Team project: Profiling and Optimization on a Rodinia program K-nearest Neighbors

Yitong Li - yitli@kth.se
Guoqing Liang - guoqingl@kth.se
Tianze Yao - tianzey@kth.se
Martin Börjeson - mborjeso@kth.se

GitHub link: https://github.com/CHU-2002/DD2360-Project-PG1.git
Expected Grade: A

January 8, 2024

1 Introduction

In statistics, the k-nearest neighbors algorithm (k-NN) is a non-parametric supervised learning method first developed by Evelyn Fix and Joseph Hodges in 1951 [1], and later expanded by Thomas Cover [2]. It is used for classification and regression. In both cases, the input consists of the k closest training examples in a data set. The output depends on whether k-NN is used for classification or regression. In k-NN classification, the output is a class membership. An object is classified by a plurality vote of its neighbors, with the object being assigned to the class most common among its k nearest neighbors (k is a positive integer, typically much more smaller than the number of all members). If k = 1, then the object is simply assigned to the class of that single nearest neighbor.
In our project, we performed the optimization and profiling to a k-NN classification program, which finds k nearest neighbors around a plot in a given set of plots data to perform the k-NN classification process. We modified and optimized logical and GPU execution modules by using Quickselect algorithm and adding muti-stream to the original code, where several notations are marking our modifications.

2 Methodology

In this project, our logic module performs and solves a k-NN problem, i.e., process the given sets of data and output a series of specific data as the closest neighbors to a specified data. In this case, we use a series of Hurricanes' locations data (expressed as longitude and latitude) as the original data, the distances between Hurricanes and the given location is used to decide if they are neighbors of the plot. When a specified location and the number of neighbors to be found are given, we can get all neighbors' data (include location, name, distance...) after execution. Moreover, we optimized both logic module and GPU execution module after it achieved the expected output goal.

1. Logic Module
   In the original version, the program uses Chooseselect to find all neighbors, i.e., use loops to first find the closest, and then the second closest, etc. This methodology is slow in execution due to large amount of loops and also causes a waste of resource to execution, because it's unnecessary to set the neighbors in order according to their distance. Thus, we use Quickselect methodology instead, an algorithm which was developed by Hoare in the early 60s [3].
   Figure.1 shows a simple example about how Quickselect works. It begins with a specified pivot and two pointers i,j. i is set to mark the first number that larger than the pivot, if not, then i++ to the next number until it's larger than the pivot. j is used to compare all elements with pivot. If it's smaller, then the element will exchange its location with the element pointed by i. At last, when all elements is compared, j is pointing the pivot and the pivot will exchange location with the element pointed by i, which makes the pivot a boundary in terms of value.
   It's apparent that Quickselect methodology notably decreases the amount of comparisons, which decrements time complexity from $O(N^2)$ to $O(N)$ on average. This greatly decreases the complexity in exection and makes a great acceleration to it.To set a proper segment size for each streams and threads to run, when considering the least time complexity, We can measure three main clas-

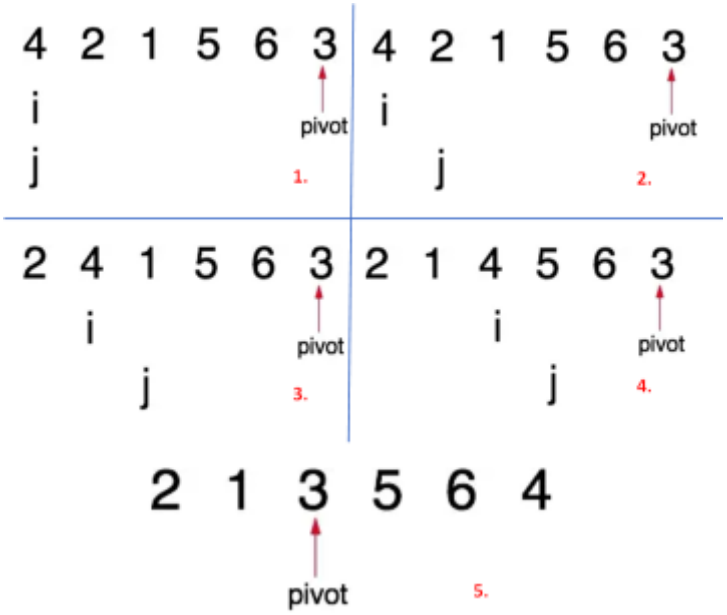

Figure 1: Quickselect example

sification methods' time complexity as follows:

(a) Quickselect
When there is m segments in total number records, the time complexity is

$$O\left(\frac{NumOfRecord}{m} + m * ResultNum\right)$$

It's easy to conclude that when $m = \sqrt{\frac{NumOfRecord}{ResultNum}}$, the time complexity gets the minimum $O\left(\sqrt{NumOfRecord * ResultNums}\right)$ where the constant is ignored.

(b) Min-Heap
With the same knowledge, the time complexity is

$$O\left((\frac{NumOfRecord}{m} + m * ResultNum) * log(ResultNum)\right)$$

We also calculate the minimum $O\left(\sqrt{NumOfRecord * ResultNums} * \log(ResultNum)\right)$ when $m = \sqrt{\frac{NumOfRecord}{ResultNum}}$.

(c) Chooseselect This scheme's time complexity is constant $O\left(NumOfRecord * \log(NumOfRecord)\right)$

Therefore, considering that ResultNum is always more than 10 and the input number records are large, it's appropriate to adopt Quickselect to optimize and set each segment size to $k * \sqrt{\frac{NumOfRecord}{ResultNum}}$, where k is an coefficient.

2. GPU Arrangements Methodology
Different from the original version, which only use GPU to calculate the distances, we also assign Quickselect execution to GPU to gain a shorter execution time and better performance.

Figure.2 shows how we arrange our GPU utilization. We implement muti-stream scheme to parallel execute Quickselect. With this arrangement, the total records are broken into segments for streams. Streams run Quickselect asynchronously to find a set of neighbors (neighbor group, NG) in their own segment. Finally, all NG is gathered to default stream (stream0) to execute the last Quickselect and output the final result. To make the total time as short as possible, we set the number of neighbors in every NG is equal to the desired result count.
The most advantageous point of this scheme is high GPU utilization and great acceleration due to strong capacity in calculation of GPU. Moreover, The pinned memory created for streams also makes acceleration to data transference.

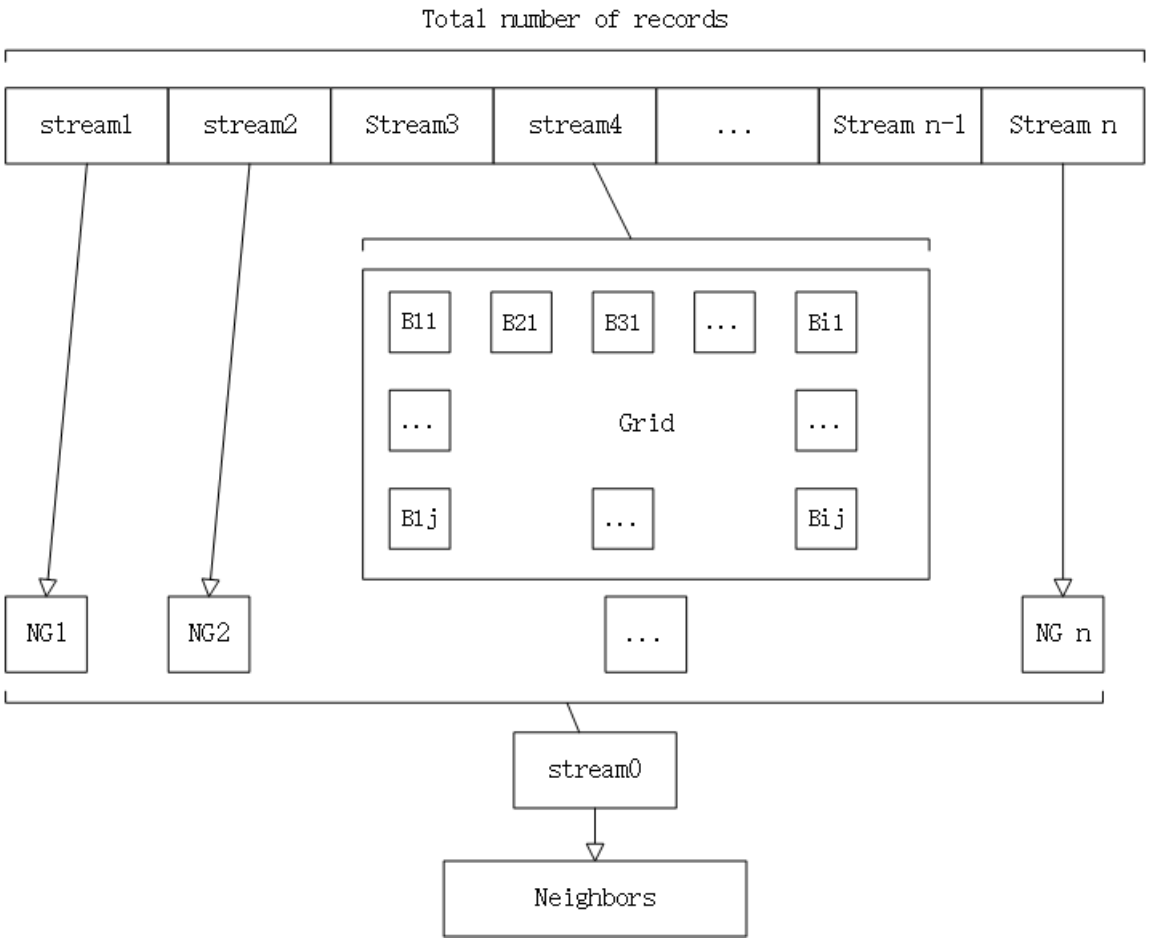

Figure 2: GPU utilization

3 Experimental Setup

1. GPU Platform
   To optimize this program, we use Visual Studio Code to coding and modifying the original version of k-NN program.
   To run and verify our improvements to the program and test its performance, we use Google Drive and Google Colab to upload our optimized program, compile program and run it.
   The GPU we use in Google Colab is NVIDIA Tesla T4. The table.1 shows some specific technical information regarding NVIDIA Tesla T4.

GPU Part	**Data**
CUDA Version	7.5
Total Global Memory	15102MB
Total Regs per Block	65536
Max Thread per Block	1024
Max Dimension of Block	(1024, 1024, 64)

Table 1: GPU Data

2. Tools
   Besides the coding tool Visual Studio Code, we use NVIDIA Visual Profiler to profile and trace the data transference in our program. The profile and visualized traced data transference results are showed in Result part.

4 Results

Our implementations of K-NN yielded the same output as the original for all tested inputs.

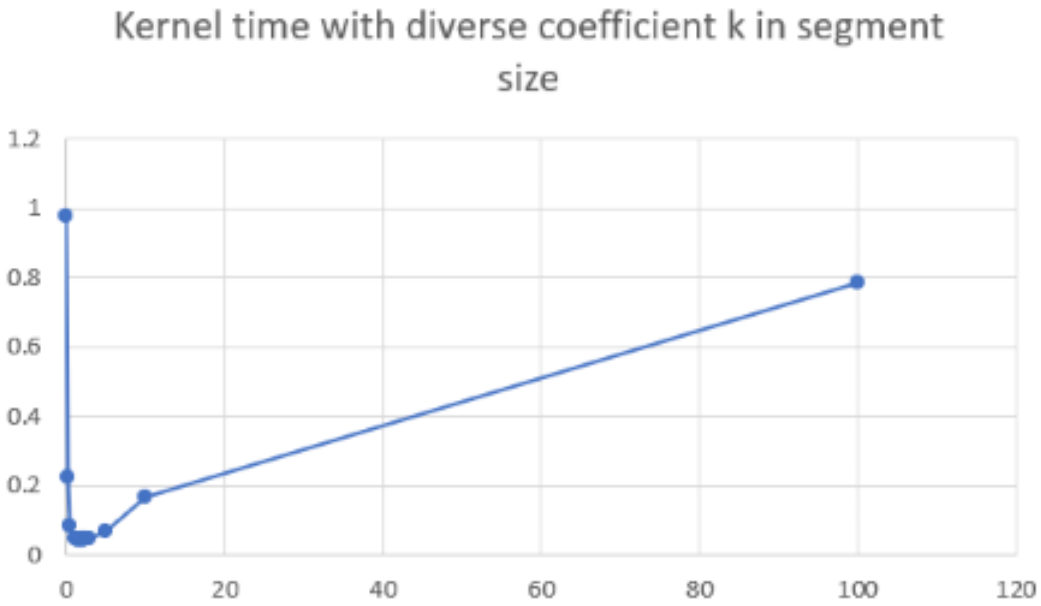

Figure 3: coefficient k choosing test. We choose k = 1.8 when it gets minimum execution time

Group size	Original	QuckSelect	QuickSelect 4 streams	QuickSelect 10 streams
100	4569 ms	308.1 ms	284.7 ms	343.5 ms
300	8673 ms	414.3 ms	386.2 ms	443.4 ms
500	9939 ms	416.8 ms	398.5 ms	421.35 ms
1000	17097 ms	555.7 ms	556.3 ms	553.2 ms
3000	45063 ms	683.5 ms	538.6 ms	812.4 ms
5000	72227 ms	721.2 ms	835.7 ms	1008.4 ms
10000	166824 ms	843.9 ms	1155.9 ms	1047.9 ms
30000	417165 ms	1259.6 ms	1474.9 ms	1822.0 ms
50000	697713 ms	1732.6 ms	1891.5 ms	2204.2 ms
100000	-	2587.2 ms	2313.6 ms	2869.8 ms

Table 2: Execution time of different configurations of KNN. The size of the input was 5120k points.

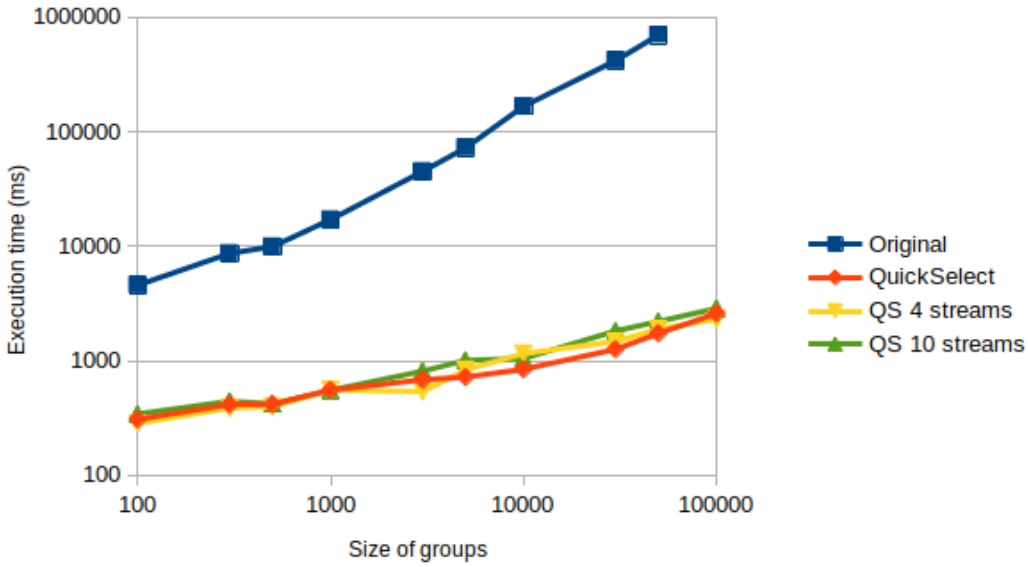

Figure 4: Loglog graph of table 2. Note that the input size is constant. It is the size of the groups that increases along the X-axis.

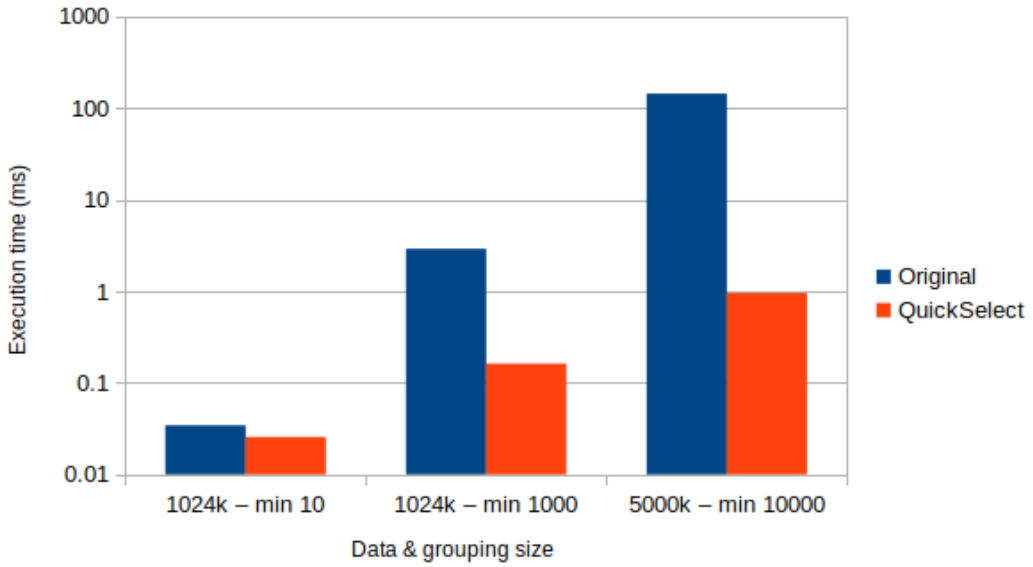

Figure 5: Comparative execution time of original implementation of Min-K and our implementation. Take note of the logarithmic scale on the Y-axis.

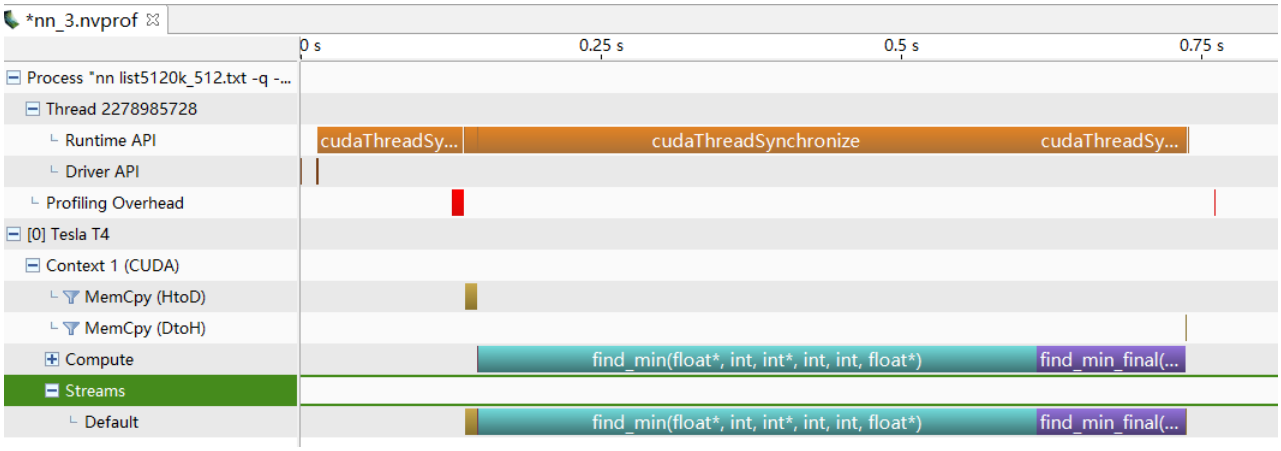

Figure 6: NVVP result of our implementation of K-NN without multiple streams

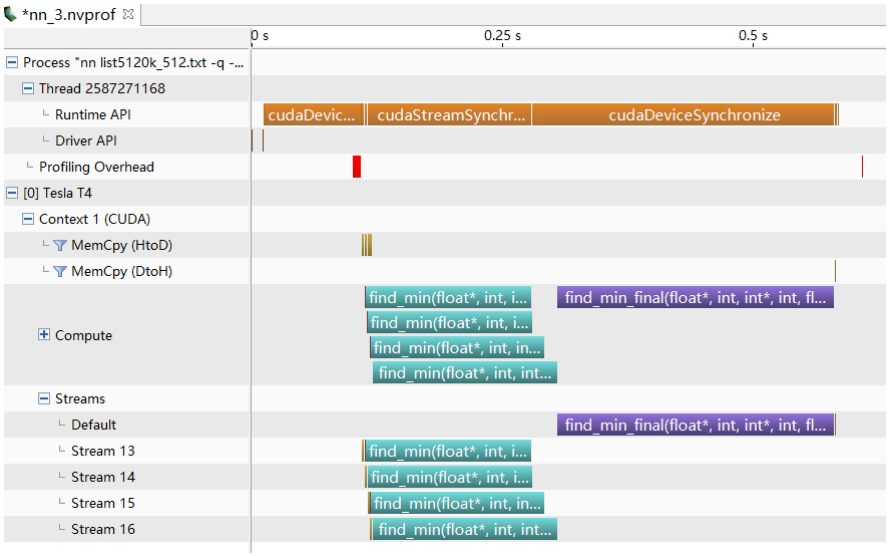

Figure 7: NVVP result of our implementation of K-NN with multiple streams

5 Discussion

The first step when optimizing a program is always to profile the original program to see which parts of the code takes the most time to run. Otherwise, you might spend a lot of time optimizing a part

of the program which didn't take long to run to begin with. When we looked at the execution time of the different parts of the original code, we noticed that the vast majority of the execution time of the program wasn't spent in the kernel at all, but was spent on finding the k:th smallest element in an array. After implementing the QuickSelect and running it in parallel on the GPU, the program ran up to 150 times faster, as can be seen in figure 5. For smaller groupings, such as $k = 10$, the performance uplift is much smaller, in the range of 33% faster.

Increasing the value of k (i.e. increasing the size of the groupings) increased the performance gap between our implementation and the original code, as can be seen in table 2 and figure 4. This makes sense when looking at the original program. Their implementation uses selection sort to sort the k smallest values of the list. This is fine for small values of k, as the program will only need to iterate over the array a handful of times. In general however, the program will need to iterate over the array k times to find and sort the k smallest elements. This is why we observe a roughly linear growth in execution time as k increases in the original code, whereas the growth in execution time in our implementation is sublinear.

Varying the thread per block didn't have a big effect on execution time for larger data sizes, but we observed small performance gains by going to 1024 threads per block.

The streams optimization didn't yield as drastic of an improvement, but we still observed a performance uplift in the largest data sets. Generally, 4 streams seems to be close to the optimum. The reason why multiple streams didn't yield larger performance gains is likely due to that the kernel execution time is much larger than the device to host memcopy time. Running multiple streams allows the program to transfer data from device to host at the same time as another stream runs in the kernel, but that benefit is minimal when the time to transfer data is relatively small. In Figure 6 and 7, we can see that the MemCpy time is miniscule compared to *find_min* and *find_min_final*.

6 Conclusion

Our implementation of k-NN was generally much faster than the original implementation. As the size of the groupings increase, the difference in performance becomes more and more pronounced. Much of this improvement comes from the improved selection algorithm, QuickSelect. Running QuickSelect in parallel on the GPU sped up the execution further. Further optimizations came from running multiple streams, with 4 appearing to be optimal, and tuning the number of threads per block to 1024.

The smallest performance uplift comes from using very small groupings, where $k = 10$. In those cases, QuickSelect was still faster, but not by much.

Overall, we have improved significantly upon the original implementation, which now utilizes the GPU to a much greater extent than it did originally. By improving the selection algorithm, and running it in parallel on the GPU, some input instances are run orders of magnitude quicker.

For further optimization on k-NN program, we may try to use more powerful GPU to run the program and it's expected that the optimizations will be more magnificent. Moreover, we can also use shared memory for more quick data transference and try to improve Min-Heap algorithm to get another version of optimization in k-NN. With the optimization knowledge in this case, we can apply this scheme in more areas such as sorting mineral in geology, classifying stars in the universe, etc. Moreover, we can also try to increase the dimension in our input data and apply Quickselect algorithm in data with more dimensions.

7 Contributions

Yitong Li and Guoqing Liang worked on improving the implementation of K-NN. Tianze Yao and Martin Börjeson worked on the written report. We all tested and ran both the original and the groups implementation of K-NN, and discussed the code and its performance. Everyone made sure to understand the improvements and why they were made.

References

[1] E. Fix and J. Hodges, *Discriminatory Analysis: Nonparametric Discrimination: Consistency Properties*. USAF School of Aviation Medicine, 1951.

[2] T. Cover and P. Hart, "Nearest neighbor pattern classification," *IEEE Transactions on Information Theory*, vol. 13, no. 1, pp. 21–27, 1967.

[3] C. A. R. Hoare, "Algorithm 65: Find," *Commun. ACM*, vol. 4, p. 321–322, jul 1961.