Architectures and Platforms for Artificial Intelligence

Parallel Implementation of the Bellman-Ford Algorithm

Pelinsu Acar
30.09.2024

1. Introduction

The Bellman-Ford algorithm [1] is a fundamental algorithm in graph theory for finding the shortest paths from a single source vertex to all other vertices in a weighted graph, even when the graph contains negative weight edges. Unlike Dijkstra's algorithm, Bellman-Ford can handle negative-weight edges, but it runs slower with a time complexity of O(V×E), where V is the number of vertices and E is the number of edges. In this project, the Bellman-Ford algorithm is parallelized using two approaches:

1. **OpenMP-based parallelization** to run the algorithm on CPU cores. [2]
2. **CUDA-based parallelization** to leverage the computational power of NVIDIA GPUs. [3]

This document describes the implementation details of the two versions of Belmann Ford algorithm and then continues with the performance evaluation results, and concludes with some possible future improvements. All the tables including experiment results can be found in the Appendix.

2. The Bellman-Ford Algorithm

2.1. Algorithm Overview [1]

The Bellman-Ford algorithm works by iteratively relaxing all edges in the graph. In each iteration, the algorithm examines each edge to determine if a shorter path to a destination vertex can be found through a given edge. The input of the algorithm takes a directed graph which can be described as G(V, E) where each edge is represented by its source node, destination node, and weight. The algorithm performs V−1 iterations, where V is the number of vertices, and at each iteration, all the edges are relaxed. After V−1 iterations, any further relaxation indicates the presence of a negative-weight cycle. The algorithm will output a message if a negative-weight cycle is detected. In case of no negative cycle, a distance array of size V (number of vertices), where each element dist[i] contains the shortest distance from the source node s to node i is stored. If a node is unreachable from the source, the corresponding distance will remain infinity (INF).

2.2. Parallelization Opportunities

The Bellman-Ford algorithm is inherently sequential due to its iterative nature. However, the relaxation step for each edge is independent, making it suitable for parallelization across multiple threads or GPU threads. Each iteration of the edge-relaxation loop can be distributed across different processing units. The final step to check for negative weight cycles is kept serial. Although it is possible to parallelize that part as well, it's only a verification step after the main computation and constructs small part of the execution, so it would not change the results drastically.

2.3 Parallel Programming Paradigms

Embarrassingly Parallel: In each iteration, multiple threads process different edges in parallel. Each thread independently checks the relaxation condition for a given edge and updates the destination node's distance if a shorter path is found. Since no communication is needed between the threads while processing individual edges, this part of the algorithm is embarrassingly parallel. The only communication happens at the end of each iteration, where the flag ***d_has_next*** is checked to determine if another iteration is needed.

Partition: The parallelization of Bellman-Ford here is based on partitioning the set of edges. Each thread handles a subset of edges. The graph's edges are implicitly partitioned across the threads by dividing the work using ***blockDim.x * blockIdx.x + threadIdx.x***. This is a Partition-based parallelization, as each thread works on a different part of the problem space (an edge in the graph).

Master-Worker: There is no explicitly a Master-Worker paradigm. The CPU does act as a "master" by invoking the CUDA kernel, but once the kernel is launched, it is fully handled on the GPU. There's no continual feedback or assignment of new work between the CPU and GPU in the middle of kernel execution.

Stencil: There is no stencil pattern since in stencil computations, there is typically a grid, and each element is updated based on its neighboring elements. Bellman-Ford does not rely on this type of spatial locality.

Reduce: The Bellman-Ford algorithm does not perform a traditional Reduce operation. However, the algorithm is indirectly reducing information since the goal is to minimize distances, but this is not achieved using reduction operations like summation or min/max across the entire data structure in a single step.

Scan: There's no scan operation (like prefix sum) because of the same reasoning in reduce.

3.0 Implementation Details

The project is written in C and it contains 2 implementation files called **cuda_bellman_ford.cu** and **openmp_bellman_ford.c**, and 2 bash scripts called **run_cuda_bellman_ford.sh** and **run_openmp_bellman_ford.sh** These bash scripts are used to automate the testing of a CUDA-based and openMP-based implementations of the Bellman-Ford algorithm. It executes the algorithm on multiple input datasets and compares the results to expected outputs. The script loops over a range of input files (from input1.txt to input10.txt) inside the **bellman_ford_data** folder and then compare the correct outputs again inside that folder with the ones generated by cuda or openmp implementations stored in **cuda_bellman_ford_results** and **openmp_bellman_ford_results** respectively. The cuda script takes one argument namely **NUMTHREADS** which specifies the number of threads to be used by the CUDA program. This value can be provided as an argument when running the script. If not provided, it defaults to 256 threads. The openmp script instead requires 3 arguments to be passed when executing:

- **NUMTHREADS:** The number of threads to be used by OpenMP.
- **SCHEDULE_TYPE:** The type of scheduling to be used in OpenMP (e.g., static, dynamic, guided).
- **CHUNK_SIZE:** The size of chunks used by OpenMP for scheduling when distributing work across threads.

Expected input format: The first line of the file specifies two integers:

V: The number of vertices (nodes) in the graph.
E: The number of edges in the graph.

Each subsequent line describes one edge in the graph. It consists of three values:

u: The starting vertex of the edge.
v: The ending vertex of the edge.
w: The weight of the edge between vertex u and vertex v.

Expected output format: Each output file (e.g., output1.txt, output2.txt, etc.) typically contains the shortest path distances from the source vertex to all other vertices, following a similar structure: Each line contains the shortest distance from the source vertex to vertex i. If a negative cycle is detected, a string "FOUND NEGATIVE CYCLE!" is printed.

3.1 CUDA Implemantation

The parallelization of the Bellman-Ford algorithm in CUDA exploits the highly parallel architecture of GPUs. In each iteration, the relaxation of each edge is assigned to an independent CUDA thread. The distance array is initialized on the host (CPU), where all vertices are assigned an infinite distance, except for the source vertex, which is assigned a distance of zero. The CUDA implementations uses two structs: node and edge. Additionally, it defines a third struct, graph, which contains pointers to arrays of node and edge structs. Overall, it contains the number of nodes (n), the number of edges (m), and arrays of nodes and edges. Node struct, represents a single node in the graph, an int id which stores the node's ID. Edge struct represents an edge between two nodes in the graph with source node ID, the destination node ID and the weight of the edge. Arrays of nodes and edges are dynamically allocated on the host using ***malloc()***, and graph distances are initialized with ***calloc()*** to handle initialization to zero. GPU memory is allocated for the edges, distance array ***(d_dist)***, and a boolean flag ***(d_has_next)*** to signal whether further iterations are required. These allocations are made using ***cudaMalloc()***. Data is transferred from the host to the device using ***cudaMemcpy()***. This includes copying the edge list and initial distances. After each iteration, updates to the ***d_has_next*** flag and the distance array are transferred back to the host to check for convergence.

The CUDA kernel ***bellman_ford_one_iter()*** is responsible for performing one iteration of edge relaxation in parallel. Each thread processes one edge, and if a shorter path to the destination node is found, the distance is updated. The number of threads is specified by the ***threadsPerBlock*** parameter, which is defined dynamically based on the input passed from the command line. The number of threads per block should typically be a multiple of the GPU's warp size. Threfore, it was experimented with 16, 64, 128, 256, and 512 threads. The number of blocks in the grid ***(blocksPerGrid)*** is calculated based on the total number of edges and threads per block by using the formula:

$$\boxed{\textit{blocksPerGrid} = (m + \textit{threadsPerBlock} - 1) / \textit{threadsPerBlock}}$$

Fig. 1: Dynamic calculation formula of "blocks per grid"

This ensures that the number of blocks is sufficient to cover all edges, even if m is not a perfect multiple of threadsPerBlock. After the n-1 iterations, the algorithm checks for the presence of negative cycles by verifying if any edge can still be relaxed. This is performed on the CPU by iterating through the host's copy of the distance array and checking if any distance can be reduced further.

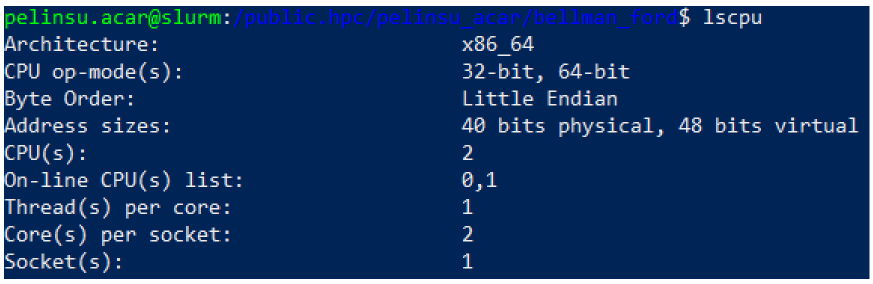
3.2 OpenMP Implementation

The OpenMP implementation follows a structure similar to the CUDA implementation, where the graph is represented by an edge list, and the Bellman-Ford algorithm is run for n-1 iterations (where n is the number of nodes). The key difference is the usage of CPU-based parallelism with OpenMP. Similar to the serial and CUDA versions, the distances to all nodes are initialized to infinity, except for the source node, which is set to 0. A secondary distance array ***(local_dist)*** is used to store updates before synchronizing them back to the primary distance array ***(dist).*** After each iteration, the results from the ***local_dist*** array are copied back to the ***dist*** array using another parallel loop. This synchronization ensures that all threads are working with the most up-to-date distance values in the next iteration. However, there is a trade-off between synchronization overhead and work distribution efficiency in the final step where the ***local_dist*** array is copied back to the ***dist*** array. This could potentially become a bottleneck, especially for very large graphs. The code uses a ***#pragma omp parallel*** for directive to distribute the workload across the available threads. The OpenMP schedule is dynamically determined based on user input, allowing the code to switch between different scheduling strategies (static, dynamic, and auto).

- **Static**: Work is divided into equal-sized chunks, and each thread is assigned a chunk of work to perform. This approach works well when the workload is evenly distributed across the edges.
- **Dynamic**: Threads are assigned chunks of work dynamically as they finish their current assignments. This approach is beneficial when the workload is unbalanced or unpredictable.
- **Auto**: The scheduling strategy is automatically determined by the OpenMP runtime based on the system configuration and workload characteristics.

4.0 Evaluation

To get a better overview, the algorithm was run 5 times for all evaluations and the average is reported in the tables. The time measurement in CUDA implementation is done using the ***gettimeofday()*** function, which provides the current time in seconds and microseconds. This function is used to measure three key segments of the program: memory allocation/setup, memory transfers (Host to Device and Device to Host), and the execution time of the CUDA kernel.
The OpenMP implementation uses OpenMP's built-in timing function ***omp_get_wtime()*** to measure the execution time of the Bellman-Ford algorithm. For the OpenMP implementation, the hardware constraints was also considered. As it can be seen from figure 2, a maximum of 2 cores is allowed, and for CUDA, the server features is available in the class materials.



```
pelinsu.acar@slurm:/public.hpc/pelinsu_acar/bellman_ford$ lscpu
Architecture:                        x86_64
CPU op-mode(s):                      32-bit, 64-bit
Byte Order:                          Little Endian
Address sizes:                       40 bits physical, 48 bits virtual
CPU(s):                              2
On-line CPU(s) list:                 0,1
Thread(s) per core:                  1
Core(s) per socket:                  2
Socket(s):                           1
```

Fig. 2: Detailed information about the CPU architecture and configuration of the system

Test Set: There are 10 graphs in the test set which are all obtained from Infoarena.ro [4] which is a community about IT and programming. Since the correct output files are also available, these outputs are used to test the correctness of the implemented algorithm before the performance analysis. Out of 10 test files, 3 of them contain negative cycle.

4.1. OpenMP Performance Considerations

Table 1: Execution Time Relative to Number of Threads

This table presents execution time with varying thread counts. Since the system only has 2 cores, the comparison between 1 and 2 threads is most relevant.

For small inputs (rows 1 and 2), there is minimal benefit from using 2 threads over 1. This indicates that for extremely small inputs, the overhead associated with creating and managing additional threads outweighs any speedup, especially given that there is very little computational workload to distribute.

For medium-sized inputs (rows 3, 4, and 6), using 2 threads shows noticeable improvements in execution time compared to using 1 thread, showing the advantage of parallelism when the input size becomes large enough to distribute across the available 2 cores. Since large input sizes provide sufficient workload to be evenly divided across both cores, the use of 2 threads is efficient and achieves notable improvements.

For larger numbers of threads (4 and 6), the execution times generally increase due to oversubscription. This highlights the inefficiency of using more threads than available physical cores, as the operating system needs to switch between threads, which introduces overhead.

Table 2: Scheduler (Static vs Dynamic) Impact on Execution Time

In this table, all experiments were done using **2 threads**, which is optimal for our system. It compares the performance of different scheduling methods, as well as different chunk sizes.

For smaller input sizes, static scheduling generally outperforms dynamic scheduling. As the input size increases, dynamic scheduling starts to perform better in some cases, especially for larger and more complex workloads.

Small chunk sizes tend to increase execution time for both static and dynamic scheduling, especially for larger inputs, because they result in more frequent synchronization and scheduling overhead. Larger chunk sizes (128 and above) typically improve performance. The optimal chunk size for both schedulers seems to be around 128 to 1024, particularly for large input sizes. The auto scheduler generally performs well across input sizes. It strikes a balance between dynamic and static chunk sizes, automatically adjusting based on the workload.

Table 3: Scaling Efficiency

This table shows the scaling efficiency according to number of threads. The formula of scaling efficiency:

$$\frac{Tserial}{numThreads * Tparalel}$$

where $Tserial = Tparalel(p = 1)$.

As it is discussed earlier, the scaling efficiency is quite low when increasing the number of threads for small graphs due to overhead costs. For medium/large input sizes, scaling efficiency improves significantly. While it’s not a perfect linear speedup (which would yield a scaling efficiency of 1), it shows that parallelism is beneficial for large input sizes on the 2-core system, resulting in a near-optimal utilization of the available hardware. When number of threads exceeds 2, the scaling efficiency decreases significantly because of the oversubscription.

In summary, the OpenMP implementation performs optimally with 2 threads for medium to large input sizes, using static scheduling with larger chunk sizes, or the auto scheduler for dynamic workloads.

4.2. CUDA Performance Considerations

Table 4: Execution Time with Varying Thread Counts

This table shows the execution time of CUDA implementation with different number of threats (16, 64, 128, 256, 512). The number of blocks is calculated dynamically as it can be seen in figure 1.

The CUDA implementation demonstrates significantly lower execution times compared to the OpenMP implementation, particularly for larger inputs. This suggests that CUDA can exploit the parallel architecture of GPUs more effectively than OpenMP on CPUs, resulting in better performance.

The execution time remains consistent across varying block and thread counts, indicating that the dynamic block size calculation effectively adapts to the input size.

Table 5: Performance Comparison Between OpenMP and CUDA

The speedup achieved by the CUDA implementation over OpenMP is substantial, particularly for larger inputs. The execution time speedup is impressive, with values reaching up to 264.53 for the largest input.

The true speedup, factoring in the CUDA memory copy time, still demonstrates substantial performance benefits for CUDA, especially in larger datasets. For instance, even with memory copy time included, the CUDA implementation significantly outperforms OpenMP for large graphs.

5.0 Conclusion

The OpenMP implementation demonstrates optimal performance for small input sizes, where the overhead of thread management is minimal. For small inputs, OpenMP outperforms CUDA, as the computational workload is not substantial enough to leverage the GPU's parallel processing capabilities effectively. However, as input sizes increase to medium and large datasets, OpenMP's performance benefits from parallelism become evident. Static scheduling tends to perform better for smaller datasets, while dynamic scheduling becomes advantageous for larger inputs, particularly with larger chunk sizes. While the CUDA implementation shows limitations for small input sizes compared to OpenMP, it excels significantly with medium and large datasets. The ability to exploit the parallel architecture of GPUs results in lower execution times for larger inputs. The execution time remains consistent across varying block and thread configurations, indicating effective adaptation to input size. The speedup achieved by CUDA over OpenMP for larger datasets is substantial, with execution time speedups reaching up to 264.53. Even when factoring in CUDA memory copy times, the performance benefits of GPU acceleration are clear for large graphs.

In conclusion, the evaluation indicates that OpenMP is more suitable for small input sizes, where it outperforms CUDA due to lower overhead. However, for medium to large input sizes, CUDA demonstrates its superiority. Future optimizations could focus on maximizing memory transfer efficiency in CUDA using shared memory within each block to store the distances. This can reduce global memory access times, as threads can quickly access shared memory, which is significantly faster than global memory.

APPENDIX

```
Algorithm 1: Bellman Ford Algorithm
  $\forall v \in V, d[v] \leftarrow \infty$ // set initial distance estimates
  //optional: set $\pi(v)$ ←nil for all $v$, $\pi(v)$ represents the predecessor of $v$
  $d[s] \leftarrow 0$ // set distance to start node trivially as 0
  for $i$ from $1 \rightarrow n-1$ do
      for $(u, v) \in E$ do
          $d[v] \leftarrow \min\{d[v], d[u] + w(u, v)\}$ // update estimate of $v$
          // optional - if $d[v]$ changes, then $\pi(v) \leftarrow u$
  // Negative Cycle Step
  for $(u, v) \in E$ do
      if $d[v] > d[u] + w(u, v)$ then
          return "Negative Cycle"; // negative cycle detected
  return $d[v]\ \forall\ v \in V$
```

	Execution Time			
Input \# of Threads	1	2	4	6
1 (n=5, m=10)	0.000012	0.000165	0.000494	0.000313
2 (n=10, m=50)	0.000014	0.000144	0.000364	0.001114
3 (n=500, m=1000)	0.002044	0.003690	0.055836	0.052027
4 (n=200, m=500)	0.000576	0.000798	0.023016	0.027297
5 (n=17000, m=17015)	1.272155	0.885843	2.424891	2.891903
6 (n=200, m=1000)	0.000896	0.001210	0.005406	0.008672
7 (n=30000, m=30010)	3.849403	2.340084	5.646446	6.631626
8 (n=15000, m=120000)	5.973851	3.249481	4.795110	5.037893
9 (n=25000, m=200000)	17.389421	9.445980	12.607947	12.123925
10 (n=50000, m=200000)	43.524863	25.717356	30.898470	30.902315

Table 1: Execution time of the OpenMP implementation for each test graph, relative to the number of threads in seconds. The configuration of the scheduler is static, with 128 chunk size.

	Execution Time										
Input \Chunk Size	1		16		128		512		1024		Auto
	Static	Dynamic	Static	Dynamic	Static	Dynamic	Static	Dynamic	Static	Dynamic	
1 (n=5, m=10)	0.000052	0.000982	0.000064	0.0020006	0.000084	0.000266	0.000146	0.000061	0.000070	0.000051	0.000139
2 (n=10, m=50)	0.000765	0.001242	0.000074	0.000607	0.000087	0.000067	0.000089	0.000109	0.000237	0.000062	0.003845
3 (n=500, m=1000)	0.006040	0.011892	0.004327	0.004636	0.004162	0.001906	0.002515	0.002751	0.001470	0.001468	0.002598
4 (n=200, m=500)	0.003375	0.003139	0.000875	0.000931	0.000840	0.002560	0.001367	0.000744	0.001436	0.000781	0.000833
5 (n=17000, m=17015)	1.340366	5.807491	0.962823	2.152235	0.903297	1.055168	0.892647	0.910667	0.865631	0.890893	0.623568
6 (n=200, m=1000)	0.012596	0.027166	0.001399	0.001507	0.001272	0.001286	0.008435	0.001320	0.002786	0.001112	0.001296
7 (n=30000, m=30010)	4.397671	19.277610	2.409743	5.334456	2.295089	2.565974	2.143023	2.618856	2.263112	2.132328	2.193406
8 (n=15000, m=120000)	7.049379	45.478676	3.575563	12.406912	3.353663	5.370519	3.208878	3.706116	3.420507	3.370491	3.213831
9 (n=25000, m=200000)	21.496356	102.495655	10.879765	32.600702	10.279448	11.167446	8.927480	11.261779	8.857156	8.909002	8.935690
10 (n=50000, m=200000)	59.284933	299.357448	34.589155	63.684073	25.229068	29.009865	23.468144	25.308524	22.887598	23.266495	23.290774

Table 2: Dynamic/Static/Auto scheduler execution time of the OpenMP implementation for each test graph, relative to the chunk size in seconds. Run on 2 threads.

	Scaling Efficiency			
Input \# of Threads	1	2	4	6
1 (n=5, m=10)	1	0.036	0.006	0.006
2 (n=10, m=50)	1	0.049	0.010	0.002
3 (n=500, m=1000)	1	0.277	0.009	0.007
4 (n=200, m=500)	1	0.361	0.006	0.004
5 (n=17000, m=17015)	1	0.718	0.131	0.073
6 (n=200, m=1000)	1	0.370	0.041	0.017
7 (n=30000, m=30010)	1	0.822	0.170	0.096
8 (n=15000, m=120000)	1	0.842	0.311	0.198
9 (n=25000, m=200000)	1	0.865	0.345	0.239
10 (n=50000, m=200000)	1	0.785	0.352	0.235

Table 3: Scaling efficiency of the OpenMP implementation, relative to the number of threads for each test graph. The configuration of the scheduler is static, with 128 chunk size.

	Execution Time				
Input \# of Threads	16	64	128	256	512
1 (n=5, m=10)	0.000021	0.000033	0.000027	0.000027	0.000028
2 (n=10, m=50)	0.000037	0.000042	0.000043	0.000043	0.000041
3 (n=500, m=1000)	0.001146	0.001109	0.001293	0.001275	0.001335
4 (n=200, m=500)	0.000530	0.000469	0.000512	0.000557	0.000536
5 (n=17000, m=17015)	0.041492	0.040844	0.039122	0.037955	0.040779
6 (n=200, m=1000)	0.000448	0.000449	0.000488	0.000515	0.000524
7 (n=30000, m=30010)	0.070813	0.069961	0.068569	0.067381	0.070765
8 (n=15000, m=120000)	0.037734	0.034936	0.034423	0.036264	0.036255
9 (n=25000, m=200000)	0.057697	0.056788	0.054935	0.051560	0.049853
10 (n=50000, m=200000)	0.102405	0.100719	0.097221	0.101767	0.103064

Table 4: Execution time of the CUDA implementation, relative to the number of blocks and threads for each test graph in seconds. Block size is calculated dynamically, based on the size of input test and number of threat.

Input	OpenMP Algorithm Execution Time	CUDA Algorithm Execution Time	CUDA Memory Copy Time	Execution Time Speedup	True Speedup
1 (n=5, m=10)	0.000165	0.000027	0.000069	6.11	1.72
2 (n=10, m=50)	0.000144	0.000043	0.000105	3.35	0.97
3 (n=500, m=1000)	0.003690	0.001293	0.003171	2.85	0.83
4 (n=200, m=500)	0.000798	0.000512	0.001386	1.56	0.42
5 (n=17000, m=17015)	0.885843	0.039122	0.105272	22.64	6.14
6 (n=200, m=1000)	0.001210	0.000488	0.001308	2.48	0.67
7 (n=30000, m=30010)	2.340084	0.068569	0.185408	34.13	9.21
8 (n=15000, m=120000)	3.249481	0.034423	0.098408	94.40	24.46
9 (n=25000, m=200000)	9.445980	0.054935	0.160299	171.9483	43.90
10 (n=50000, m=200000)	25.717356	0.097221	0.304573	264.53	64.00

Table 5: The speedup of the CUDA implementation, relative to the OpenMP, with and without considering the memory copy time in seconds.

REFERENCES

1. Bellman, R. E. (1958). On a routing problem. *Quarterly of Applied Mathematics, 16*(1), 87-90. https://doi.org/10.1090/qam/99892
2. Chandra, R., Dagum, L., Kohr, D., Menon, R., Maydan, D., & McDonald, J. (2001). *Parallel programming in OpenMP*. Morgan kaufmann.
3. NVIDIA, Vingelmann, P., & Fitzek, F. H. P. (2020). *CUDA, release: 10.2.89*. Retrieved from https://developer.nvidia.com/cuda-toolkit
4. InfoArena. (n.d.). Bellman-Ford algorithm. Retrieved September 30, 2024, from https://www.infoarena.ro/problema/bellmanford