High-Performance Data and Graph Analytics - Fall 2023 Contest

Galli Filippo

Abstract

As applications and websites get more data, the challenge of clustering similar elements or user profiles arises. Nowadays, to solve this, many companies use an algorithm called K-Nearest Neighbors (KNN) which was developed by Evelyn Fix and Joseph Hodges in 1951.

KNN is a highly scalable and customizable algorithm that works straightforwardly: for each input query point, it calculates the distance between the query and all points in a reference set and returns the first K results ordered by descending similarity.

With the current amount of data available practically in real-time, the need to process them quickly and in large quantities has led the CPUs to saturation. It brings experts to implement optimized versions of this algorithm on GPUs.

My work is about how various levels of parallelization affect the performance of this algorithm in GPU.

Keywords

GPU, Cuda, C, C++, KNN, K-Nearest Neighbors

Introduction

As Graphic Processing Units (GPUs) became popular, developers realized the potential to use the computational power of these devices for calculations beyond just graphics. GPUs are designed to perform well with SIMD (Single Instruction Multiple Data) workloads, making them better for tasks that involve a "small" set of instructions that need to be repeated many times on a large amount of data. K-nearest neighbors (KNN) is an example of such a task.

To achieve high throughput, vendors create boards with many readapted CPUs. Each of these CPUs is simpler than the CPUs we are accustomed to, which means that GPUs perform exceptionally well on highly parallel workloads. However, on single-core tests, GPUs are far behind CPUs.

Problem

This is the final project of the course "High Performance Graph and Data Analytics" (HPDGA) attended at Politecnico of Milan. The goal is to develop a K-Nearest Neighbors (KNN) algorithm that runs on Nvidia's GPU using C, C++ and Cuda without any external library.

The main idea that directed the development is to simplify the workflow associated with each thread by increasing the parallelization of the code. This idea is followed by the intrinsic nature of GPU: many simple works repeated on a large set of data.

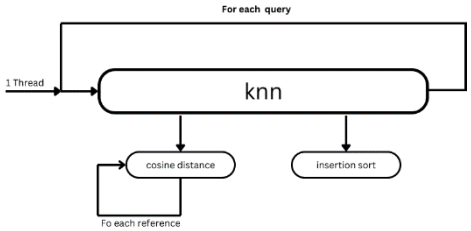
Implementations

Starting from a CPU version given by our lecturer Ian, I developed four incremental solutions in performance, which are "Solution1.cu", "Solution2.cu", "Solution3.cu" and "Solution4.cu".

My primary goal was to maximize parallelism and decrease the work done by each thread. Because of this there has been less care about using more efficient data structures than one-dimensional arrays or optimized memory access patterns.

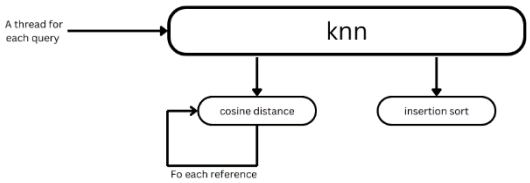
Solution 1

I began tackling the challenge by converting the CPU version to a GPU one to find the primary bottleneck and address it. However, while this approach was helpful at the outset, it has its limitations in terms of performance. The main issue with this solution is the GPU itself, which is perfected to maximize throughput at the expense of individual core performance. As a result, using it for a single-core task isn't ideal, and I found this solution to be slower than its CPU counterpart.



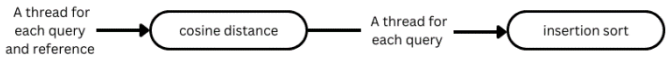
Solution 2

As a direct successor of Solution 1, in this solution, I start to decrease work-per-thread. The main improvement is the parallelization of the function called *knn*, each thread does all calculations for one query. So, it calculates every reference-query distance for its specific query.



Solution 3

The idea of this solution is to parallelize the function which calculates cosine distances. In this way, each thread computes one reference-query distance and not all the reference-query distance as in the earlier solution.



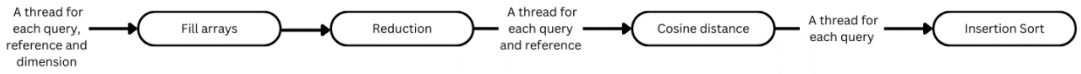
Solution 4 – Draft

The primary goal of this solution is to reach the maximum possible level of parallelization. To achieve this, I have deconstructed the cosine distance function into separate functions. This allows me to adjust the number of threads to optimize the performance of each module.

The decomposition brings me to have three functions instead of one:

- *fill_gpu*: calculates one dimension of distance for each reference-query distance.
- *reduce0*: performs a reduction of the previously calculated data.
- *cosine_distance_gpu*: perform the divisions of the reduce0's data to have cosine distance.

By breaking down the work units in the cosine distance function, we can better parallelize each one.



Please keep in mind that the solution provided is still a work in progress, as I have faced some time constraints during its development. Therefore, it has not undergone extensive testing like our other solutions.
My primary focus is on improving the reduction process and finding ways to optimize the *fill_gpu* function for faster performance.

Performance Comparison

Disclaimer 1: *During tests, Solution 1 took so much time with the third and the fourth set of parameters (parameters 2 and parameter 3) that Google Colab doesn't return the results after hours of working.*
Disclaimer 2: *Solution 4 is work in progress so, at the moment of writing, it doesn't have realistic comparable values to other solutions.*

Each solution is evaluated on samples of parameters given in *knn.cpp*, all parameters to set are:

- **ref_nb**: number of point references taken by knn.
- **query_nb**: number of query points for which knn must return the k-nearest references.
- **dim**: dimension of each point.
- **k**: how many references are taken in increasing order of distance.

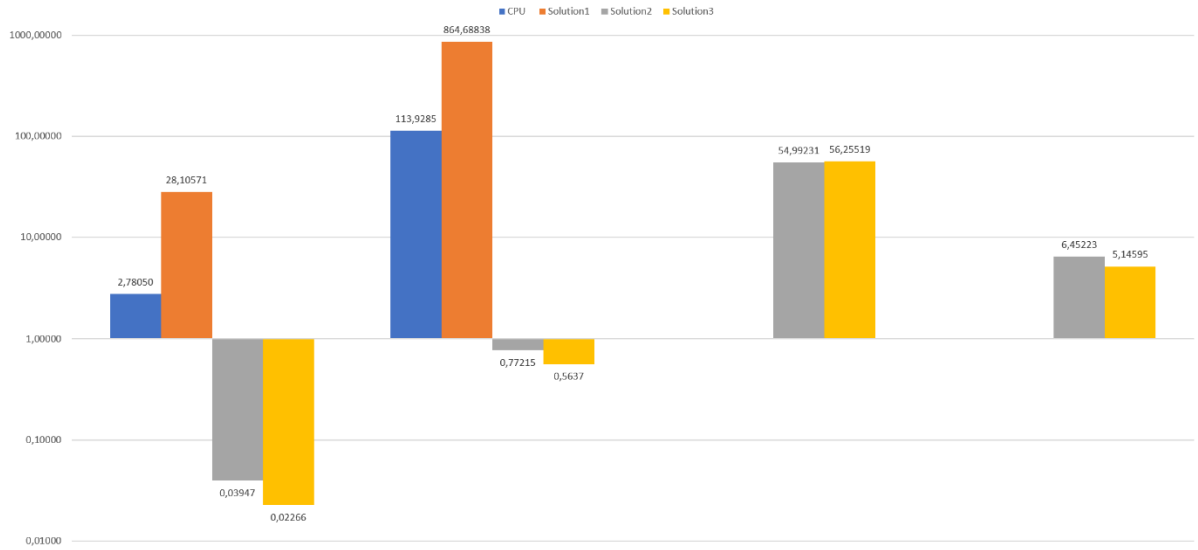
The sets of parameters on which I test my solutions are:

N° set of parameters	**0**	**1**	**2**	**3**
ref_nb	4096	16384	163840	16384
query_nb	1024	4096	40960	4096
dim	64	128	128	1280
k	16	100	16	16

In the following table there are times of execution of each solution in second:

N° set of parameters	**0**	**1**	**2**	**3**
CPU	2,7805	113,9285	~ 30.000	~ 5.000
Solution1	28,10571	864,68838	/	/
Solution2	0,03942	0,77215	54,99231	6,45223
Solution3	0,02266	0,5637	56,25519	5,14595

To visualize earlier results, I make a figure with times (in seconds) in logarithmic scale as y-axis and the parameter set as x-axis. To correctly read it, smaller values are better:



Further Improvements

As this work could be improved and modified by anyone, I think that the most promising features to develop are:

- Use the potential of 2D or 3D block and grid dimensions.
- Develop some 2D or 3D data structures to manage data.
- Using caching methods to avoid some access to global memory.

The goal of the first two improvements is to use optimized built-in functions to transfer and manage 2D\3D data structure and to create simpler indexes than flattened arrays.
The third improvement is focused on cutting the time of accessing global memory.

Conclusions

These results are based on a small number of tests, and each solution may not be perfected from every point of view. However, we can conclude that a higher level of parallelization leads to faster execution times. This observation was held for every set of parameters assessed.

It's worth noting that due to the GPU's internal structure and organization, purely consequential code performance is significantly worse compared to the CPU.