UNIVERSITÀ
DI SIENA
1240

University of Siena

Department of Information Engineering and Mathematics

KNN algorithm in CUDA

Students:

Kevin Gagliano

Jacopo Andreucci

Professor:

Roberto Giorgi

Academic Year 2023-2024

Abstract

The k-Nearest Neighbors algorithm stands as a fundamental technique in machine learning classification and regression tasks. By harnessing the parallel processing capabilities of GPUs, CUDA (Compute Unified Device Architecture) accelerates the execution of the algorithm, leading to significant speedup compared to traditional CPU-based implementations. Moreover, video cards excel in handling the inherently parallel nature of kNN computations, efficiently processing massive amounts of data in parallel therefore reducing computational bottlenecks.

In this project, we examine the benefits of employing CUDA and GPUs for kNN computation, examining how various dataset sizes and feature vector dimensions affect performance in relation to two distinct hardware configurations.

Contents

Chapter 1

Introduction

1.1 Background

In the context of machine learning, the k-Nearest Neighbors (kNN) algorithm stands as one of the simplest yet powerful classification and regression techniques. Introduced in the 1950s, kNN remains relevant due to its simplicity and effectiveness, particularly in scenarios with small to moderate-sized datasets.

1.2 Basic Principle of kNN

The fundamental principle behind the kNN algorithm is straightforward: to classify a new data point, it looks at the k closest labeled data points (neighbors) in the training set and assigns the most common class label among those neighbors. For regression tasks, it predicts the output value by averaging the values of the k nearest neighbors.

1.3 Algorithm Steps

The steps that characterize the algorithm can be summarized as follows:

1. Calculate the distance between the target point and all points in the training dataset.
2. Select the k nearest neighbors based on the calculated distances.
3. For classification tasks, assign the class label most common among the selected neighbors to the target point. For regression tasks, predict the output value by averaging the values of the selected neighbors.

1.4 Parameter K

The main parameter of the kNN algorithm is k, the number of neighbors to consider. Choosing an appropriate value for k is crucial, as it significantly affects the algorithm's performance. A smaller value of k may lead to overfitting, while a larger value may result in underfitting.

Chapter 2

Timing Methods

2.1 Introduction

Accurate measurement of execution time is crucial for evaluating the performance of algorithms. In the project were employed two timing methods: `gettimeofday` and NVProf profiler to measure the execution times of the algorithm.

For the CUDA implementation, execution time related to kernels was measured using the `gettimeofday` function alongside the NVProf profiler to gain a comprehensive performance overview. Conversely, in the sequential CPU implementation, the algorithm exclusively utilized the `gettimeofday` function for measuring execution time.

2.2 `gettimeofday`

`gettimeofday` is a function that provides a high-resolution timestamp that allows for precise measurement of time intervals. The basic usage involves capturing the timestamp before and after the code segment of interest and calculating the difference to determine the execution time. Within this project's context, the function has been encapsulated within the `cpuSecond()` function (contained in the file `common.h`), which provides the CPU time, including microseconds, as its return value, with the aim of measuring elapsed times for the two kernels that make up the complete classification algorithm in CUDA, as well as the execution times of the standard C implementation.

2.3 NVProf Profiler

NVProf is a profiling tool provided by NVIDIA for CUDA applications. It offers comprehensive insights into the performance characteristics of GPU-accelerated code, collecting various metrics and providing detailed analysis to identify performance bottlenecks. In the project scope the profiler was used to analyze in detail the behavior of the algorithm in CUDA during the execution phase (only for device with compute capability less than 8.0).

2.4 Comparison

Both `gettimeofday` and NVProf profiler offer valuable insights into the performance characteristics of software implementations. While `gettimeofday` provides a platform-independent solution for basic timing measurements, NVProf offers advanced profiling capabilities tailored specifically for CUDA applications.

In order to obtain more robust measurements of execution times while mitigating noise, each test is iterated five times to calculate both the mean and variance. All data gathered during the executions are saved to files, facilitating comprehensive performance analysis and enabling its graphical representation.

Chapter 3

Datasets

3.1 Introduction

In this chapter, we present the datasets utilized in our project for evaluating the performance of the k-Nearest Neighbors algorithm on GPU and CPU platforms. These datasets include real-world and artificial data, providing diverse scenarios for performance assessment.

3.2 Diabetes Dataset (UCI)

The Diabetes dataset from the UCI Machine Learning Repository comprises eight baseline variables (pregnancies, glucose, blood pressure, skin thickness, insulin, BMI, diabetes pedigree function, age) with the binary label of the associated diagnosis, for 2768 diabetes patients, divided into 2460 individuals for the training set and 308 for the test set.

3.3 Iris Dataset (UCI)

The Iris dataset, also sourced from the UCI Machine Learning Repository, consists of 150 samples of iris flowers, each characterized by four features: sepal length, sepal width, petal length, and petal width. The dataset is splitted in two folds, training set and test set (one third of the training set).

3.4 Artificial Dataset

They were also generated artificial datasets using a custom function that generates clustered data samples distributed around a given mean value with a specified number of features. Each sample consists of one component with the mean value plus random noise and all remaining components with random noise only. Each data point is labeled based on its cluster index, allowing for supervised learning tasks.

Chapter 4

kNN Kernels

To realize the kNN algorithm in CUDA, the implementation process is bifurcated into two main stages: computing the distances between test and training data points (performed by the `knnDistances` kernel) and determining the k nearest neighbors (handled by the `knn` kernel).

4.1 `knnDistances`

When computing in parallel the distances between each test data point and all training data points, the results can be conceptualized as a matrix, with the number of rows equal to the points in the test set and the number of columns equal to the points in the training set. This geometric interpretation facilitates the natural parallelization of k-nearest neighbor calculation. The kernel comprises a grid structure where the size of each block (tile) can be dynamically configured at runtime by specifying the desired dimensions. Additionally, the kernel provides three types of distance measures: Euclidean, Manhattan, and Minkowski. These measures offer different ways to calculate the distance between two points in a multi-dimensional space.

For Euclidean distance, the formula is:

$$\text{Euclidean Distance} = \sqrt{\sum_{i=1}^{n}(x_i - y_i)^2}$$

For Manhattan distance, the formula is:

$$\text{Manhattan Distance} = \sum_{i=1}^{n}|x_i - y_i|$$

For Minkowski distance with parameter p, the formula is:

$$\text{Minkowski Distance} = \left(\sum_{i=1}^{n}|x_i - y_i|^p\right)^{\frac{1}{p}}$$

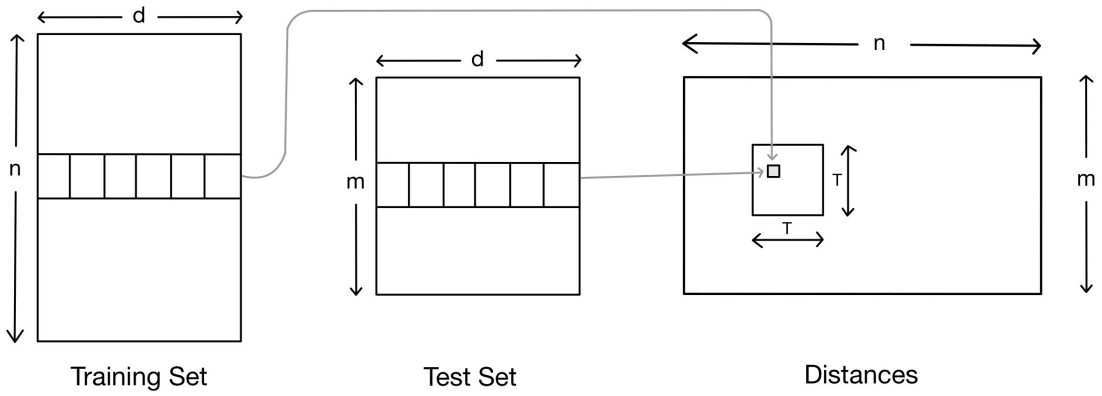

Figure 4.1: Data organization for distances computation, where d is the number of features, n and m are respectively the number of training and test set samples and T is the tile dimension (*blockDim*)

To optimize the performance of the CUDA kernel `knnDistances`, the cache configuration is set using the `cudaFuncSetCacheConfig` function. In this specific kernel, the configuration is set to prefer a 48KB L1 cache and a 16KB shared memory.

Setting the cache configuration to prioritize L1 cache resulted in an acceleration of the distances calculation within the kernel, leveraging the quicker access times provided by these type of memory. This enhancement contributed to an overall improvement in the execution speed of the CUDA kernel.

4.2 knn

After the calculation of distances, the knn kernel engages in the identification of the k nearest neighbors in a *divide et impera* approach, followed by the determination of labels through a majority voting process. This involves counting the contributions of labels from the k neighbors and selecting the class with the highest frequency.

While processing, the assignment of threads working within the device's global memory is based on maximizing the utilization of shared memory for the subsequent algorithmic steps. This optimization is achieved by considering the total available shared memory per block, the k value and the *alpha* parameter, as demonstrated in the following code snippet:

Listing 4.1: Code snippet to calculate the number of threads per block in knn kernel.

```
int maxSharedMemory = getSharedMemoryPerBlock(device);
int itemSize = sizeof(double) + sizeof(int);
int workers = maxSharedMemory/(k*itemSize*(1.5+(double)1/alpha));
workers = nearestPowerOfTwo(workers);
```

where the constant `1.5` is needed to sets an upper bound in the threads allocation and the function `nearestPowerOfTwo(workers)` rounds threads number to the nearest power of two in order to keep it a multiple of the warp size.

For small datasets, this method may result in the allocation of more threads than needed, as observed in the iris database scenario. To avoid this issue, a validation step is implemented to ensure that each thread processes a consistent portion of data. In this case only the factors *alpha* and k dictate the number of threads per block that operate in parallel in each row (train size elements) of the distance matrix, where each thread is tasked to handling *alpha * k* elements within a row, which are subsequently sorted in ascending order.

Listing 4.2: Code snippet for updating the number of threads per block in knn kernel.

```
if (workers > (int)trainSize/(alpha*k)){
    workers = nearestPowerOfTwo((int)trainSize/(alpha*k));
}
```

It is worth noting that in this approch each block in the grid operates simultaneously on individual rows of the distance matrix, thereby leveraging parallelization to enhance the efficiency of processing for batches of test samples across both rows and columns of the distance matrix.

After the initial parallel operation executed by all threads on their respective data portions (first iteration), the results (k smallest distances per thread) are stored in shared memory. This type of memory is specifically allocated to accommodate the data and serve as a 'swap' space for subsequent iterations of the reduction process.

The alpha factor, set by default to 2 (customizable by the user at runtime), governs the selection of active threads in each block for subsequent iterations, ensuring that each one processes a consistent portion of data (*alpha * k*). Consequently, the number of working threads employed decreases over each iteration, according to the formula:

$$\#threads(iter) = \left\lfloor \frac{\text{BlockDim.x}}{\alpha^{(\text{iter}-1)}} \right\rfloor \text{ for } iter \geq 2. \tag{4.1}$$

Parallel computation within a row concludes when the data to be processed falls below the threshold *beta * k* determined by the parameter *beta* (customizable by the user at runtime or defaulted to 4). In such case, only one thread per block (the first one) operates sequentially on the remaining data portion, performing the label prediction for the corresponding test point.

Chapter 5

Test Results and Conclusions

This chapter delves into the testing procedures executed during the project's development phase. Each test was conducted five times, and the accuracy of the predictions was verified using the `checkResult` function, contained in the file `common.h`.

5.1 Test on Alpha Factor

With the aim of analyzing how the *alpha* factor affects the execution time of the `knn` kernel and therefore the overall execution time, which is been defined with the purpose of regulate the degree of parallelization on the rows concerning the distance matrix, a test was devised with alpha ranging from 2 to 8.

Figure 5.1: Plot of the execution time in function of *alpha* factor

As can be seen from the graph, it's evident that higher values of the alpha factor lead to longer execution times for both GPUs. This outcome is inherently expected, as the alpha factor was specifically crafted to control the level of parallelization.

5.2 Test on Block Dimension

In order to investigate the impact of block sizes on the performance of the `knnDistances` kernel, a test was conducted using various squared block, with linear sizes belonging to the set [2, 4, 8, 16, 32], to assess their respective effects on execution time.

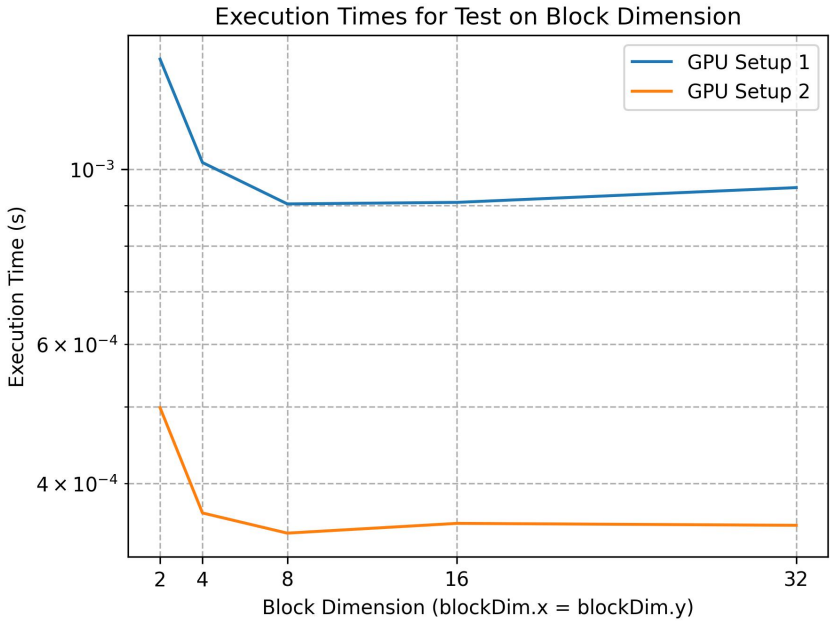

Figure 5.2: Plot of the execution time in function of linear Block dimension

It is possible to observe from the graph how increasing the block size up to the value 8 (64 threads per block) leads to a decrease in execution time and subsequently for higher values a slight increase. As is is well known, enlarging the block size results to an increase in the degree of concurrency among threads (organized in warps of 32) within the Streaming multiprocessors (SMs) as long as the maximum limit of threads that can be instantiated is respected and the amount of hardware resources per SM is sufficient.

5.3 Test on Number of Features

Through this test, the objective is to examine how the algorithm's runtime behavior is affected by an increasing number of artificially generated features, spanning from 10 to 100 with intervals of 10.

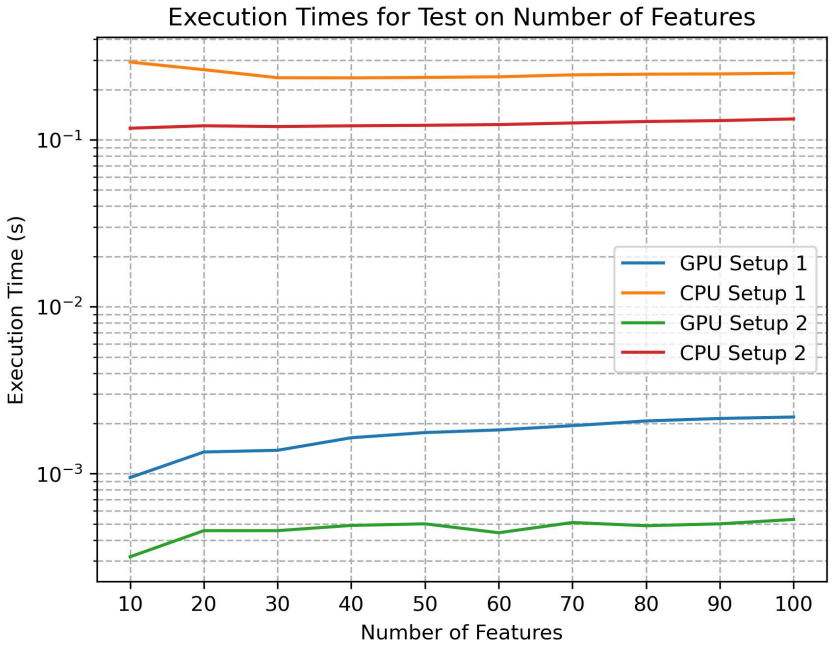

Figure 5.3: Plot of the execution time in function of features number

This test underscores that as the number of input features rises, there is a more pronounced relative increase in execution time for GPUs compared to CPUs. This outcome aligns with expectations, as the computation of distances between two points follows a sequential processing paradigm due to the implementation's reliance on a for loop.

5.4 Test on K Parameter

As the parameter k plays a direct role in determining the portion of data handled by each individual thread (k * $alpha$), it's interesting to analyze how its variation affect the overall execution time of the algorithm, for this purpose, we tested a range of k values from 5 to 50, with increments of 5.

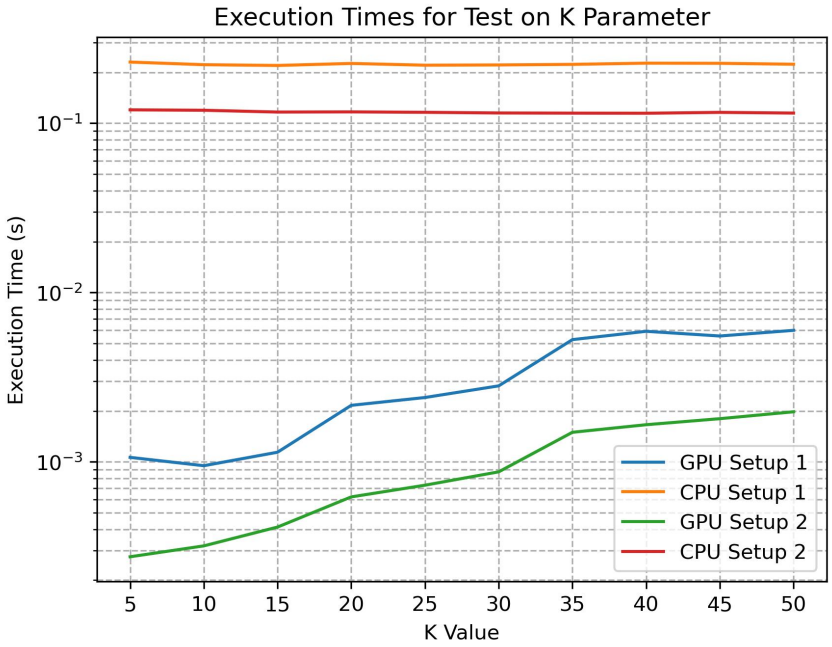

Figure 5.4: Plot of the execution time in function of K parameter

As expected, increasing the parameter k led to a rise in execution time during GPU processing. This result concurs with the algorithm's parallelization approach, where each thread within the knn kernel handles a data portion equivalent to k * *alpha*. Therefore a higher k value leads to more sequential processing of each thread.

5.5 Test on Training Set Size

To examine the impact of training set size on algorithm execution times, various sizes were evaluated, ranging from 1000 to 10000 with increments of 1000.

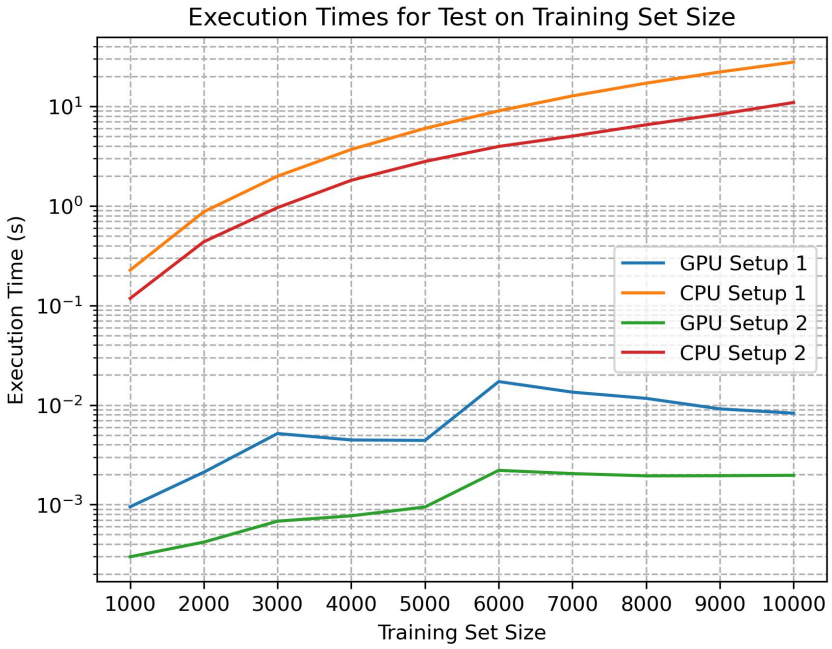

Figure 5.5: Plot of the execution time in function of training set size

Predictably, the execution time trend of GPUs, when compared to CPUs, exhibits an overall shallower slope, leading to a significant scalability in the comparison between the two algorithm implementations.

5.6 Test on Test Set Size

With the aim of analyzing also the impact of the test set size with respect to the algorithm, a test was designed which, with a range of sizes confined between 100 and 1000, at intervals of 100, measures execution times.

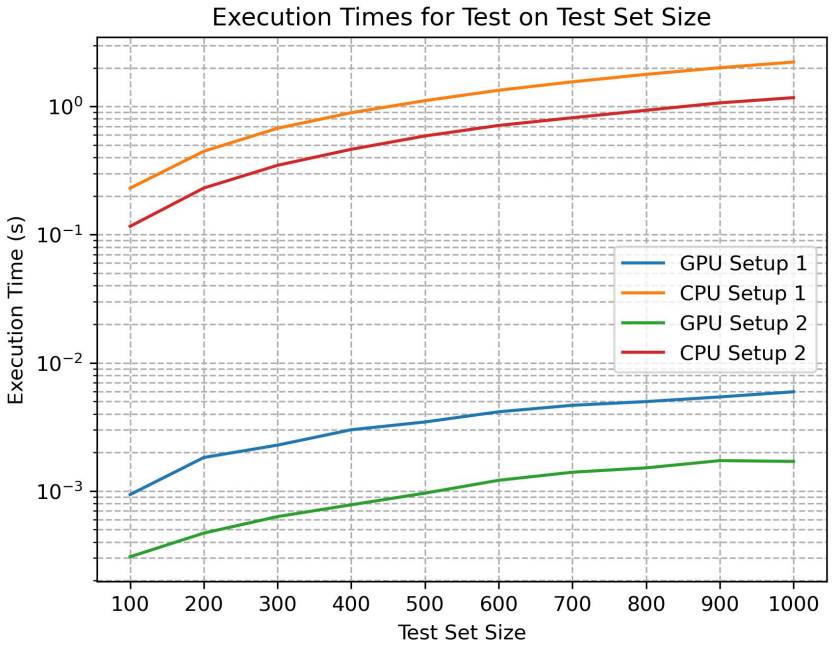

Figure 5.6: Plot of the execution time in function of test set size

As for the training set size test, but with a growing range of query data points, similar considerations can be made by observing how the GPU implementation of the algorithm demonstrates a notable reduction in execution time compared to the CPU counterpart.

5.7 Test on Diabetes Dataset

In order to use not only artificial data, but also real benchmark dataset, a specific test was designed to analyze the behavior of the two implementations of the algorithm on the diabetes dataset, aiming to assess the performance on a medium-sized set of real data.

Hardware	**Execution time**
GPU 1	0.007321 s
CPU 1	4.146595 s
GPU 2	0.001640 s
CPU 2	2.160212 s

Table 5.1: Execution times for diabetes dataset

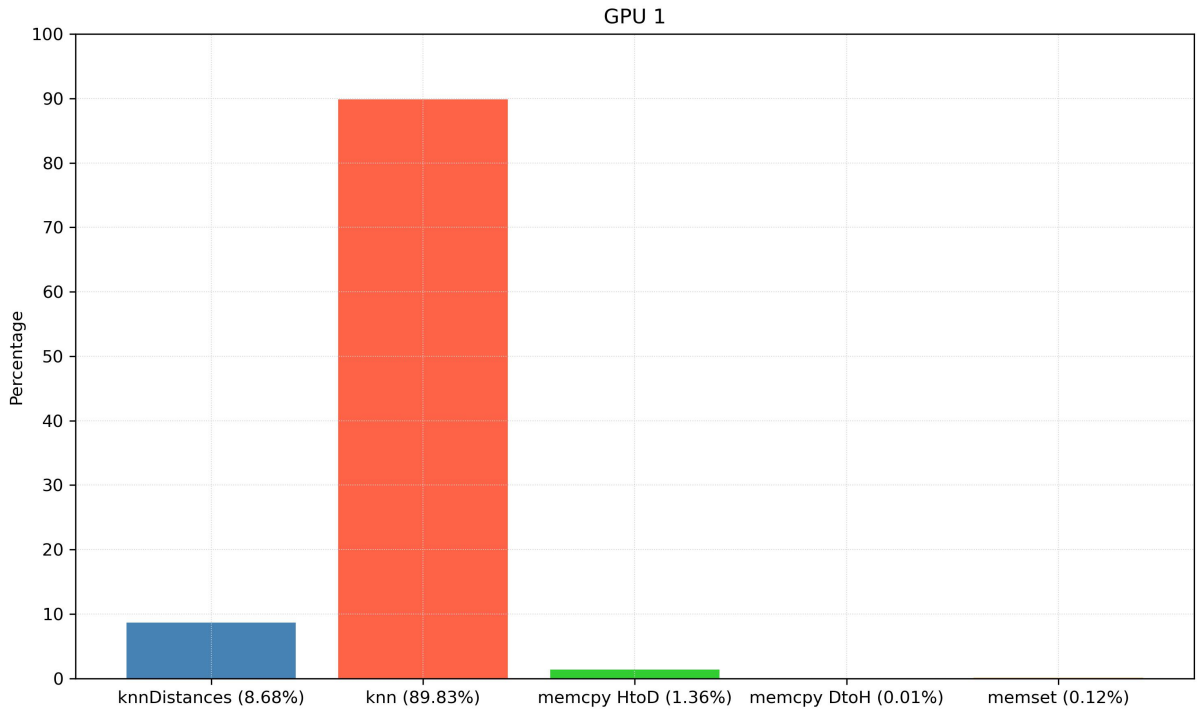

Figure 5.7: Profiling histogram for GPU 1 in diabetes dataset

The table clearly demonstrates a significant gap of video card performance compared to processors. Another noteworthy observation arises when examining the histograms depicting profiling data, indicating that the `knn` kernel significantly influences the overall execution time.

5.8 Test on Iris Dataset

Another test was carried out to assess the performance of both algorithmic implementations using the Iris dataset. The objective was to examine how their performance varies when dealing with small datasets.

Hardware	**Execution time**
GPU 1	0.000481 s
CPU 1	0.003296 s
GPU 2	0.000167 s
CPU 2	0.002221 s

Table 5.2: Execution times for iris dataset

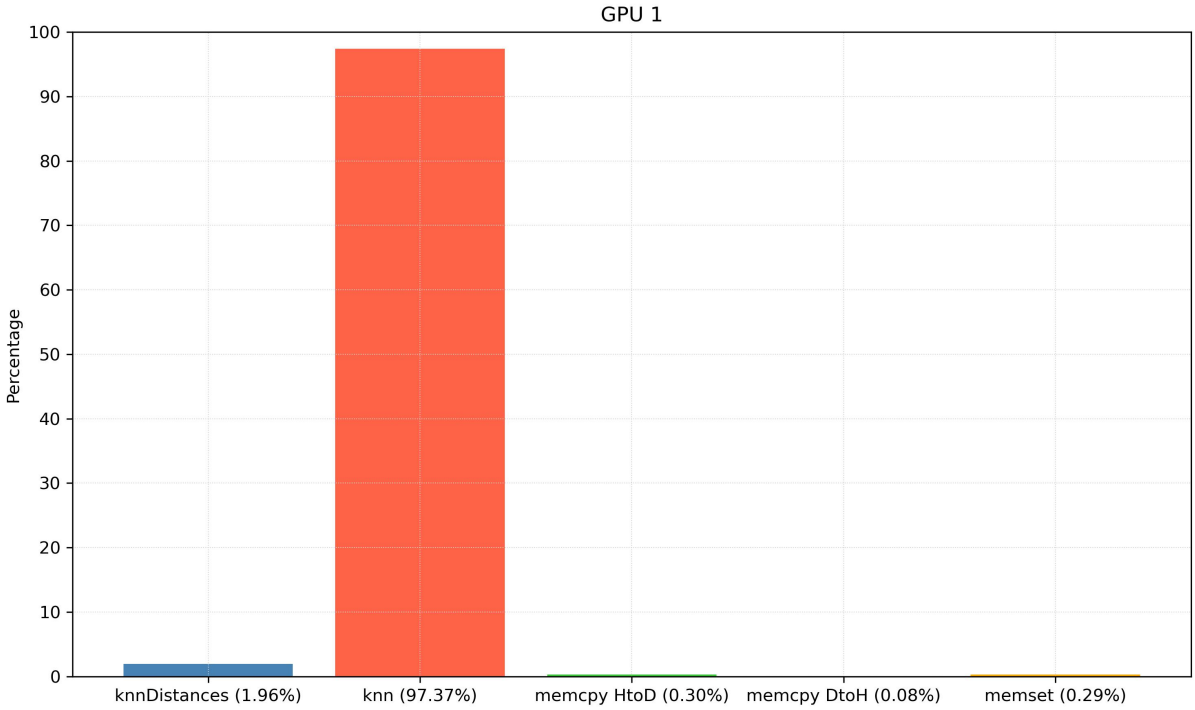

Figure 5.8: Profiling histogram for GPU 1 in iris dataset

While the speedup obtained in this case is smaller of that achieved in the diabetes dataset, the CUDA implementation still presents a considerable advantage in terms of execution time compared to the plain C version of the code. Similarly in this instance, it is evident from the profiling histogram that the `knn` kernel significantly influenced the execution time.

5.9 Conclusions

In conclusion, it is clear the remarkable speedup obtained by employing GPUs over CPUs for the k-Nearest Neighbors algorithm.

The experiments conducted on our hardware setups revealed that GPU-accelerated implementations of the algorithm consistently outperformed their processor counterparts across various dataset sizes and feature dimensions, underscoring the importance of exploiting GPUs for computational tasks that can be parallelised and highlighting the substantial performance gains achievable with such hardware accelerators.

Appendix A

Code Organization

In this appendix, we discuss the organization of the `knn_cuda` project, including a brief description of each code file.

A.1 Directory Structure

The project directory is organized as follows:

- **datasets/** - Contains datasets used for testing algorithms.
  - `diabetes_testing.csv` - Testing dataset for diabetes prediction.
  - `diabetes_training.csv` - Training dataset for diabetes prediction.
  - `iris.csv` - Iris dataset for classification.
- **include/** - Contains header files for function definition.
  - `c_functions.h` - Header file for C functions definitions.
  - `common.h` - Common functions header file.
  - `cuda_functions.h` - Header file for CUDA functions and kernels definitions.
  - `diabetes_functions.h` - Header file for diabetes-related functions.
  - `iris_functions.h` - Header file for iris-related functions.
- **source/** - Contains main files for testing algorithms.
  - `par_knn_artificial_data.cu` - Main CUDA file for generic testing of parallel execution on artificial data.
  - `seq_knn_artificial_data.c` - Main C file for generic testing of sequential execution on artificial data.
  - `par_knn_diabetes.cu` - Main CUDA file for testing parallel execution on diabetes dataset.
  - `seq_knn_diabetes.c` - Main C file for testing sequential execution on diabetes dataset.

  - `par_knn_iris.cu` - Main CUDA file for testing parallel execution on iris dataset.
  - `seq_knn_iris.c` - Main C file for testing sequential execution on iris dataset.
  - **tests/** - Contains main files for specific tests.
    * `artificial_alpha.cu` - Main CUDA file for testing parallel execution on artificial data with different alpha values.
    * `artificial_blockDims.cu` - Main CUDA file for testing parallel execution on artificial data with different block dimensions in distances kernel.
    * `artificial_features.cu` - Main CUDA file for testing parallel execution on artificial data with different number of features.
    * `artificial_features.c` - Main C file for testing sequential execution on artificial data with different number of features.
    * `artificial_k.cu` - Main CUDA file for testing parallel execution on artificial data with different values of k parameter.
    * `artificial_k.c` - Main C file for testing sequential execution on artificial data with different values of k parameter.
    * `artificial_testSizes.cu` - Main CUDA file for testing parallel execution on artificial data with different test dataset sizes.
    * `artificial_testSizes.c` - Main C file for testing sequential execution on artificial data with different test dataset sizes.
    * `artificial_trainSizes.cu` - Main CUDA file for testing parallel execution on artificial data with different training dataset sizes.
    * `artificial_trainSizes.c` - Main C file for testing sequential execution on artificial data with different training dataset sizes.

- **plot_script/** - Contains Python scripts for plotting.
  - `alpha.py` - Script for plotting graphs for test on alpha value.
  - `blockDim.py` - Script for plotting graphs for test on Block dimensions.
  - `features.py` - Script for plotting graphs for test on features number.
  - `k.py` - Script for plotting graphs for test on k parameter.
  - `trainSizes.py` - Script for plotting graphs for test on training set size.
  - `testSizes.py` - Script for plotting graphs for test on test set size.
  - `histogram_diabetes.py` - Script for plotting histograms of the GPU 1 profiling in diabetes dataset.
  - `histogram_iris.py` - Script for plotting histograms of the GPU 1 profiling in iris dataset.

- **run_test.sh** - Bash script for compiling and executing tests.

A.2 Compilation and Execution

In order to replicate the tests, is sufficient to navigate to the project directory and run the `run_test.sh` script. This script automates the compilation and execution of tests, saving all output files related to the execution in a folder named `userName_results/`.

Appendix B

Experimental Setup

B.1 Hardware and Software Specifications

The experimental setup utilized two computer systems with the following specifications:

First Machine

Component	**Specification**
CPU	IntelCore i7-6820HK @ 2.70GHz (4 cores, 8 threads)
CPU Cache	L1 256 KB, L2 1 MB, L3 8 MB
RAM	32GB DDR4 2133MHz
GPU	NVIDIA GeForce GTX 980M @ 1.13 GHz
GPU Architecture	Maxwell
GPU RAM	4GB GDDR5 2505 Mhz
CUDA Capability	5.2

Table B.1: Hardware Specifications of the Experimental **Setup 1**

The experimental environment was configured with the following software:

- Operating System: Ubuntu 22.04.1 LTS
- CUDA Toolkit: 12.3
- C Compiler: GCC Version 11.4.0
- CUDA Compiler: NVCC Version 12.3.103

Second Machine

Component	**Specification**
CPU	IntelCore i7-13700HX @ 2.10 GHz (16 cores, 24 threads)
CPU Cache	L1 80 KB/Core, L2 24 MB, L3 30 MB
RAM	32GB DDR5 5600MHz
GPU	NVIDIA GeForce RTX 4070 @ 2.17 GHz
GPU Architecture	Ada Lovelace
GPU RAM	8GB GDDR6X 8001 Mhz
CUDA Capability	8.9

Table B.2: Hardware Specifications of the Experimental **Setup 2**

The experimental environment was configured with the following software:

- Operating System: Ubuntu 22.04.1 LTS
- CUDA Toolkit: 12.3
- C Compiler: GCC Version 11.4.0
- CUDA Compiler: NVCC Version 12.3.107

Bibliography

[1] Quansheng Kuang, and Lei Zhao. *A Practical GPU Based KNN Algorithm*. School of Computer Science and Technology, Soochow University, 2009.

[2] Ahmed Shamsul Arefin, Carlos Riveros, Regina Berretta, Pablo Moscato. *GPU-FS-kNN: A Software Tool for Fast and Scalable kNN Computation Using GPUs*, 2012.

[3] Guoyang Chen, Yufei Ding, and Xipeng Shen. *Sweet KNN: An Efficient KNN on GPU through Reconciliation between Redundancy Removal and Regularity*, Computer Science Department North Carolina State University.

[4] John Cheng, Max Grossman, Ty Mckercher. *Professional Cuda C Programming*.