Parallel KNN on GPU with CUDA Implementation, Experiment and Analysis

Quoc Bao Tran
Department of Computer Science
Virginia Commonwealth University
University Richmond, Virginia 23284
tranq3@vcu.edu

***Abstract*—The k-Nearest Neighbors (KNN) algorithm is a fundamental machine learning technique, but its computational complexity poses challenges for large-scale datasets. This study implements and analyzes parallel versions of the KNN algorithm on GPU to address these scalability issues. We compare the performance of this implementation across various dataset sizes (small, medium, and large). Our results demonstrate significant speedups and improved scalability, particularly for larger datasets. We analyze runtime, speedup, and efficiency metrics, estimating the proportion of parallelizable code.**

I. Introduction

Machine learning algorithms are increasingly applied to massive datasets, pushing the limits of computational resources and highlighting the need for efficient, scalable implementations. The k-Nearest Neighbors (KNN) algorithm, despite its simplicity and effectiveness, faces significant challenges when scaled to large datasets due to its computational complexity of $O(\mathbf{num\_test} \times \mathbf{num\_train} \times \mathbf{num\_features})$.

KNN operates by computing distances between each test instance and all training instances, followed by a majority vote among the k nearest neighbors to predict the class of the test instance. While conceptually straightforward, this process becomes computationally intensive as dataset sizes grow, limiting the algorithm's applicability to large-scale machine learning problems.

Parallel computing techniques offer promising solutions to address these scalability issues. In this study, we implemented a GPU version of the KNN algorithm, which demonstrated a significant reduction in runtime. This GPU implementation was benchmarked against single-thread, multi-thread, and MPI implementations. The results illustrate the effectiveness of the parallel implementations, particularly the GPU version, in terms of speedup.

II. METHODOLOGY

We implemented two GPU versions of the KNN algorithm using CUDA. The first implementation ($\mathbf{1-phase-parallel}$) leverages the massive parallelism offered by GPUs to significantly speed up the distance calculations, which are the most computationally intensive part of the KNN algorithm. To obtain these distances, we use a loop in the $\mathbf{KNN}$ kernel function to sum up all the differences between attributes of a pair of train-test datapoints. The calculated distances are copied back to the CPU, and we choose the k = 3 nearest distances and predict the classes corresponding to them.

The second implementation ($\mathbf{2-phases-parallel}$) is an upgraded version of the first one. Instead of sorting and choosing the k = 3 nearest distances on CPU, this approach calculates the distance by the first stream and then uses the calculated distances as input for the second stream to predict the potential class. We find the k = 3 nearest distances of each datapoint in our test set parallelly. We prepare the class for each data point in the h_train_class array and copy them to the device memory d_train_class. This step serves to identify the class of a data point in the training set, used by kernel function $\mathbf{find\_k\_nearest}$. In the $\mathbf{find\_k\_nearest}$, we get the calculated distance in d_dist array and store the top-3 in $candidates$ array, and we count the frequency of each class by $classCounts$ array. Finally, we choose the class with the highest value in $classCounts$ and store the predicted classes in the $predictions$ array.

a) Thread Allocation: Our GPU implementations allocate $\mathbf{num\_test} \times \mathbf{num\_train}$ threads, where $\mathbf{num\_test}$ is the number of instances in the test dataset and $\mathbf{num_train}$ is the number of instances in the training dataset. Each thread is responsible for computing the distance between one test data point and one training data point.

For the $\mathbf{2-phases-parallel}$ implementation, we allocate $\mathbf{num\_test}$ threads, and each of them is responsible for identifying the class of each data point in the test belong to.

b) Memory Optimization for the first phase: computing the distances: To achieve coalesced memory access, we organized the data as follows:

- Stored the original data in a one-dimensional array.
- Distinct attributes of one data are divided and the same attribute of all the data is aggregated together.
- The interval is equal to the number of instances.

c) Kernel Configuration: Regarding the first phase, we assign each block as a 1d grid with $(num\_test \times num\_train + threads\_per\_block - 1)/threads\_per\_block$ blocks.

For the second phase, we assign each block as a 1d grid with $(num\_test + threads\_per\_block - 1)/threads\_per\_block$ blocks.

III. Experimental results

1) Datasets: We run the GPU version on three datasets with different sizes. The small dataset contains 160 test instances and 1184 train instances. The medium dataset contains 740 test instances and 14708 train instances. The large dataset contains 3436 test instances and 61606 train instances. We run the KNN algorithm with k = 3 of the sequential version and parallel and distributed versions. The experiments are run on the Athena servers, and the detailed data and results are stored in the $excel$ folders. The file $cuda.cu$ is the 1-phase-parallel implementation and the file $cuda\_2\_phases$ is the 2-phases-parallel implementation.

A. Kernel configurations

We vary different number of threads per block with our two implementations. We run our code on the H100 GPU and choose a specific CUDA SDK version and Microarchitecture by enabling:

```
--gpu-architecture=compute_90
--gpu-code=sm_90
```

(the newest one on Athena) to make sure we have the best performance.

1) Performance analysis: We illustrate the execution times in milliseconds (ms) for our two implementation versions of the KNN algorithm on small, medium, and large data in the table I.

Table II shows the fastest runtime of multi-thread versions (POSIX Threads and OpenMP) and multi-process version (MPI) and our GPU implementations. For MPI implementation, we choose the results on 1 node because it's more consistent and accurate than multiple nodes (using the results from our previous assignment).

To evaluate the performance improvement of our GPU implementations, we calculate the speedup over the sequential version. The speedup is defined as:

$$speedup = \frac{T_s}{T_{GPU}}, \qquad (1)$$

where T_s is the total runtime of the sequential/ POSIX threads/ OpenMP/ MPI version, and T_{GPU} is the total runtime of the parallel version.

Table III shows the speed up of our 2-phases-parallel over 1-phase-parallel on GPU.

Table IV, V, VI and VII illustrate the speed up of our 2-phases-parallel over sequential, POSIX threads, OpenMD and MPI, respectively.

To visualize the speedup trends, Figure 1, 2, 3 and 4 show the speedup of our 2-phases-parallel over sequential, POSIX threads, OpenMD and MPI, respectively.

TABLE I
RUNTIME (MS) OF 1-PHASE-PARALLEL AND 2-PHASES-PARALLEL KNN IMPLEMENTATIONS

Threads per block	1-phase-parallel			2-phases-parallel		
	Small	Medium	Large	Small	Medium	Large
1	1.09	54.70	1027.12	1.07	49.90	664.61
2	1.09	56.06	1024.86	1.06	37.17	603.95
4	1.09	57.60	1022.14	2.86	37.24	566.89
8	1.09	54.76	1025.46	2.75	31.85	555.42
16	1.08	53.27	1026.67	1.04	34.46	556.92
32	1.09	54.71	1023.64	2.94	33.39	549.24
64	1.10	54.74	1025.69	1.06	31.49	545.75
128	1.09	54.45	1025.71	1.06	31.78	548.39
256	1.10	54.82	1023.91	1.05	36.61	540.83
512	1.08	54.58	1024.75	1.04	34.54	555.39
1024	2.17	54.75	1031.28	1.07	37.49	552.90

TABLE II
RUNNING TIME OF DIFFERENT IMPLEMENTATIONS FOR SMALL, MEDIUM, AND LARGE DATASETS

Method	Small	Medium	Large
Sequential	18.00	798.33	11659.66
POSIX Threads	5.67	285.67	3442.33
OpenMP	9.00	295.33	3646.00
MPI (on 1 node)	25.33	79.67	926.00
1-phase-parallel GPU	1.08	53.27	1022.14
2-phases-parallel GPU	1.04	31.49	540.83

TABLE III
SPEED UP OF 2-PHASES-PARALLEL COMPARED WITH 1-PHASE-PARALLEL

No. threads_per_block	Small	Medium	Large
1	1.02	1.10	1.55
2	1.03	1.51	1.70
4	0.38	1.55	1.80
8	0.40	1.72	1.85
16	1.04	1.55	1.84
32	0.37	1.64	1.86
64	1.04	1.74	1.88
128	1.03	1.71	1.87
256	1.05	1.50	1.89
512	1.04	1.58	1.85
1024	2.03	1.46	1.87

TABLE IV
SPEEDUPS OF GPU IMPLEMENTATION OVER SEQUENTIAL IMPLEMENTATION ON SMALL, MEDIUM AND LARGE DATASETS, RESPECTIVELY

No. threads_per_block	Small	Medium	Large
1	16.82	16.00	17.54
2	16.98	21.48	19.31
4	6.29	21.44	20.57
8	6.55	25.07	20.99
16	17.31	23.17	20.94
32	6.12	23.91	21.23
64	16.98	25.35	21.36
128	16.98	25.12	21.26
256	17.14	21.81	21.56
512	17.31	23.11	20.99
1024	16.82	21.29	21.09

TABLE V

SPEEDUPS OF GPU IMPLEMENTATION OVER POSIX THREADS IMPLEMENTATION ON SMALL, MEDIUM AND LARGE DATASETS, RESPECTIVELY

No. threads_per_block	Small	Medium	Large
1	5.30	5.72	5.18
2	5.35	7.69	5.70
4	1.98	7.67	6.07
8	2.06	8.97	6.20
16	5.45	8.29	6.18
32	1.93	8.56	6.27
64	5.35	9.07	6.31
128	5.35	8.99	6.28
256	5.40	7.80	6.36
512	5.45	8.27	6.20
1024	5.30	7.62	6.23

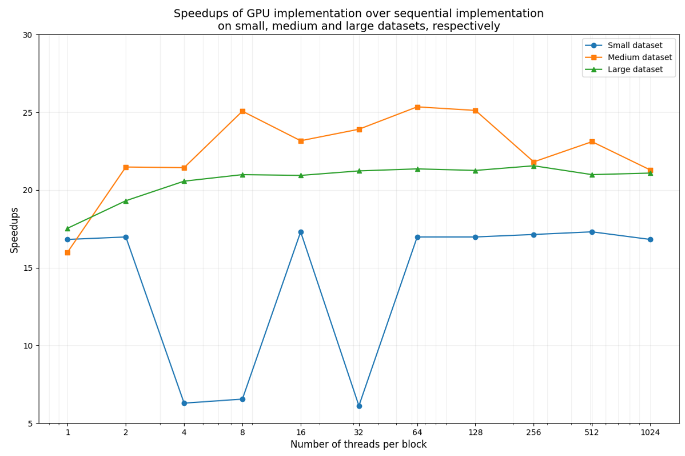

Fig. 1. Speedup of GPU KNN implementation over sequential version

TABLE VI

SPEEDUPS OF GPU IMPLEMENTATION OVER OPENMP IMPLEMENTATION ON SMALL, MEDIUM AND LARGE DATASETS, RESPECTIVELY

No. threads_per_block	Small	Medium	Large
1	8.41	5.92	5.49
2	8.49	7.95	6.04
4	3.15	7.93	6.43
8	3.27	9.27	6.56
16	8.65	8.57	6.55
32	3.06	8.84	6.64
64	8.49	9.38	6.68
128	8.49	9.29	6.65
256	8.57	8.07	6.74
512	8.65	8.55	6.56
1024	8.41	7.88	6.59

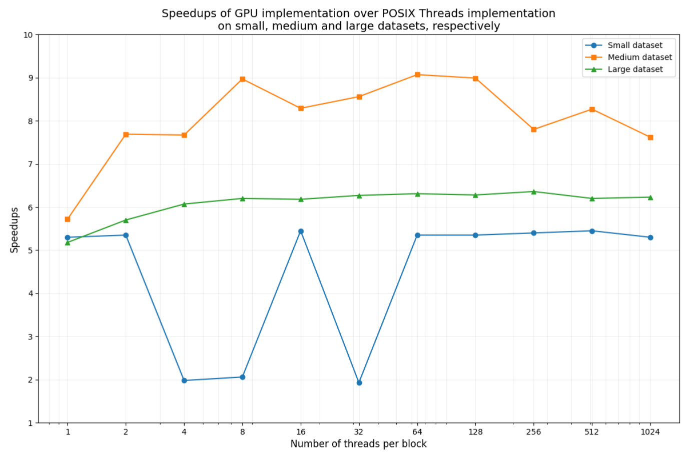

Fig. 2. Speedup of GPU KNN implementation over multi-thread POSIX version

TABLE VII

SPEEDUPS OF GPU IMPLEMENTATION OVER MPI IMPLEMENTATION ON SMALL, MEDIUM AND LARGE DATASETS, RESPECTIVELY

No. threads_per_block	Small	Medium	Large
1	23.67	1.60	1.39
2	23.90	2.14	1.53
4	8.86	2.14	1.63
8	9.21	2.50	1.67
16	24.36	2.31	1.66
32	8.62	2.39	1.69
64	23.90	2.53	1.70
128	23.90	2.51	1.69
256	24.12	2.18	1.71
512	24.36	2.31	1.67
1024	23.67	2.13	1.67

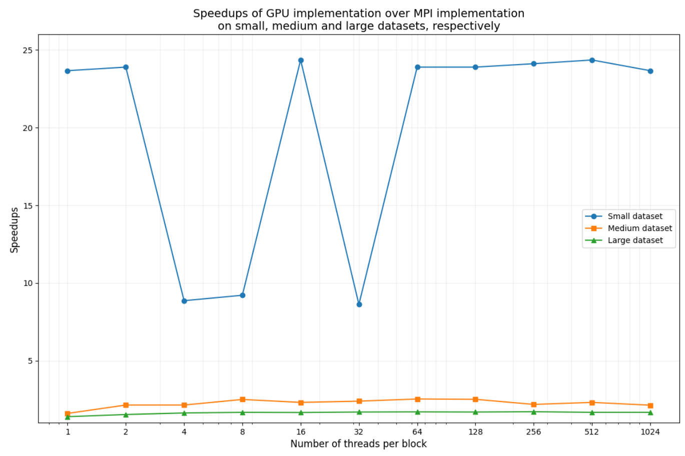

Fig. 3. Speedup of GPU KNN implementation over OpenMP version

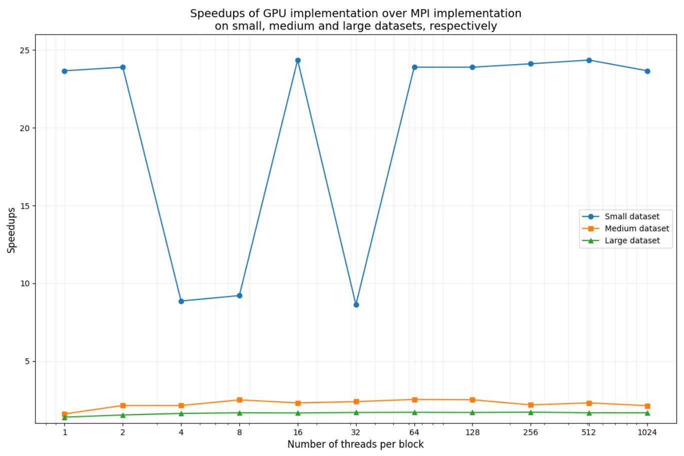

Fig. 4. Speedup of GPU KNN implementation over MPI version

IV. Conclusion

Our study demonstrates that GPU implementations of the KNN algorithm can achieve significant performance improvements over sequential and other parallel versions. The 2-phases-parallel GPU implementation consistently outperformed all other versions across small, medium, and large datasets, with speedups ranging from 16.82x to 25.35x over the sequential implementation for different thread configurations.

Our GPU implementation shows superior performance compared to POSIX Threads, OpenMP, and MPI implementations. For the large dataset, we observed speedups of up to 6.36x over POSIX Threads, 6.74x over OpenMP, and 1.71x over MPI.

For future work, we propose exploring multi-GPU implementations to achieve even greater speedups. We also can use Profiler from Nvidia to analyze the bottleneck of the streams of the program.