Architectures And Platforms For Artificial Intelligence
Module 1: Bellman-Ford parallel implementation

Lorenzo Galfano
Master's Degree in Artificial Intelligence, University of Bologna
lorenzo.galfano@studio.unibo.it

1 Introduction

The Bellman-Ford algorithm is a fundamental algorithm in the field of computer science and graph theory. It is used to find the shortest paths from a single source vertex to all other vertices in a weighted graph, even in the presence of negative weight edges, provided that the graph does not contain any negative weight cycles reachable from the source vertex.

One of the key features of the Bellman-Ford algorithm is its ability to handle graphs with negative edge weights, making it suitable for a wide range of applications, including network routing protocols, traffic planning, and resource allocation in distributed systems.
For the scope of this project the algorithm will be parallelized with the OpenMP and CUDA frameworks and see if its performance can be improved.
Both frameworks have their peculiarities:

- **OpenMP** achieves parallelization by distributing the computation across multiple threads, it primarily targets shared memory systems, such as multi-core CPUs.
- **CUDA** on the other hand exploits the massively parallel architecture of GPUs, which consist of thousands of smaller processing units called CUDA cores. It uses a Single Instruction, Multiple Thread (SIMT) execution model, where many threads execute the same program code in parallel.

2 Bellman-Ford Algorithm

The algorithm works by iteratively relaxing the edges of the graph, gradually improving the estimates of the shortest path distances until they converge to their optimal values. At each iteration, the algorithm considers all the edges in the graph and updates the distance estimates based on the relaxation criteria.

The relaxation criteria states that:

- For the graph having V vertices, all the edges should be relaxed V-1 times to compute the single source shortest path.
- In order to detect whether a negative cycle exists or not, relax all the edge one more time and if the shortest distance for any node reduces then we can say that a negative cycle exists.

That means that if we relax the edges V times, and there is any change in the shortest distance of any node between the $(V-1)^{th}$ and V^{th} relaxation then a negative cycle exists, otherwise it does not exist.

To provide a clearer explanation of the algorithm's implementation, I have incorporated pseudocode:

Algorithm 1 Bellman-Ford

```
procedure BellmanFord($G, s$)
    for $i \leftarrow 1$ to $|V|$ do
        $d[v] \leftarrow \infty$
    end for
    $d[s] \leftarrow 0$
    for $i \leftarrow 1$ to $|V| - 1$ do
        for each edge $(u, v) \in E$ do
            if $d[u] + w(u, v) < d[v]$ then
                $d[v] \leftarrow d[u] + w(u, v)$
            end if
        end for
    end for
    for each edge $(u, v) \in E$ do
        if $d[u] + w(u, v) < d[v]$ then
            return "Graph contains negative weight cycle"
        end if
    end for
    return $d[]$
end procedure
```

The time complexity of the algorithm, as evident from the pseudocode's nested for loop, is $\mathbf{O(}|V|\mathbf{*}|E|\mathbf{)}$, where $|V|$ represents the number of vertices and $|E|$ represents the number of edges. On the other hand, the space complexity is $\mathbf{O(}|V|\mathbf{)}$ since we need to store an array of size $|V|$, specifically the distance array, in memory.

2.1 Parallelization

Following the pseudocode provided above, the algorithm will be parallelized in three different parts:

- **Initialization of the Distance Array**: This phase involves initializing the distance array, where each element corresponds to a vertex in the graph.
- **Relaxation Step**: Parallelization occurs within the inner loop of the relaxation step, which iterates over each edge of the graph. Since the outer loop cannot be parallelized due to its dependency on the results of the inner loop, parallelization is applied at the level of individual edge relaxations.
- **Search for Negative Cycles**: The last parallelization opportunity arises during the search for negative cycles in the final for loop.

3 Graphs and visualization

The input graphs utilized in the program are integral to evaluating the algorithm's speed and scalability, particularly when comparing the sequential and parallel versions. As we aim to assess the algorithm's performance across varying graph sizes, the generation of increasingly larger graphs becomes essential.
To generate such graphs I created a very simple python program called *graph_creator.py*, as searching online provides scarse results. The graphs created by the program are not intended to provide meaningful examples (in terms of the distance array, i.e. the values might be infinity for most of the indexes), but to serve as large-scale test cases for the algorithm's evaluation.

Visualizing results plays a pivotal role in problem evaluation. To streamline this process, I developed an additional program named *speedup.py* specifically for generating insightful visualizations which will be further discussed in section 6.

4 OpenMP

In OpenMP, a wide range of directives is available to assist in parallelizing code effectively. In this case, the most commonly used directives were ***pragma omp parallel*** and ***pragma omp for***. These directives were collapsed into ***pragma omp parallel for***, which, as the name suggests, parallelizes for loops. If no schedule option is specified, the loop is evenly distributed among threads prior to its execution.
This directive can be used in the three loops mentioned in section 2.1, however, if we simply parallelize the nested for loop we would be dealing with **race conditions and inconsistent results**. The problem arises from the update of the distance array inside the inner loop. Without additional directives, multiple threads may enter the if statement before the variable is correctly updated, potentially yielding incorrect results. This effect is enhanced with growing number of threads and bigger interconnected graphs.
To solve this issue a critical section was implemented, using the ***pragma omp critical*** directive:

```
if $d[u] + w(u, v) < d[v]$ then
    $new\_dist \leftarrow d[u] + w(u, v)$
    $\#pragma\_omp\_critical\{$
    if $new\_dist < d[v]$ then
        $d[v] \leftarrow new\_dist$
    end if
    $\}$
end if
```

After a thread successfully enters the if statement the new d[v] value will be temporarily stored into a new variable, called new_dist. Threads will now enter the critical section one at a time ensuring the correct update of the d[v] variable. The update step is done after checking if new_dist is still lesser than d[v].
Another aspect to consider is the placement of the critical section within the parallelized code. Instead of creating an additional temporary variable and if statement, we could have utilized the critical section directly within the first if statement. However, critical sections drastically slow down the process as overhead is introduced in terms of synchronization between threads. Implementing the critical section before the first if statement would mean that each thread has to halt in **each iteration**, for a total of **|V|*|E| synchronizations**, which would slow the program too much and would result in little or no optimization in terms of speed.
Finally, the presence of a negative cycle was checked using the ***reduction(|:check)*** clause. After the completion of the loop, a logical OR operation was performed across all thread-private copies of the variable *check*. If the result was equal to one, it indicated the presence of a negative cycle.

5 CUDA

CUDA separates the program into the **CPU program** (host program) and the **GPU program** (device program), to execute the latter, CUDA utilizes kernel launches
Kernel launches are characterized by the specification of the number of blocks and the number of threads per block. It's important to note that the first parameter, representing the number of blocks, varies depending on the specific parallelization section. In the implementation, three distinct kernel calls were implemented to address different parallelization tasks. The number of blocks is determined by the formula **(tV + BLK_DIM - 1)/BLK_DIM** when initializing the distance array, and **(tE + BLK_DIM - 1)/BLK_DIM** in the other two loops. This formula ensures that the problem size, whether it's the number of vertices or edges, is evenly divided among the blocks, with each block processing a contiguous portion of the data.
The number of threads for block is constant and is equal to **BLK_DIM** which was set to **256**, other values could've been chosen spanning from the minimum being 32 and the maximum being 1024.
Two different kernel launches were created for each parallization step, dividing the parallel implementation launch from the sequential one (both number of blocks and threads for block are 1).
When entering a parallel kernel launch each thread will compute it's index and check whether it's out of bounds or not (the launch will end if not in bound), while in the sequential implementation a

simple for loop was used.
Dealing with the nested loop was fairly simple in CUDA as the library offers the **atomicMin()** operation, which can be used to correctly update the distance array without leading to any race condition.
Finally **cudaDeviceSynchronize()** was used to ensure proper synchronization between CPU and GPU after the completion of a kernel launch.

6 Results

To present the results of this study, I adopted a two-step approach for efficient analysis and visualization. At the end of both the openMP and CUDA programs I wrote the results on a .txt file, such that this raw data could then be used by ***speedup.py*** for the visualization part. The results of sequential openMP and CUDA were computed by running the program with one thread/one block and one thread for block.
Firstly I tested the speedup obtained with growing number of vertices and with growing number of edges, for openMP the speedup was computed using the parallel run with four threads as it's the fastest one, this result will be shown later in this section.

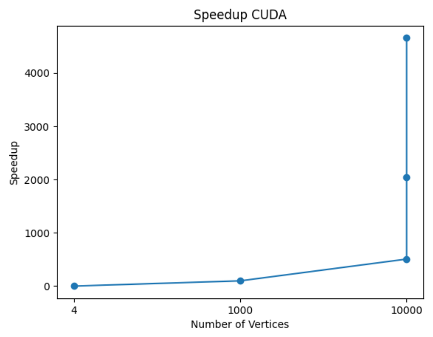

Figure 1: Speedup of CUDA program for growing number of vertices

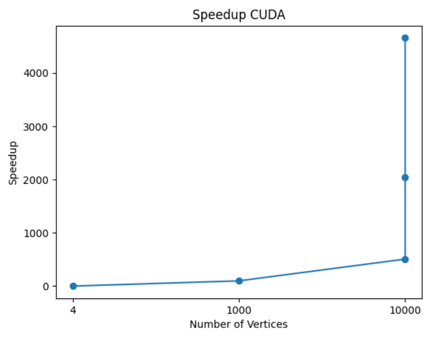

Figure 2: Speedup of CUDA program for growing number of edges

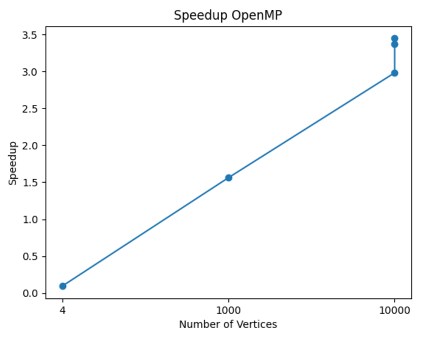

Figure 3: Speedup of openMP program for growing number of vertices

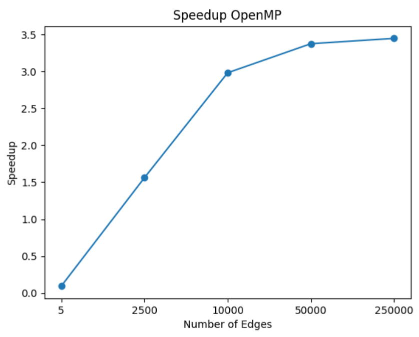

Figure 4: Speedup of openMP program for growing number of edges

The last three examples have the same number of vertices but an increasing number of edges, the reason is that Bellman-Ford performs really well on sparse graphs with respect to other Shortest Path algorithms such as **Dijkstra's Algorithm** (O($|V^2|$) time complexity) or the **Floyd-Warshall Algorithm** (O($|V^3|$) time complexity). With its time complexity of O($|V|*|E|$), Bellman-Ford excels particularly in scenarios where graphs are sparsely interconnected. Therefore, I tested the algorithm's efficiency under conditions of consistent vertex counts but progressively denser edge connections.

The observed results align with expectations, the parallel version of the algorithm is faster than the sequential one for both the implementations. The speedup increases as the size of the graphs grows, highlighting the scalability of this implementation, CUDA stands out reaching a **4000x** speedup computed for the last example.

In contrast to CUDA, openMP was evaluated across a range of thread counts, spanning from 2 to 64. Plots illustrating the speedup and elapsed time were generated to visualize the performance trends with increasing thread numbers.

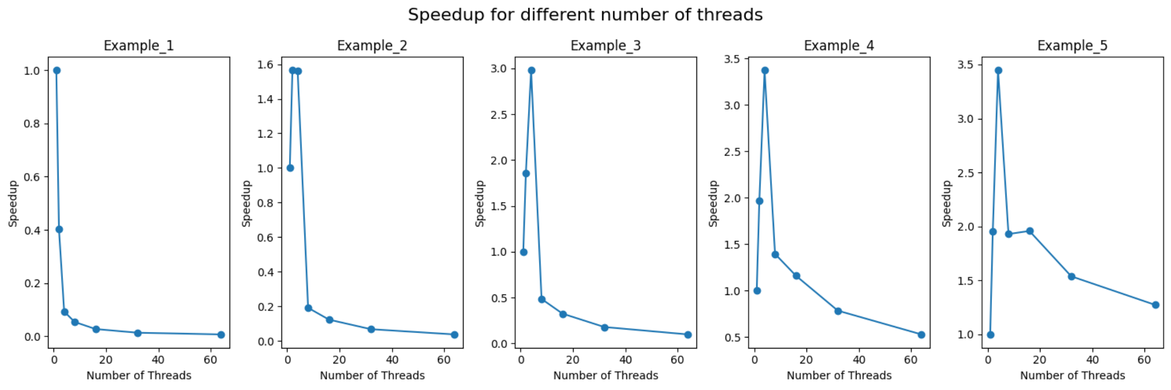

Figure 5: Speedup of openMP program for growing number of threads

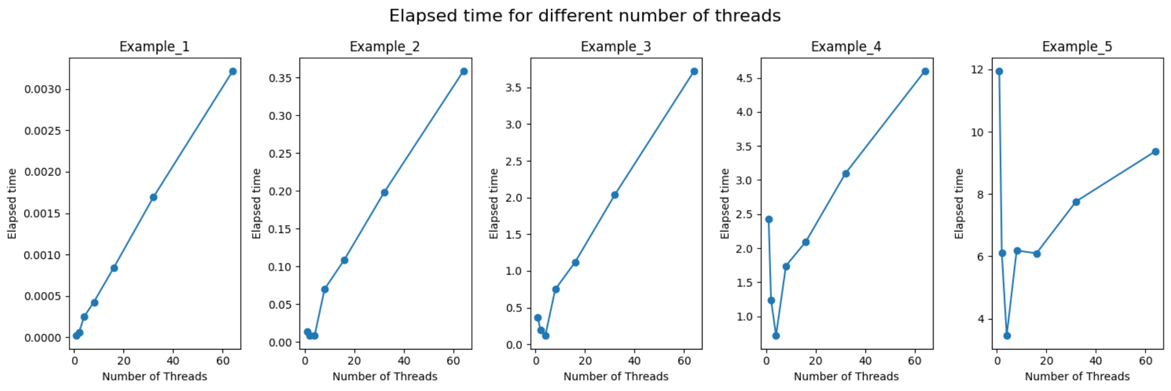

Figure 6: Elapsed time of openMP program for growing number of threads

Two key observations emerge from the graphs. Firstly, in the initial example, the **parallel version exhibits slower performance**. This discrepancy can be attributed to the graph's small size, comprising less than 10 vertices and edges. Consequently, the parallel process incurs overhead in synchronizing threads with the critical section, resulting in suboptimal performance compared to the sequential counterpart.
Secondly, a noteworthy trend is the consistent **speed superiority of the program utilizing 4 threads across most cases**. This phenomenon likely stems from the hardware configuration of the Slurm machines, where each CPU possibly incorporates up to four cores. As a result, employing more threads may prove inefficient, thereby rendering the four-threaded implementation as the fastest alternative.
Finally, I aim to illustrate the runtime durations for both sequential and parallel CUDA processes, as they offer valuable insights for further analysis:

Table 1: Wall-clock time for Sequential and Parallel CUDA Processes

Example	Sequential	Parallel
1	0.001241	0.001418
2	0.367062	0.003696
3	10.629396	0.020932
4	67.080635	0.032721
5	336.078247	0.072134

The results underscore the remarkable efficiency of CUDA parallelization, demonstrating its ability to **swiftly process even large graphs with minimal time overhead**. In contrast, the sequential version exhibits considerably longer execution times. This disparity can be attributed to the inherent optimization of CUDA for data-parallel workloads, leveraging the GPU's architecture.
It appears that CUDA also demonstrates suboptimal performance when parallelizing small graphs and therefore the sequential version may be more suitable for those graphs.

7 Conclusion

Both CUDA and OpenMP implementations offer distinct advantages, catering to different computational scenarios. **CUDA excels in scenarios involving a vast quantity of operations**, leveraging the GPU's architecture for superior performance. However, for smaller graphs, **OpenMP remains a viable option**, albeit not as fast as CUDA. Additionally, it's noteworthy that the CUDA sequential version does not perform as efficiently as the OpenMP sequential implementation for larger graphs.

In conclusion, CUDA demonstrates superior scalability, making it ideal for processing large quantity of data efficiently. Conversely, while OpenMP may not match CUDA in terms of speed, it still presents a reliable parallel implementation option, particularly for smaller-scale computations.