experiments remains unchanged.

1.2 Experiment Setup

Experiment setup: Generate the data using the rand function in C with a range of 0 to 5000 divided by RAND-MAX to create a float.

Specifications of machine:

- i9-10850k @ 3.6GHz

- 10 Cores, 20 thread CPU
- 32Gb RAM
- Windows 11 Pro, version 10.0.22631, build 22631

1.3 Results and analysis

1.3.1 Parallel Section Vs Serial Implementation

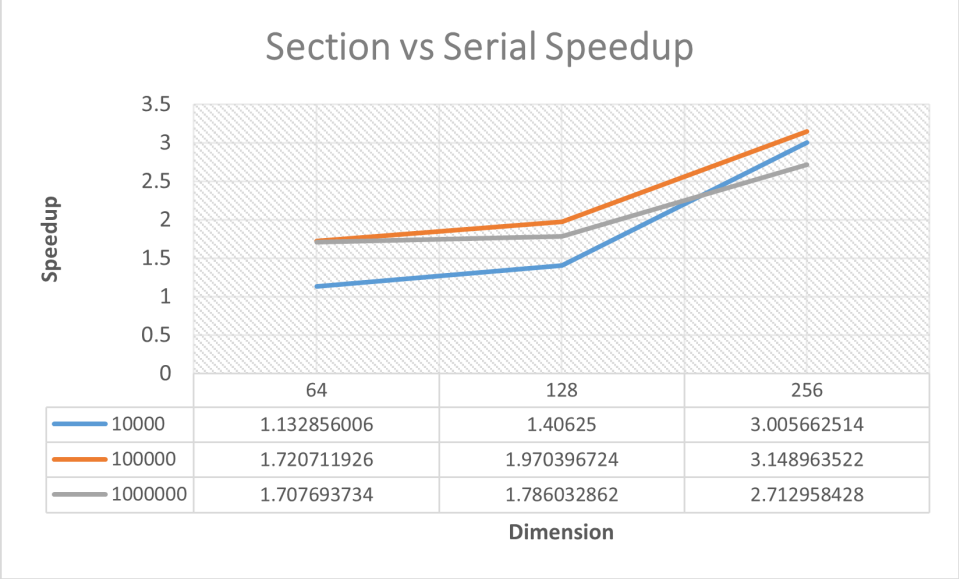


Figure 1: Results of Parallel Section implementation vs Serial implementation

The parallel section implementation exhibits a speedup over the serial implementation. Parallel for-loops calculate the distance array. Using a section quicksort implementation, sort the distance array in increasing order. The results show that, as the dimension of the points increases, so does the speedup. Furthermore, as the number of reference points increases, so does the speedup. Since we parallelized two computational components, we can examine their execution time as a percent of total execution.

Size	64	128	256
10000	57.36%	62.00%	42.53%
100000	69.55%	69.70%	53.17%
1000000	79%	80.03%	67.50%

Table 1: Section sorting time as percent of total execution time

The above percentages show that sorting dominates the execution. The results agree with the expected outcome, seeing as the section implementation of quick-sort is not the most efficient parallelization. Comparing to the results presented under parallel task implementation, the percent of execution corroborates this lack of efficiency.

1.3.2 Parallel Task Vs Serial Implementation

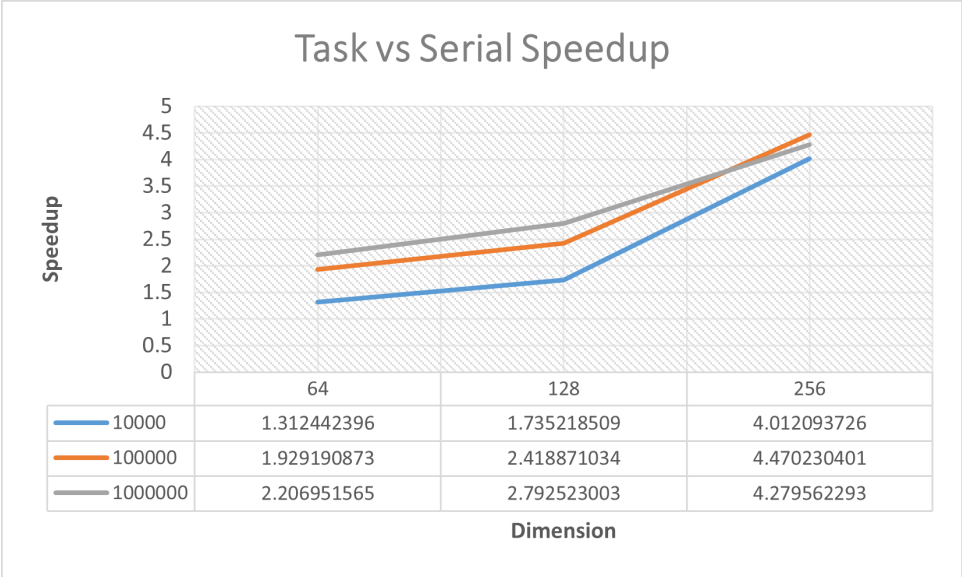


Figure 2: Results of Parallel Task implementation vs Serial implementation

The parallel task implementation exhibits a speedup over the serial implementation. Parallel for-loops calculate the distance array. Using a task quicksort implementation, sort the distance array in increasing order. The results show that, as the dimension of the points increases, so does the speedup. Furthermore, as the number of reference points increases, so does the speedup. The increase in speedup is notable in higher dimensions, reaching up to 4.5 times speedup. Since we parallelized two computational components, we can examine their execution time as a percent of total execution.

Size	64	128	256
10000	29.74%	27.01%	19.03%
100000	28.79%	22.85%	17.90%
1000000	36%	21.94%	21.85%

Table 2: Task sorting time as percent of total execution time

The above percentages suggest that distance dominates the execution. The results agree with the expected outcome, seeing as the task implementation of quick-sort is an efficient parallelization. Compared to the results presented under parallel section implementation, the percentage of execution corroborates this efficiency.

1.4 Correctness Assertion

For any parallel implementation, correctness has to be confirmed. In the above implementations, we do this by checking that every distance array is in increasing order, allowing us to confirm the implementation's correctness. We perform this check after every quick-sort call.

1.5 Conclusion

The results of this experiment show that the parallel task implementation is indeed more efficient than the parallel section implementation. Furthermore, the results show that both implementations perform better than the serial implementation despite the difference in the percentage of computation as discussed.

2 CUDA C Image Convolution

Consider the human eye and its functionality. The retina of your eye transforms an image. For example, it sharpens edges between bright and dark sections. This process is similar to image convolutions. Image convolutions use weighted sums of pixels defined by the image kernel. The brute force implementation loops over all elements covered by the kernel mask to compute and store the weighted sum. We can improve the speed by parallelizing this on a gpu. The following sections will compare the GPU Global and GPU Shared implementations with the Serial CPU implementation.

2.1 Test methodology

The test methodology involves a bash script to run the program on a set of input files, with specific operations. The program leverages the built-in sdk timers to measure the execution of the convolution algorithm. The script performs this benchmark for the CPU, GPU with global memory, and GPU with shared memory. For the reliability of results, the benchmark is performed four times for various images. The results are piped to an output file and are then tabulated. The experiment aims to investigate the difference in performance of the two GPU implementations over the serial implementation. It accomplishes this through a controlled environment, i.e., the hardware used to run the experiments remains unchanged.

2.2 Experiment Setup

Experiment setup: The image is read in from the pgm file. The appropriate memory allocations are made on the gpu. Then the image convolution is calculated. First on the CPU, then GPU Global Memory, and lastly, GPU Shared memory. The times and associated speedups are also recorded.

Specifications of machine:

- i7-1165G7 @ 1.2GHz
- 4 Cores, 8 thread CPU
- 16Gb RAM
- Windows 11 Pro, version 10.0.22631, build 22631
- MX350 GPU

2.3 Results and analysis

The results are as expected, the serial cpu implementation is the slowest of the three implementations. The global memory implementation has a decent speedup, since each thread will calculate a single pixel in parallel. The shared memory implementation does essentially the same as global memory implementation, with the exception of memory location. This shows in the speedup achieved by the shared memory implementation. From table 3, we see that the GPU

Run	Serial	GPU Global	GPU Shared
1	27.3	649.95	1391.3
2	28.48	646.46	1242.72
3	9.47	281.01	688.17
4	8.5	278.3	640

Table 3: Mpixels/sec for image convolution

implementations are considerably faster than the CPU implementation. It is evident that on average, the CPU processes 18.44 Megapixels per second, where as the GPU global and Shared implementations process 463.96 Megapixels per second and 990.55 Megapixels per second respectively.

Run	Serial	GPU Global	GPU Shared
1	1	23.80	50.96
2	1	22.69	43.63
3	1	29.67	72.67
4	1	32.74	75.29

Table 4: Speedup for image convolution

It is important to note the size of the kernel has a considerable effect on the speedup achieved. For instance, the last two entries made use of 5x5 kernels instead of 3x3 kernels. It is evident that the larger kernels performed more efficiently in the shared and global implementations.

2.4 Correctness Assertion

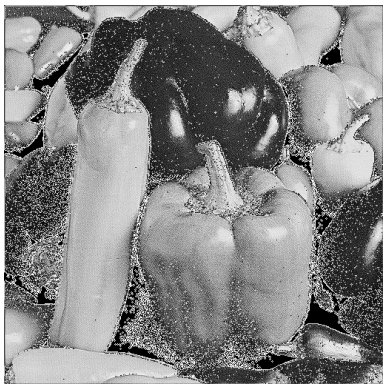
For any parallel implementation, correctness has to be confirmed. In order to do this, I perform the convolution of all three implementations, and then inspect visually. The three implementations should output the same image. I know if they do, then their implementations are correct as I know the serial implementation is correct. For example, in figure 3 below, the average kernel is applied to the image and compared for all three implementations. I perform the analysis for all kernels, as can be seen in figures 4, 5, and 6 as well. Note for figures 5 and 6, noise has been amplified substantially by all three implementations.

(a) CPU

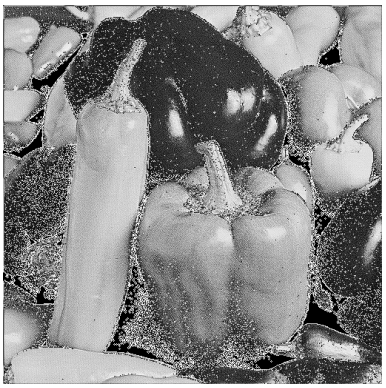
(b) GPU Shared

(c) GPU Global

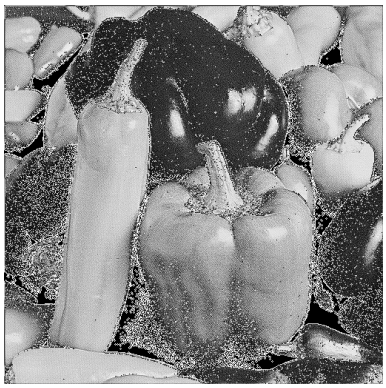
Figure 3: Results of Average Kernel



(a) CPU



(b) GPU Shared

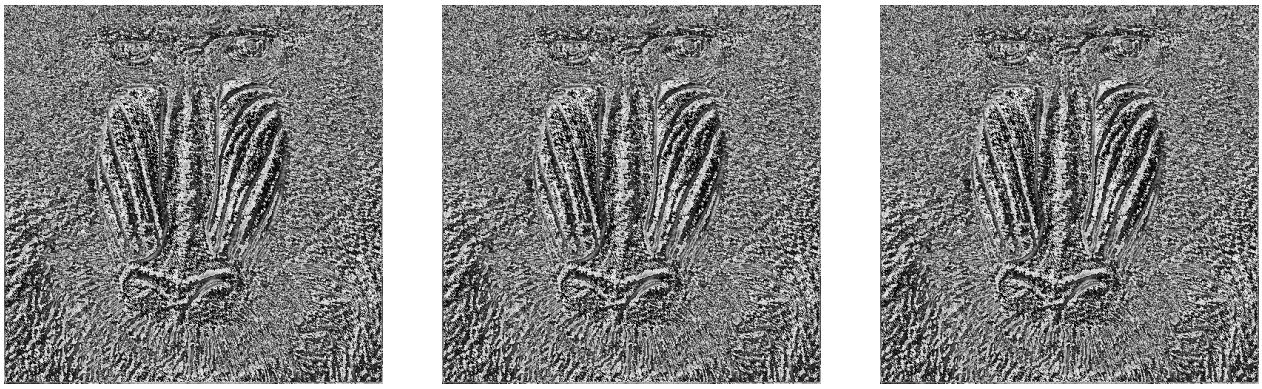


(c) GPU Global

Figure 4: Results of Sharpen Kernel

(a) CPU (b) GPU Shared (c) GPU Global

Figure 5: Results of Emboss (3x3) Kernel



(a) CPU (b) GPU Shared (c) GPU Global

Figure 6: Results of Emboss (5x5) Kernel

2.5 Conclusion

The results of this experiment show that the GPU Shared Memory implementation is indeed more efficient than the Global Memory implementation. Furthermore, the results show that both implementations perform better than the serial implementation. It also shows that the larger kernel size experience improved efficiency over smaller kernels.

References

[1] Z. Zhang, "Introduction to machine learning: k-nearest neighbors," *Annals of Translational Medicine*, vol. 4, p. 218, June 2016.