GPU Computing
Parallel implementation of Dijkstra's Algorithm

Tricella Davide 08361A

August 14, 2023

Instructor: Professor Grossi Giuliano

Abstract

The purpose of this paper is to describe the implementation and benchmarking of various parallel implementations of Dijkstra's Algorithm to solve the shortest path problem.

Contents

1 Introduction

The problem of shortest path in a graph consists in finding the path, from node A to node B, which minimizes the sum of edges weights of which the path is composed.

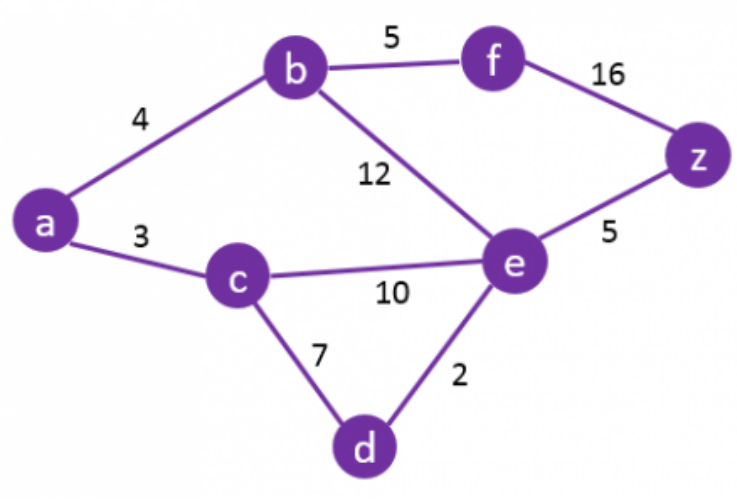

Figure 1: Graph with edge weights

In particular, the algorithm used for this document aim to solve the all-pairs shortest path problem, which consists of finding the lenght of the shortest path for all the pairs of nodes.

The algorithm assumes that there are not negative weights and that the graph is undirected without isolated parts.

This problem is tackled in a lot of practical applications, like road networks path-finding, telecommunication routing and robot navigation.

2 Dijkstra's algorithm

To solve the problem introduced before, there are several solutions which have been studied and improved troughout the years. The option chosen for this task has been the Dijkstra's algorithm, which has been implemented using the paper [1] as a guidance.

This is a very well known algorithm, initially designed at the end of the 1950's, and used in a huge number of different applications.
The first version implemented was basically a parallel extension of the sequential algorithm, then a more advanced approach has been taken for the improved version.

2.1 Sequential Version

The basic algorithm solve the problem of shortest path between one Source Node and all the other nodes. The sequential version uses a for loop to launch this version on every node of the graph, so at the end of the loop every node possesses the shortest path to every other node.

The main elements used during the computation are:

- Graph Adiacency Matrix: which represents the graph indicating the weight of the edge which connects a node to another
- Vt Set: used as a stop condition and to check if a node has already a minimal path assigned
- l Array: which contains the minimal path found for a certain node at a certain moment

We can divide the procedure into two main passages:

- Initialization
- Main loop

During the first phase the Vt set is created containing only the Source Node, then the l Array is initialized, with the direct connections from the Source Node to every other node, putting a symblic value of "infinity" where a direct connection is not available.

The the main loop begins, and continues until all the nodes are present in the Vt set. The Main cycle, is composed of two internal phases:

- Local minimum search: between the elements of l not included in Vt we look for the minimum weight. Then we add the node selected to Vt.
- Update loop: the value of l corresponding to the node selected before gets updated with the minimum between the current value and the sum between the value selected during the minimum search and the weight present in the matrix. This part verifies if the value calculated by the search brings to a path actually shorter than the one stored.

When the Vt set is equal to the set of nodes of the graph, the Main loop terminates and the algorithm returns the l Array, where all the shortest paths are stored.

To solve the all pairs Shortest Paths problem we add an external loop that launch the algoritmh for every row of the original Adiacency Matrix, and at the end returns a new matrix, composed of the various l arrays computed for every node.

2.2 Basic parallel version

The idea behind the initially implemented parallel version is simple: to solve the problem we have to launch the algorithm |V| times, where V is the set of nodes of the graph. Using only one block of the GPU, every thread executes the algorithm for one node of the graph, and at the end insert the results in the correct row of the matrix in Global Memory.

The shared memory is not used because the variables needed for the computation are all locally present inside the thread, which is the main advantage of this first technique, because there are not requirements of any kind on inter-process communications. Each thread works completely isolated from the others.

This implementation requires a thread per node of the graph, which still uses the various for loops present in the sequential version, this fact make this implementation simple to program, with a decent speedup, provided the graph is big enough.
The main problem is that it is not taking full advantage of the Gpu capabilities, which is the weak spot that the second version tries to address.

2.3 Improved parallel version

This version aim to exploit more the parallel execution of the GPU, using one Block per node of the graph, and one thread for every node to evaluate in the block.

3 Implementation details

4 Benchmarking

4.1 Sequential vs Parallel

References

[1] Vivek Sarkar. *Parallel Graph Algorithms*. 2008. URL: https://www.cs.rice.edu/~vs3/comp422/lecture-notes/comp422-lec24-s08-v2.pdf.