CMSC 603 Assignment 2: KNN on Cuda

Darshini Mahendran
Computer Science VCU
Richmond, USA
mahendrand@vcu.edu

***Abstract*—K-Nearest Neighbors algorithm (KNN) is a classification algorithm and one of the most used learning algorithms. In high dimensional spaces, the run time of the KNN is considered as a bottleneck. In this project we address this problem by performing the KNN on the GPU (Graphics Processing Unit) using CUDA. Different kernel configurations are implemented to reduce the run time and the performance is analyzed using the NVIDIA profiler.**

***Index Terms*—KNN, GPU, Cuda, parallelization**

I. Introduction

KNN is a simple classification algorithm. The algorithm is based on the feature similarity, it classifies a given data point based on how closely out-of-sample attributes, it resembles the training set. Three different kernel configurations are implemented: 1D grid and 1D-x block mapping (thread per instance), 1D grid and 1D-x block mapping with shared memory (block per instance), 1D grid of 2D blocks mapping (column per instance).The number of k is taken as a user parameter. Code is optimized as both the accuracy of the sequential and MPI and Cuda versions are same. The grid and block size are determined depending on the number of threads assigned per block.

II. Implementation

A. 1D grid and 1D-x block mapping

First approach is a 1D grid with 1D-x block mapping. For a given dataset, each thread is assigned to perform KNN for a particular instance. Number of threads are equal to the number of instances. Depending on the number of threads per block, the block number is determined. The mapping for *Approach 1* is shown in Fig. 1

B. 1D grid and 1D-x block mapping with shared memory

Second approach, is 1D grid and 1D-x block mapping with shared memory. For a given dataset, each block is assigned to perform KNN for a particular instance. Euclidean distance calculation of one instance with all other instances are divided between the threads in each block. Since all threads in the block are performing the euclidean distance calculation simultaneously, the distance matrix is initialized in the shared memory where all threads were synchronized before selecting the *k* number of nearest of elements. Since multiple threads are assigned to one instance instead of one thread the processing time is reduced which makes this mapping more efficient than the *Approach 1*. The mapping is shown in the *Approach 2* in Fig. 1

C. 1D grid of 2D blocks mapping

Third approach is a 1D grid divided into 2D blocks and are indexed using row and column values. For a given dataset, each column is assigned to perform KNN for a particular instance. The euclidean distance calculation performed by different threads are written to a 2D matrix in the shared memory and read into a temporary array while selecting the *k* number of nearest of elements. Since one block is assigned to more than one instance to perform KNN, this reduces the time compared to the approach where a thread is assigned to an instance but not than the approach where each block is assigned one instance. The mapping is shown in the *Approach 3* in Fig. 1

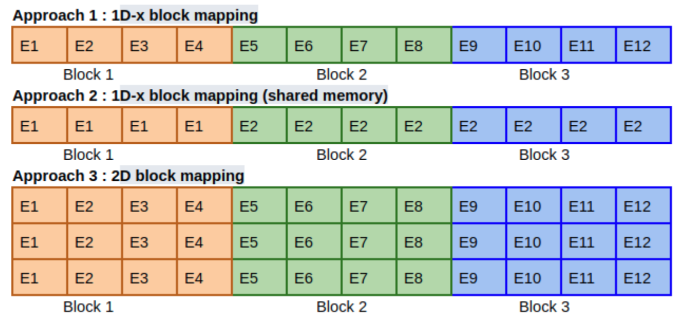


Fig. 1. Mappings of the different kernel configurations used. The instances are denoted by E .

D. Dataset

- Small dataset has 336 instances and when k = 10 the algorithm gives the highest accuracy of 0.8363.
- Medium dataset has 4898 instances and when k = 2 the algorithm gives the highest accuracy of 0.6615.

III. Results and Discussion

Table. I shows the run time of the single-threaded, MPI and GPU versions in maple server considering the two datasets and the run time of the GPU approaches are compared with the best run time obtained with MPI processes for both datasets respectively (small dataset - 4 processes, medium dataset - 32 processes). The results show that the runtime of all 3 GPU

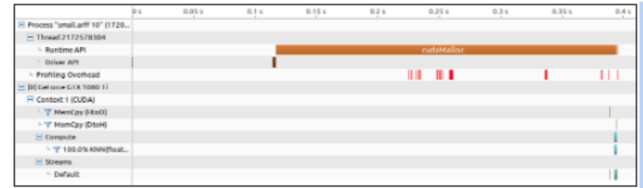
TABLE I
COMPARISON OF THE RUNTIME WHEN KNN IS EXECUTED IN THE MAPLE SERVER

	Small dataset	Medium dataset
Single-threaded	79ms	11164 ms
MPI - 4 processes	297 ms	3790 ms
MPI - 32 processes	510 ms	1220 ms
GPU - approach 1	5.65 ms	35.27 ms
GPU - approach 2	**0.71ms**	**89.12 ms**
GPU - approach 3	3.85 ms	596.75 ms

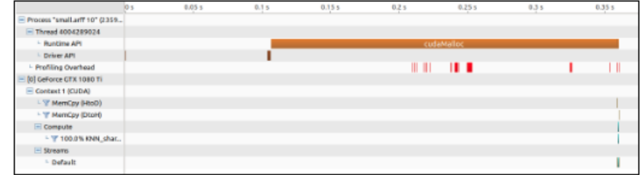
TABLE II
COMPARISON OF THE RUNTIME FOR DIFFERENT NUMBER OF THREADS PER BLOCK FOR THE GPU APPROACHES ON SMALL DATASET.

threads/ block	Approach 1	Approach 2	Approach 3
2	6.18 ms	2.32 ms	3.90 ms
4	6.18 ms	1.43 m	-
8	5.68 ms	0.97 ms	-
16	5.67 ms	0.71 ms	-
32	5.82 ms	0.60 ms	-

Approach 1 : 1D-x block mapping



Approach 2 : 1D-x block mapping (shared memory)



Approach 3 : 2D block mapping

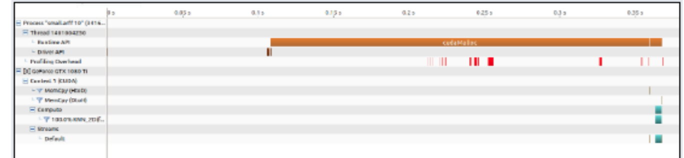


Fig. 2. Performance of the GPU approaches visualized using the NVIDIA profiler on the small dataset .

approaches are significantly lower than the single-threaded and the MPI versions. Among the GPU approaches, approach 2 has the least runtime. Approach 2 has a 1D grid and 1D-x block mapping with shared memory where each block performs KNN for an instance, since all threads in a block is assigned for one instance this appraoch is more efficient. Approch 3 has the second lowest runtime as more than one thread is assigned for one instance whereas the approach 1 is the least efficient as it only 1 thread is assigned for one instance.

IV. ANALYZE THE PERFORMANCE

More experiments were performed to further analyze the performance of each approaches. Small dataset was used for the experiments.

Table. II shows the runtime of each approach when the number of threads of each block is changed. From the results we can see the change in number of threads doesn't affect the performance of *Approach 1* as one thread is assigned to one instance and the performance doesn't depend on the number of blocks. In *Approach 2* we can a gradual drop in the runtime, this is because in this approach each block is assigned for an instance. Therefore, more threads in a block will increase the performance as all threads run simultaneously performing KNN for an instance. For *Approach 3* we cannot assign more than 2 threads per block because we can not allocate more shared memory in a block than the limit. Since we are writing the intermediate 2D distance matrix to the shared memory, the memory capacity is limited. This is a disadvantage of this approach. From this experiment, we can conclude *Approach 2* is most efficient.

The NVIDIA Visual Profiler is a cross-platform performance profiling tool that delivers developers vital feedback for optimizing CUDA C/C++ applications. It performs automated analysis of the application to identify performance bottlenecks and get optimization suggestions that can be used to improve performance. Fig. 2 shows the results when running the profiler on the 3 GPU approaches. Larger amount of time goes to allocate memory initially and KNN computation takes less time comparatively. If we consider the KNN computation part in all the approaches we can see *Approach 2* takes less time for computation of KNN. Therefore we can conclude, *Approach 2* is more efficient.

CODE

Code for the above implementation can be found in the following repository: https://github.com/SamMahen/2019-603-A2-Mahendran