High-Performance Data and Graph Analytics - Fall 2023 Contest

Table of Content

Abstract

This is the final project of the course "High Performance Graph and Data Analytics" (HPDGA) attended at Politecnico of Milan. The challenge is to develop a K-Nearest Neighbors (KNN) algorithm that runs on GPU Nvidia using C, C++ and Cuda as programming languages. Starting from a CPU version given by our lecturer Ian, I developed my own solutions which are "Solution1.cu", "Solution2.cu", "Solution3.cu" and "Solution4.cu". The main idea that directed the development is to simplify the work associated with each thread. So, as I will describe better in the Implementations chapter, this is the focus and evolutionary steps of each solution.

Implementations

From each solution to the following one I decrease the work-per-thread but there is an increment of the parallelization level of each task. Apart from data-coalescent, I haven't developed any other technique to improve memory access patterns or anything else.

Solution 1

I started working on the challenge by transposing the CPU version to a GPU one to identify the major bottleneck and work on it. However, as I will discuss in detail in the Performance Comparison chapter, this approach, although useful as a starting point, has several limitations in terms of performance. This is because GPUs are not optimized to work on sequential code like CPUs. As a result, I found that this solution is slower than its CPU counterpart.
Mettere grafichino relazioni funzioni

Solution 2

With "Solution2" I start to decrease work-per-thread. The main improvement is the parallelization of the function "knn_gpu". Each thread calculates $\frac{\textnormal{query number}}{1024}$ queries. So for each query, it calculates every reference-query distance. In this way, all the work done by 1 thread is spread across 1024

threads.
Mettere grafichino relazioni funzioni

Solution 3

The idea of this solution is to parallelize the function "cosine_distance". In this way, each thread computes one reference-query distance and not all the reference-query distance as in the previous solution. This is possible thanks to a massive parallelization and an increase of working threads, specifically $\textnormal{query number} \cdot \textnormal{reference number}$.
Mettere grafichino relazioni funzioni

Solution 4 - Draft

The main goal of this solution is to achieve the highest possible level of parallelization. To accomplish this, I have performed a functional decomposition of every function, except for the insertion sort function. This allows me to adjust the number of threads to better suit the needs of each logical module. For instance, I created a function called "fill_gpu" that calculates one dimension of one reference-query distance and fills an auxiliary array called "dots".

It's worth mentioning that this solution is still a work in progress due to time constraints. However, there are still many possibilities for improvement such as the reduction process hasn't been optimized yet.
Mettere grafichino relazioni funzioni

Weak Point

Each solution has some lacks of optimization. I think the possible weak points of my solutions are:

- the use of only 1 dimension for grid and block;
- the massive use of global memory;
- using flattened array and not 2D or 3D structured to work with data;

Perfomance Comparison

I tested all of my solutions and CPU one to 3 possible settings of parameters gave me by default.
Disclaimer: I didn't try the second possible setting of parameters because it has so large numbers that Google Colab returned me a Memory Allocation Error and during the test, Solution 1 and CPU took so much time that it only confused the chart about times of executions.

Parameter	**0**	**1**	**3**
ref_nb	4096	16384	16384
query_nb	1024	4096	4096
dim	64	128	1280
k	16	100	16

In the above matrix, there are the names of the variables in the code which correspond to:

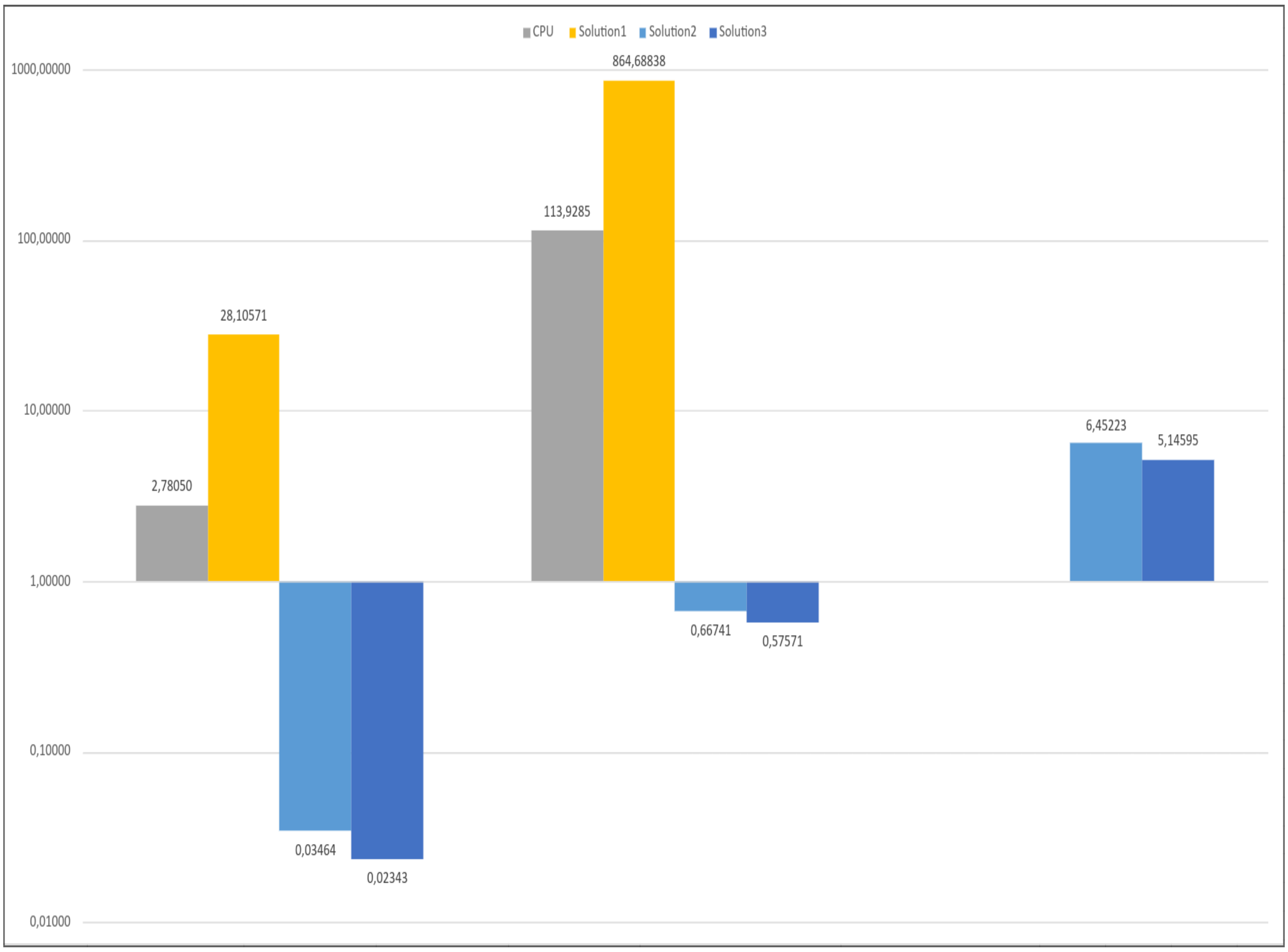
- ref_nb $\Rightarrow$ the number of references;

- query_nb $\Rightarrow$ the number of queries;
- dim $\Rightarrow$ the dimension of both references and queries;
- k $\Rightarrow$ the number of the element to consider as the final result.

We obtain the following times of run (in seconds):

$Solution /Parameter$	0	1	3
CPU	2,78050	113,9285	
1	28,10571	864,68838	
2	0,03464	0,66741	6,45223
3	0,02343	0,57571	5,14595

To better visualize the difference of perfomance I made this chart, with the y-axis in logarithmic scale, to see the differences of times (less is better):


CPU
Solution1
Solution2
Solution3