Parallel Implementation of the Bellman-Ford Algorithm

Călin Diaconu

January 8, 2024

Abstract

This document describes the results of two parallel implementations of the Bellman-Ford algorithm [1] [4] [6], one based on the OpenMP [2] software library, and the other one based on the CUDA [7] software library. The document starts with a description of the implementation, it continues with evaluation procedures and results, and ends with ideas about future improvements. The appendix contains pseudocode snippets and all the tables with the experimental results.

1 Algorithm Details

The Bellman-Ford algorithm solves the single-source shortest-path problem in the general case, supporting also negative-valued weights for the edges.

It runs in time $O(VE)$, since the initialization takes $\Theta(V)$ time, each of the $|V|-1$ passes over the edges takes $\Theta(E)$, and the loops takes $O(E)$ time.

1.1 Algorithm Structure

The algorithm is taken from the Introduction to Algorithms book [3], and has the procedure described in the Appendix, in Algorithms 1, 3, and 2.

The input of the problem consists of a weighted directed graph $G = (V, E)$, a source node s, and a weight function $w : E \to R$, that gives the weight of each edge.

The output is a boolean that shows whether there are no negative-weight cycles reachable from the source. In case of a true output, the shortest paths and the corresponding weights are stored in each node.

1.2 Implementation

Most of the project is written in C, with the exception of 2 Python files. These two are used for managing input test files. One of them is not used anymore, but it was designed for creating files that correspond to the required input format, that contain random input graphs with given input sizes. The other one is designed to convert the test files from the initial format, as taken from infoarena [5], to the one required by the implementation.

The structs necessary for storing the graphs are defined in the graph_structs header file. There are three structs, one for nodes, containing its own name, the name of the previous node on the shortest path, and the shortest distance to the source node, a struct for edges, containing a pointer to the source node, one for the destination node, and the weight expressed with an integer, and a struct for graphs, with a list of nodes, one of edges, and two integers representing the lengths of the two lists.

The graph_file_reader contains functions for reading a text file in the required format described before, interprets it, and returns a pointer to a graph object that corresponds to the one defined in the input file.

The OpenMP implementation resides in *bellman_ford.c*. Two regular *relax* and *initialize_single_source* functions are defined, with the omp parallel part appearing in the bellman_ford function. Most of the implementation follows the pseudocode described in the Appendix, with the exception of the parallel for pragma directive, which aims to divide the relax calls between the worker threads. The schedule is set to the best configuration obtained through tests, namely dynamic, with a chunk size of 1024.

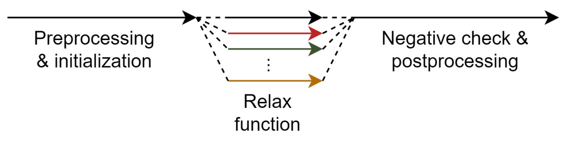

Figure 1: Serial and parallel parts of the entire program

The CUDA solution is written in the *cu_bellman_ford.cu* file. It works similarly to the CPU implementation, with the difference being that the parallel for loop is replaced with a number of calls to the relax function on device.

The utils and timing files contain utility functions for finding nodes in lists, for string interpretation, for file reading, and for timing the execution time.

The project also contains a main file, which is the entering point of the entire program, and which takes care of reading and interpreting the passed arguments. These decide between the CPU and GPU implementations, and give the paths to the directories containing the input and the expected output files. There is also the *tester.c* file, which iterates through the files in the two directories, calls the function described previously to create the graph, applies the selected implementation of Bellman-Ford, and verifies whether the obtained result is correct. Finally, there is also a *CMakeLists* file, because the project was originally intended to be built and run through CMake.

The required input file format is the following: on the first line, 2 numbers, n and m, representing the number of nodes and edges, respectively; on the second line, n strings of exactly 4 characters, each one separated by a blank space, representing the names of the nodes; on each of the following m lines, in this order, there are a string representing the name of the source node, a similar string for the destination node, and an integer, representing the edge weight. The names of the nodes can only contain small letters from the English alphabet. This should not be a limitation, since with 26 characters in this alphabet, there are $26^4 = 456,976$ possible names, with the largest input file only containing 50,000 nodes. It can be easily expanded by increasing the maximum number of nodes. This format for the name has been chosen for ease of implementation.

1.3 Parallel Programming

Regarding parallel programming, the Bellman-Ford algorithm has the advantage of its relaxation operations being independent to one another. Another plus is that the algorithm is insensitive to the order in which the edges are processed in a iteration. Thirdly, the fixed number of loops (n * m iterations, no matter what happened during the processing of an edge), makes it easy to divide the work in both, CPU and GPU, implementations.

Most of the parts outside the relax function can also be parallelized, representing one loop with n iterations, and one loop with m iterations. However, since they represent a small part of the entire execution, especially for large graphs, they were left to be executed in a serial manner. The obtained process can be graphically described like in Figure 1.

Inside the relax function, because of the condition, either 2 or 4 assignments are performed, together with the comparison. Some relaxation operations may be slower than others, but because of the randomness in the test graphs, one can assume that the true and false outcomes of the conditional predicate have a similar distribution.

1.3.1 Patterns

There exists a number of parallel programming patterns. Below, they are described, with explanations for whether or not they were used in this solution.

1. Embarrassingly Parallel - For this pattern, the task should require little to no communication. This is used because of the flexibility of the Bellman-Ford algorithm: in one iteration, the order in which the edges are processed is not important, so no synchronisation is needed between the threads. The only synchronisation happens after all edges have been processed exactly once, at which point the next iteration of edge processing can start.

2. Partition - This pattern uses disjoint regions of data, which can not be applied in our situation, since the threads work on the same nodes as edges.
3. Master-Worker - This pattern refers to the dispatching of jobs to worker-threads by a master-thread. This is applied in the current project, since there is a single master thread applying different techniques to divide and dispatch the RELAX functions to workers.
4. Stencil - This refers to updating an ordered structure of data based on a number of neighbours; this one can not be used because already updated nodes are accessed during a iteration.
5. Reduce - In the case of this pattern, an associative binary operation is applied to the elements of the array. This is not used, because the operations are done on individual elements of arrays, which do not interact among each other.
6. Scan - Similar to reduce, but the binary operation is recurrent. Not used, with a similar reasoning to the case of reduce.

1.3.2 OpenMP Implementation

Since the order of processing the edges is not important, a simple omp parallel for was suitable for this solution. More elaborate variants, with parallel sections or omp tasks, where the chunk of work is decided by the worker itself based on its id, or by the master, may give more flexibility and would work just as good. However, they would essentially move the work of developing the scheduler to the programmer, and the available ones are already performing well enough.

On the topic of schedulers, the evaluations further down below show 4 of them. The static one gives chunk-size tasks to workers in an ordered manner. The dynamic one is similar to the static one, with the difference that it dispatches the tasks to the workers that become available. Guided scheduling is similar to dynamic, with the difference being that the chunk size starts large, at the value of total work / number of workers, and gradually decreases to a minimum chunk size (in this project, this is left to its default value, which is 1). This should offer better work balancing, with little scheduling computational overhead. Auto lets the compiler choose the scheduling policy.

1.3.3 CUDA Implementation

As a difference from the OpenMP implementation, the CUDA one does not make use of the structs anymore. In initial trials, the memory allocation and host-to-device copying proved to be too difficult. Instead, the nodes are given consecutive positive integer ids, and all the data is represented as arrays (e.g. one array for the ids of the nodes that are sources of the corresponding edge, like sources[i] will be the id of the source node of the i^{th} edge, and so on). Since they are stored consecutively in memory, the arrays should also have better read-write times compared to the struct representation.

Although experiments have been done with different configurations as well, the initial intent was to keep the number of threads and blocks as close to each other as possible, in order to keep them at the lowest. This has been achieved by dynamically computing the number of threads and blocks for each test case. Since in each iteration m relax operations had to be performed, and a call to this function would yield $n_{threads} * n_{blocks}$ workers, the values for the two were approximated with addition to the square root of m. As it is shown later in the report, this method gave one of the best results.

Another consideration was the usage of streams, but since the largest portion of time is spent transferring the data from host to device, and the relaxation procedure happens rather close to the end of the n * m iterations (exclusively after (n-1) * m iterations), the streamlining of these 3 procedures would not bring too big of a benefit, so the implementation overhead was not justified.

2 Evaluation

For all evaluations, the algorithm was run 3 times, and the average is reported in the current document. For everything else, the configuration obtaining the shortest execution time on average is used. Since the test files have significantly different sizes and, in turn, different behaviours, the values are reported individually for each test.

The timing library and procedure used are similar to the ones from the class materials: time.h was deployed, only counting the time spent in the parallel execution part, unless otherwise specified (like the cudaMemalloc function time).

The hardware limitations must also be considered. For the OpenMP implementation, a maximum of 4 cores was allowed, and for CUDA, the server features can be found in the class materials.

2.1 Test Files

The set of files used for testing is taken from infoarena.ro [5], a public online forum dedicated to programming challenges. The test files are assumed to be correct since they come from a public domain, an active forum, so they went through public scrutiny and corrections if it was needed. They were uploaded in 2014. Only 18 of the 20 available files were used, because the last 2 would give segmentation faults on the SLURM server. These tests replace the initial randomly generated ones in order to have a balanced mix between negative cycle graphs and solvable ones.

2.2 Results

The numerical results are reported in 9 tables below, situated in the Appendix section.

2.2.1 OpenMP

For the OpenMP implementation, the following data is measured:

- In Table 1, there is the execution time, in seconds, of each of the 18 test cases, relative to the number of threads, from 1 to 6. The scheduler was set to dynamic, with a chunk size of 128.
- In Tables 2 and 3, the thread number is set to 4, and the time, measured in seconds, is reported for multiple types of scheduler (static, where a chunk-size of tasks is given in order to the threads, dynamic, similar to static, but with task dispatching to the free threads, auto, where the scheduler is decided by the compiler, and guided, similar to dynamic, but with a steadily decreasing chunk size). For the guided scheduler, the chunk size was left to the default value.
- The scaling efficiency in Table 4 is a number between 0 and 1 that represents, for each test file, how much it helps to use more threads. The scheduler is again set to dynamic, with 128 chunk size. It is computed using the following formula: $\frac{T_{serial}}{n_{threads} * T_{parallel}}$. The values are approximated to the third digit.
- Table 5 reports the estimated Amdahl's Law for each test file. In the exclusively serial part, there are two operations: the initialisation of the node information, which takes n iterations, and the final check for negative loops, which takes m iterations, while the parallel one will take n * m iterations, where n represents the number of nodes, and m represents the number of edges. So the α value will be $\frac{n+m}{n+m+n*m}$, with the final value computed as $\frac{1}{(1-\alpha)+\frac{\alpha}{n_{threads}}}$.
- In Table 6, the speedup is reported, which is computed as $\frac{T_{serial}}{T_{parallel}}$. The scheduler is set to dynamic, with a chunk size of 128.

2.2.2 CUDA

Conventional speedup could not be used, since there is no control over the number of CUDA cores used. Thus, the following measurements are reported:

- In Table 7, there is the execution time, in seconds, of each of the 18 test cases, relative to the number of blocks and threads. In the first part of the table, the threads and blocks are computed dynamically, such that they have values close to each other, so they can be approximated to the square root of the number of edges. For the second part, either the thread or the block number was fixed, let's say to a value x. In this situations, the value for the other division unit is set to $max(x, m/x)$, where m is the number of edges.

- Table 8 shows the throughput of the CUDA implementation, represented as the number of input size/second. Only the relaxation operations are counted, thus excluding the copying operations. The formula used is $\frac{n*m}{execution\ time}$, where n and m represent the number of nodes and edges, respectively. These values are used since the relaxation operations are applied on all data, in $n*m$ loops.

- In Table 9, the speedup relative to the OpenMP implementation is shown. The formula is T_{OpenMP}/T_{CUDA}. In the first part of the table, only the relaxation time is taken into account, while in the second part, this is summed with the cudaMemcpy operation time. The time is expressed in seconds.

2.3 Observations

2.3.1 OpenMP

To compute the execution time relative to the number of threads, the scheduler policy and chunk size were fixed to dynamic and 128, respectively. In this experiment, the hardware limitation of the SLURM machine to 4 cores is visible. Increasing the number of threads over this values leads to larger execution times, since they have to wait until a computational core becomes available. Also, for tests with small graphs, using more threads may actually cause increased execution time, because of the overhead created by the time required to dispatch the jobs. In ideal situations, doubling or tripling the number of threads, actually almost leads to half or a third of the time needed for the serial version, but better information on this topic can be seen in the speedup table.

To compute the execution time relative to the scheduler policy and the chunk size, the number of threads was fixed to the best value obtained in the previous experiment - 4. The best results are obtained for the guided policy and for the dynamic one, with large chunk sizes. For medium-sized graphs however, the policies give similar results, but also require medium-sized chunks. Really small chunk sizes are detrimental to large graphs, especially in the case of the dynamic policy, increasing the execution time close to ten-fold. This is a usual problem in parallel programming, where work partitioning has to find a trade off between load balancing and dispatching overhead.

Once these execution times are obtained, one can derive relative values. One of these is the scaling efficiency, which is a measure of how much an increased number of threads help

When talking about speedup, one must first consider Amdahl's law and observe the serial part of the entire program. In this case, since other operations are only performed once (1 n loop and 1 m loop), the Amdahl's law gives values close to ideal, especially in the large graph cases, where the maximum speedup is almost equal to the number of threads. For graphs with at most tens of nodes and edges however, the ideal speedup is not reached. And indeed, looking at the values in the next table, one can notice that larger graphs give higher speedup values. However, none of them reaches the maximum speedup, in part due to the scheduling overhead. There is also the expected decrease in speedup once the maximum number of cores is reached.

2.3.2 CUDA

Similarly to the CPU implementation, the time increases with the size of the graph, and on average the best times are obtained with the dynamic number of threads and blocks, and with the smaller values in the case of the fixed number of threads. This probably happens because the dispatching overhead to CUDA cores is smaller. Impressively, all configurations solve the problem below one second, even for the largest test graphs.

The throughput represents the number of processed data items per second, as a function of the input size. Here, the input size is considered as a product of nodes and edges, since this is also the number of loops in the parallel part. For this evaluation, the data copying time was not taken into consideration. The results show that the highest throughput is obtained with the largest graphs, probably due to initialization times and other overheads.

The final evaluation is a speedup relative to the OpenMP implementation. This is done in two parts, one where the host-to-device copying time was not taken into consideration, where CUDA, as expected, outperforms the CPU implementation in almost all tests, with values for the large graphs as high as 32. However, when taking into account the data transfer time as well, CUDA is on par or worse compared to the OpenMP implementation.

3 Future Improvements

One easy and quick improvement would be to change the data representation such that the list of edges is divided among the nodes in a way that each node contains a sub-list of edges that can change its d and π values. Such a data structure does not make sense from a logical point of view, but may yield even shorter execution times, eliminating the need of finding the node in the node list.

The biggest bottleneck for the CUDA implementation overall is the time required for the host-to-device memory copy operations, since this represents the majority of the overall exectuion time. On two different hardware configurations the problem persisted, so a more in-depth analysis of the existent solutions, such as using shared memory, would be needed.

Once the copying speeds are solved, the entire input array for each edge information could be divided into multiple subarrays and solved separately, such that even larger graphs can be solved by the current configuration.

For code clarity, one could experiment with a struct implementation for the CUDA solution as well, however the access times might be larger, since the data might not be allocated continously.

With this data implementation, streams maybe useful, to start the relaxing procedure immediately after the information for a node has been copied on the device, and start the device to host data transfer once the edges associated to a node are completely processed. However, this will not give significantly better results until the long times for data transfer between host and device are solved.

A Appendix - Pseudocode Snippets & Experimental Results Tables

Algorithm 1 The Bellman-Ford Algorithm [3]

```
procedure BELLMAN-FORD($G, w, s$)
    $INITIALIZE - SINGLE - SOURCE(G, s)$
    $i \leftarrow 1$
    while $i < |G.V|$ do
        $j \leftarrow 0$
        while $j < |G.E|$ do                          ▷ Embarrassingly parallel loop
            $RELAX(u[j], v[j], w)$        ▷ $u$ and $v$ are the sources and the destinations of the edges
            $j \leftarrow j + 1$
        end while
        $i \leftarrow i + 1$
    end while
    $j \leftarrow 0$
    while $j < |G.E|$ do
        if $v[j].d > u[j].d + w(u[j], v[j])$ then return $FALSE$
        end if
        $j \leftarrow j + 1$
    end while
     return $TRUE$
end procedure
```

Algorithm 2 The Relaxation Algorithm [3]

```
procedure RELAX($u, v, w$)
    if $v.d > u.d + w(u, v)$ then
        $v.d \leftarrow u.d + w(u, v)$
        $v.\pi \leftarrow u$
    end if
end procedure
```

Algorithm 3 The Source Initialization Algorithm [3]

```
procedure INITIALIZE-SINGLE-SOURCE($G, s$)
    $i \leftarrow 0$
    while $i < |G.V|$ do
        $G.V[i].d \leftarrow \infty$
        $G.V[i].\pi \leftarrow \infty$
        $i \leftarrow i + 1$
    end while
    $s.d = 0$
end procedure
```

Test \Threads	**1**	**2**	**3**	**4**	**5**	**6**
1	0,000260	0,000084	0,000126	0,000134	0,000437	0,000467
2	0,000007	0,000030	0,000033	0,000047	0,000199	0,000229
3	0,000013	0,000037	0,000051	0,000047	0,000505	0,000505
4	0,000009	0,000020	0,000029	0,000021	0,000440	0,000500
5	0,003170	0,002805	0,002665	0,002475	0,023111	0,027527
6	0,000072	0,000109	0,000126	0,000141	0,002542	0,002711
7	0,000874	0,001776	0,001875	0,001721	0,009874	0,012047
8	1,521083	0,907605	0,634349	0,602642	1,526764	1,542813
9	3,045707	1,614072	1,128418	1,047220	1,626111	1,647400
10	0,002316	0,003422	0,003663	0,003116	0,010489	0,012729
11	6,393992	3,360658	2,335696	2,162562	3,966346	4,066764
12	11,032112	6,086806	4,247558	3,945520	5,150379	5,065017
13	10,292236	5,560791	3,984163	3,766819	5,218245	4,981169
14	35,684196	20,244035	13,886690	12,445881	14,552443	14,263294
15	25,569988	14,791214	10,139964	9,440433	11,022979	11,036558
16	0,004699	0,007897	0,008115	0,006871	0,029119	0,032030
17	88,940964	47,850554	32,717475	29,990287	35,154667	34,369456
18	0,011126	0,015759	0,015194	0,012852	0,035188	0,034796

Table 1: Per-test execution time of the OpenMP implementation, relative to the number of threads. The time is measured in seconds. The configuration of the scheduler is dynamic, with 128 chunk size.

Test \Chunk Size	**1**	**16**	**128**	**1024**	**4096**
1	0,000177	0,000147	0,000309	0,000191	0,000209
2	0,000034	0,000038	0,000056	0,000051	0,000039
3	0,000047	0,000042	0,000067	0,000042	0,000094
4	0,000020	0,000032	0,000031	0,000046	0,000026
5	0,002699	0,002759	0,002712	0,005230	0,004437
6	0,000127	0,000114	0,000145	0,000138	0,000246
7	0,001725	0,001595	0,001522	0,001440	0,001366
8	0,810697	0,769560	0,540919	0,544372	0,560898
9	1,332957	1,437679	0,922087	0,871701	0,887928
10	0,003468	0,003016	0,002849	0,002908	0,002944
11	3,159823	2,743885	1,604942	1,696729	1,661447
12	5,268353	5,315321	3,377457	3,064915	3,253473
13	4,984376	4,928207	3,143944	2,904273	3,007580
14	17,034027	16,766125	10,715887	10,213434	10,158801
15	12,649028	12,593727	8,009969	7,416324	7,486856
16	0,006477	0,006615	0,006592	0,005704	0,005692
17	41,466160	38,988289	26,763226	25,954637	26,058287
18	0,012693	0,012607	0,012726	0,015259	0,011953

Table 2: Static scheduler per-test execution time of the OpenMP implementation, relative to the chunk size. The time is measured in seconds. Run on 4 threads.

Test \Scheduler	**1**	**16**	**128**	**1024**	**4096**	**Auto**	**Guided**
1	0,003595	0,000135	0,000134	0,000259	0,000219	0,000167	0,000148
2	0,000061	0,000036	0,000047	0,000067	0,000039	0,000070	0,000069
3	0,000053	0,000047	0,000047	0,000071	0,000088	0,000090	0,000077
4	0,000042	0,000398	0,000021	0,000020	0,000027	0,000027	0,000049
5	0,019164	0,002883	0,002475	0,005211	0,004571	0,002586	0,003414
6	0,027773	0,000201	0,000141	0,000152	0,000135	0,000122	0,000219
7	0,016466	0,001806	0,001721	0,001386	0,001283	0,001613	0,001909
8	10,873288	0,841847	0,602642	0,524159	0,593679	0,558361	0,516140
9	18,705564	1,453423	1,047220	0,797575	0,827256	0,902547	0,796387
10	0,009260	0,003356	0,003116	0,003102	0,002923	0,003116	0,003461
11	35,654839	2,828068	2,162562	1,662321	1,799326	1,543334	1,680869
12	69,236257	5,644782	3,945520	2,824622	2,940355	3,431421	2,836865
13	62,648732	5,096519	3,766819	2,684604	2,835690	2,842364	2,703567
14	197,250504	17,830064	12,445881	8,999015	8,792835	9,814333	8,989610
15	147,262330	12,296913	9,440433	6,616773	6,827373	7,203889	6,663911
16	0,020669	0,007494	0,006871	0,006164	0,006093	0,005969	0,006778
17	401,437699	37,856434	29,990287	22,585737	23,061395	25,254513	22,516113
18	0,045265	0,013000	0,012852	0,015780	0,013100	0,012054	0,012367

Table 3: Dynamic scheduler per-test execution time of the OpenMP implementation, relative to the chunk size. The time is measured in seconds. Run on 4 threads. The scheduler is represented either by chunk size, in the case of the dynamic scheduler, or by the type of scheduler for guided and auto, with the chunk size left to default in these 2 cases.

Test \Threads	**1**	**2**	**3**	**4**	**5**	**6**
1	1,000	1,548	0,687	0,486	0,119	0,093
2	1,000	0,125	0,074	0,040	0,007	0,005
3	1,000	0,174	0,085	0,069	0,005	0,004
4	1,000	0,221	0,102	0,109	0,004	0,003
5	1,000	0,565	0,397	0,320	0,027	0,019
6	1,000	0,330	0,190	0,128	0,006	0,004
7	1,000	0,246	0,155	0,127	0,018	0,012
8	1,000	0,838	0,799	0,631	0,199	0,164
9	1,000	0,943	0,900	0,727	0,375	0,308
10	1,000	0,338	0,211	0,186	0,044	0,030
11	1,000	0,951	0,913	0,739	0,322	0,262
12	1,000	0,906	0,866	0,699	0,428	0,363
13	1,000	0,925	0,861	0,683	0,394	0,344
14	1,000	0,881	0,857	0,717	0,490	0,417
15	1,000	0,864	0,841	0,677	0,464	0,386
16	1,000	0,297	0,193	0,171	0,032	0,024
17	1,000	0,929	0,906	0,741	0,506	0,431
18	1,000	0,353	0,244	0,216	0,063	0,053

Table 4: Per-test scaling efficiency of the OpenMP implementation, relative to the number of threads. The configuration of the scheduler is dynamic, with 128 chunk size.

Test	**n**	**m**	**alpha**	**1**	**2**	**3**	**4**	**5**	**6**
1	5	10	0,23077	1,00	1,63	2,05	2,36	2,60	2,79
2	5	20	0,20000	1,00	1,67	2,14	2,50	2,78	3,00
3	10	20	0,13043	1,00	1,77	2,38	2,88	3,29	3,63
4	10	50	0,10714	1,00	1,81	2,47	3,03	3,50	3,91
5	500	1000	0,00299	1,00	1,99	2,98	3,96	4,94	5,91
6	500	100	0,01186	1,00	1,98	2,93	3,86	4,77	5,66
7	200	500	0,00695	1,00	1,99	2,96	3,92	4,86	5,80
8	17000	17015	0,00012	1,00	2,00	3,00	4,00	5,00	6,00
9	10000	50000	0,00012	1,00	2,00	3,00	4,00	5,00	6,00
10	200	1000	0,00596	1,00	1,99	2,96	3,93	4,88	5,83
11	30000	30010	0,00007	1,00	2,00	3,00	4,00	5,00	6,00
12	15000	120000	0,00007	1,00	2,00	3,00	4,00	5,00	6,00
13	20000	80000	0,00006	1,00	2,00	3,00	4,00	5,00	6,00
14	25000	200000	0,00004	1,00	2,00	3,00	4,00	5,00	6,00
15	25000	150000	0,00005	1,00	2,00	3,00	4,00	5,00	6,00
16	500	1000	0,00299	1,00	1,99	2,98	3,96	4,94	5,91
17	50000	200000	0,00002	1,00	2,00	3,00	4,00	5,00	6,00
18	500	2000	0,00249	1,00	2,00	2,99	3,97	4,95	5,93

Table 5: Per-test estimated Amdahl's law. n and m represent the number of nodes and edges, respectively.

Test \Threads	**1**	**2**	**3**	**4**	**5**	**6**
1	1,00	1,10	2,06	1,95	0,59	0,56
2	1,00	0,25	0,22	0,16	0,04	0,03
3	1,00	0,35	0,25	0,28	0,03	0,03
4	1,00	0,44	0,31	0,44	0,02	0,02
5	1,00	1,13	1,19	1,28	0,14	0,12
6	1,00	0,66	0,57	0,51	0,03	0,03
7	1,00	0,49	0,47	0,51	0,09	0,07
8	1,00	1,68	2,40	2,52	1,00	0,99
9	1,00	1,89	2,70	2,91	1,87	1,85
10	1,00	0,68	0,63	0,74	0,22	0,18
11	1,00	1,90	2,74	2,96	1,61	1,57
12	1,00	1,81	2,60	2,80	2,14	2,18
13	1,00	1,85	2,58	2,73	1,97	2,07
14	1,00	1,76	2,57	2,87	2,45	2,50
15	1,00	1,73	2,52	2,71	2,32	2,32
16	1,00	0,59	0,58	0,68	0,16	0,15
17	1,00	1,86	2,72	2,97	2,53	2,59
18	1,00	0,71	0,73	0,87	0,32	0,32

Table 6: Per-test speedup of the OpenMP implementation, relative to the number of threads. The configuration of the scheduler is dynamic, with 128 chunk size.

Test	Threads	Blocks	Time	Fixed 16t	Fixed 128t	Fixed 1024t	Fixed 1024b
1	3	4	0,00004	0,00004	0,00004	0,00005	0,00005
2	4	6	0,00004	0,00003	0,00003	0,00005	0,00005
3	4	6	0,00007	0,00007	0,00007	0,00009	0,00009
4	7	8	0,00006	0,00006	0,00006	0,00008	0,00009
5	31	33	0,00293	0,00286	0,00286	0,00443	0,00447
6	10	11	0,00029	0,00029	0,00029	0,00044	0,00045
7	22	23	0,00115	0,00115	0,00117	0,00178	0,00178
8	130	131	0,11360	0,11328	0,11341	0,17711	0,17696
9	223	225	0,08651	0,08714	0,08685	0,11917	0,11942
10	31	33	0,00124	0,00120	0,00123	0,00177	0,00179
11	173	174	0,21674	0,21586	0,21620	0,30855	0,31370
12	346	347	0,17197	0,17243	0,17162	0,21508	0,21558
13	282	284	0,19436	0,19268	0,19366	0,25769	0,25476
14	447	448	0,36491	0,35042	0,35249	0,41916	0,42874
15	387	388	0,31561	0,30889	0,31390	0,38024	0,38189
16	31	33	0,00313	0,00321	0,00320	0,00410	0,00457
17	447	448	0,69568	0,67544	0,67481	0,81283	0,80581
18	44	46	0,00306	0,00294	0,00294	0,00409	0,00410

Table 7: Per-test execution time of the CUDA implementation, relative to the number of blocks and threads. The time is measured in seconds. The first three columns of values represent the per-test number of threads and blocks, as it is calculated dynamically, based on the size of the input test.

Test	n x m	Time	Throughput
1	50	0,00004	1.162.791
2	100	0,00004	2.857.143
3	200	0,00007	3.000.000
4	500	0,00006	7.853.403
5	500000	0,00293	170.726.155
6	50000	0,00029	175.438.596
7	100000	0,00115	87.057.458
8	289255000	0,11360	2.546.326.047
9	500000000	0,08651	5.779.500.497
10	200000	0,00124	161.725.067
11	900300000	0,21674	4.153.869.578
12	1800000000	0,17197	10.466.840.468
13	1600000000	0,19436	8.232.061.823
14	5000000000	0,36491	13.702.083.813
15	3750000000	0,31561	11.881.627.308
16	500000	0,00313	159.948.816
17	10000000000	0,69568	14.374.411.248
18	1000000	0,00306	327.082.425

Table 8: The throughput of the CUDA implementation. The configuration is set to the dynamic computation of the number of blocks and threads. The time is measured in seconds. n and m represent the number of nodes and edges, respectively.

Test	**OpenMP Time**	**CUDA Time**	**Speedup**	**cudaMemcopy Time**	**”True” Speedup**
1	0,00026	0,00004	6,02	0,098	0,003
2	0,00007	0,00004	1,91	0,001	0,044
3	0,00007	0,00007	1,07	0,001	0,056
4	0,00002	0,00006	0,32	0,001	0,020
5	0,00521	0,00293	1,78	0,003	0,906
6	0,00015	0,00029	0,53	0,001	0,114
7	0,00139	0,00115	1,21	0,001	0,552
8	0,52416	0,11360	4,61	1,027	0,459
9	0,79758	0,08651	9,22	1,782	0,427
10	0,00310	0,00124	2,51	0,002	1,022
11	1,66232	0,21674	7,67	3,174	0,490
12	2,82462	0,17197	16,42	6,419	0,429
13	2,68460	0,19436	13,81	5,587	0,464
14	8,99902	0,36491	24,66	18,444	0,478
15	6,61677	0,31561	20,96	13,045	0,495
16	0,00616	0,00313	1,97	0,003	1,015
17	22,58574	0,69568	32,47	37,978	0,584
18	0,01578	0,00306	5,16	0,005	1,967

Table 9: The speedup of the CUDA implementation, relative to the OpenMP one, with and without taking into consideration the memory copy time. The time is measured in seconds.

References

[1] R. Bellman. On a routing problem. *Quarterly of applied mathematics*, 16(1):87–90, 1958.

[2] R. Chandra, L. Dagum, D. Kohr, R. Menon, D. Maydan, and J. McDonald. *Parallel programming in OpenMP*. Morgan kaufmann, 2001.

[3] T. Cormen, C. Leiserson, R. Rivest, and C. Stein. *Introduction to Algorithms, fourth edition*. MIT Press, 2022.

[4] L. R. Ford. Network flow theory. 1956.

[5] V. Gavrila. infoarena.ro - algoritmul bellman-ford, 2014.

[6] E. F. Moore. The shortest path through a maze. In *Proc. of the International Symposium on the Theory of Switching*, pages 285–292. Harvard University Press, 1959.

[7] NVIDIA, P. Vingelmann, and F. H. Fitzek. Cuda, release: 10.2.89, 2020.