

# DD2360HT25 Project Report

Alexis Bon<sup>1</sup>, Borna Bilas<sup>2</sup>, and Bouziane Abderrahmane<sup>3</sup>

<sup>1</sup>KTH, *alexisbo@kth.se*

<sup>2</sup>KTH, *bornab@kth.se*

<sup>3</sup>KTH, *bouziane@kth.se*

2026-02-02

## 1 Contributions

This section summarizes the work performed by each group member.

### 1.1 Alexis Bon

- CPU solver
- Hybrid solver

### 1.2 Borna Bilas

- Performance measurements
- Report write-up

### 1.3 Bouziane Abderrahmane

- Software architecture
- GPU solver

## 2 Introduction

We address the *Single Source Shortest Path (SSSP)* problem: computing the shortest path distances from a single source node to all other nodes in a graph. SSSP is a classical graph problem and a useful benchmark for comparing algorithmic approaches and hardware execution models. In this project, we try to compare on the same algorithm (workfront sweep algorithm [1]) 3 hardware execution models : CPU-only, GPU-only, and hybrid CPU+GPU using CUDA unified memory.

## 3 Methodology

### 3.1 Algorithmic approach

Our goal is to speed up a SSSP algorithm with mass parallelization offered by a GPU. Therefore, we chose an approach based on the Bellman-Ford algorithm, which is more parallelizable than the other classical SSSP algorithm : Dijkstra's algorithm.

The algorithm we implemented, is using an approach called called workfront sweep [1], is based on the idea of visiting a vertex only if it is useful in the distances computation. It uses a vertex queue, where is put vertices which their distance has been modified in the previous iteration. During an iteration, each thread takes a vertex from the queue, and handle all of its edges. If a thread updates the distance of another vertex, it put this vertex into the queue of the next iteration. The algorithm stops if the vertex queue is empty.

The following subsections describes our implementations of the workfront sweep algorithm.

### 3.2 Data structure

In graph processing, choosing an efficient data structure is critical for performance and memory management. We utilize the Compressed Sparse Row (CSR) representation, which is specifically optimized for sparse graphs where the number of edges is much lower than the square of the number of vertices. Unlike an adjacency matrix, which requires  $O(V^2)$  space regardless of the number of edges, CSR reduces the memory footprint to  $O(V + E)$ . The CSR format decomposes the graph into three contiguous arrays to ensure coalesced memory access on the GPU:

- **Row Pointers (`row_ptr`):** Stores the starting index in the edge list for each vertex. The number of edges for vertex  $i$  is efficiently calculated as `row_ptr[i + 1] - row_ptr[i]`.
- **Column Indices (`col_idx`):** A contiguous array of size  $E$  containing the destination vertex ID for every edge in the graph.
- **Edge Weights (`weights`):** An array of size  $E$  containing the cost or weight associated with each corresponding edge in the `col_idx` array.

### 3.3 CPU implementation

This implementation uses CPU threads to visit vertices. It uses a concurrent queue structure for the vertex queue, that provides atomic enqueue and dequeue operations using locks.

During an iteration, each thread keeps getting vertices from the queue until it's empty, since there may be way more vertices in the queue than available CPU threads. Output arrays (distances and predecessors) are updated using another lock to ensure consistency. Vertices are marked as updated using a boolean array. At the end of an iteration, the main thread creates a new vertex queue, and iterates over the boolean array to put updated vertices into it, after what the array is reset for a new iteration to begin.

### 3.4 GPU implementation

The algorithm exploits the massively parallel architecture of the GPU by assigning vertices to individual CUDA threads.

The algorithm in practice does the following :

- **Workfront Check:** Each thread identifies its assigned vertex  $i$  and checks if it is currently active by evaluating if `workFront_in[i] == 1`.
- **Neighbor Relaxation:** For every active vertex, the thread iterates through its outgoing edges as defined by the range `[row_ptr[i], row_ptr[i + 1])`.
- **Atomic Operations:** To prevent race conditions when multiple threads attempt to update the distance of the same neighbor simultaneously, the `atomicMin()` function is utilized. This ensures that only the shortest discovered path is stored in the `distances` array.
- **Propagation:** If a thread successfully decreases the distance to a neighbor (`new_dist < old_dist`), it marks that destination vertex as active in the `workFront_out` array, flagging it for processing in the subsequent iteration.

### 3.5 Hybrid implementation using CUDA unified memory

#### 3.5.1 Offloading condition

The idea of our hybrid implementation is for each iteration to use a different processing unit (rather CPU or GPU) depending of how many vertices are in the queue. Indeed, if there is something like only less than 16 vertices in the queue, launching a kernel on a GPU for such an iteration is a waste of time, since we could have done it directly on the CPU.

Therefore, our hybrid implementation offloads an iteration of the algorithm to the GPU only if the queue contains more vertices than available CPU threads. We chose this offloading condition with the idea that during an iteration, each thread should handle only one vertex, but other offloading conditions may lead to different speed results.

#### 3.5.2 Memory management

In order to make graph (csr arrays) and output data (distances and updated vertices arrays) accessible from both CPU and GPU, we used CUDA unified memory to allocate our arrays.

Then, since our boolean array marking updated vertices is accessible from both sides, we can easily do a GPU iteration after a CPU iteration and vice-versa. At the beginning of an iteration, the main thread refills the vertex queue of the chosen processing unit (even the device vertex queue can be modified by the main thread since it is also allocated using unified memory), and then launches threads. These threads can read data from the graph, update distances and mark vertices without any manual data transfer needed thanks to unified memory.

At the end of the iteration, the main thread counts how many vertices were marked during this iteration, and then choose on which processing unit to launch next iteration.

## 4 Experimental Setup

This section describes the hardware/software environment used to run experiments and profile performance.

### 4.1 Hardware Platform

Experiments were run on our own laptops, Google Colab T4, and rented H200 node [1].

- CPU: INTEL(R) XEON(R) PLATINUM 8592+
- vCPU: 24
- RAM: 235 GiB
- GPU: NVIDIA H200, 143771 MiB, 575.57.08

### 4.2 Software Stack and Tools

- OS: Ubuntu 22.04.4 LTS
- Kernel: 5.15.0-113-generic
- Compiler: GCC 11.4.0, NVCC 12.9.86
- CUDA: cuda\_12.9.r12.9\_compiler.36037853\_0
- Libraries: thrust
- Profiling tools: nsu, perf
- Build system: makefile

## 5 Results

### 5.1 Performance and Profiling

We run basic time profiling with `perf` [2] with standard graph problems. We used open datasets from SNAP project[3]. Edge weights were generated randomly, if missing from the source graph. Test runs were run with same seed for all runs. Runs were run 10 times to get average and spread. The single node that the shortest path was calculated was node 0. Correctness was asserted by measurements on very small test graphs that were manually checked. Datasets considered are in Table 1.

On small datasets CPU solver dominates (Fig 1), and GPU based solvers are consistently equal across problem size. Indicating that problems are too small and time is spent in just initializing CUDA runtime at 300ms. Larger graphs (Fig. 2) Show a nice speed up of the GPU and hybrid approaches, across all problems, yet, for road networks

Dataset	Graph size ( $ V  +  E $ )	$ V $ (nodes)	$ E $ (edges)	$d(d_{0.9})$
amazon0312	3,601,167	400,727	3,200,440	7.6
p2p-Gnutella31	210,478	62,586	147,892	6.7
roadNet-CA	4,731,813	1,965,206	2,766,607	500
roadNet-PA	2,629,990	1,088,092	1,541,898	530
web-BerkStan	8,285,825	685,230	7,600,595	9.9
Email-Eu-core	26,576	1,005	25,571	2.9
Wiki-Vote	110,804	7,115	103,689	3.8
p2p-Gnutella04	50,870	10,876	39,994	5.4
p2p-Gnutella08	27,078	6,301	20,777	5.5
p2p-Gnutella30	125,010	36,682	88,328	6.6
gplus_combined	13,781,067	107,614	13,673,453	3.0
soc-Epinions1	584,716	75,879	508,837	5.0
soc-LiveJournal1	73,841,344	4,847,571	68,993,773	6.5
soc-Slashdot0811	982,828	77,360	905,468	4.7

Table 1: SNAP dataset sample. Graph size is defined as  $|V| + |E|$ . “diameter” is SNAP’s 90-percentile effective diameter ( $d_{0.9}$ ).

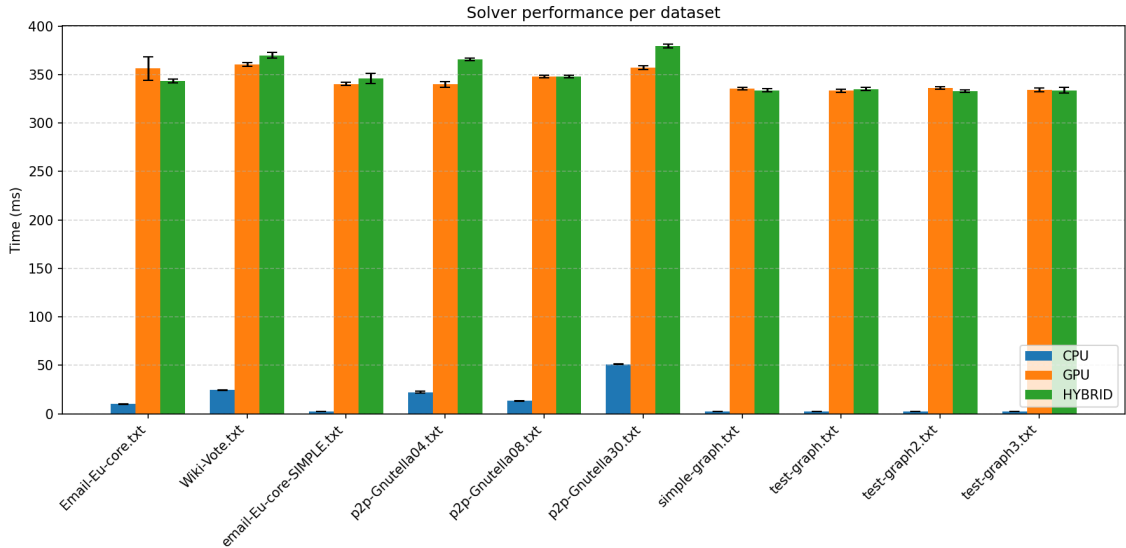


Figure 1: Performance against small datasets

performance speed up is much more pronounced than for other graphs. Testing only on social graphs (Fig. 3 shows much worse performance, where CPU based solver even beat out the GPU based one. This is only due to the structure of the graphs.

We observe that speedup varies strongly with graph structure: road networks (large effective diameter, bounded degree) behave differently from social graphs (power-law degree, many hubs), consistent with atomic contention and load imbalance.

We observed an outlier on `gplus_combined` dataset. Fig 3 A plausible cause is vertex ID range exceeding our internal index type. We did not fully validate this due to lack of time-management. Remapping vertex IDs to a compact range might be required for correct CSR indexing.

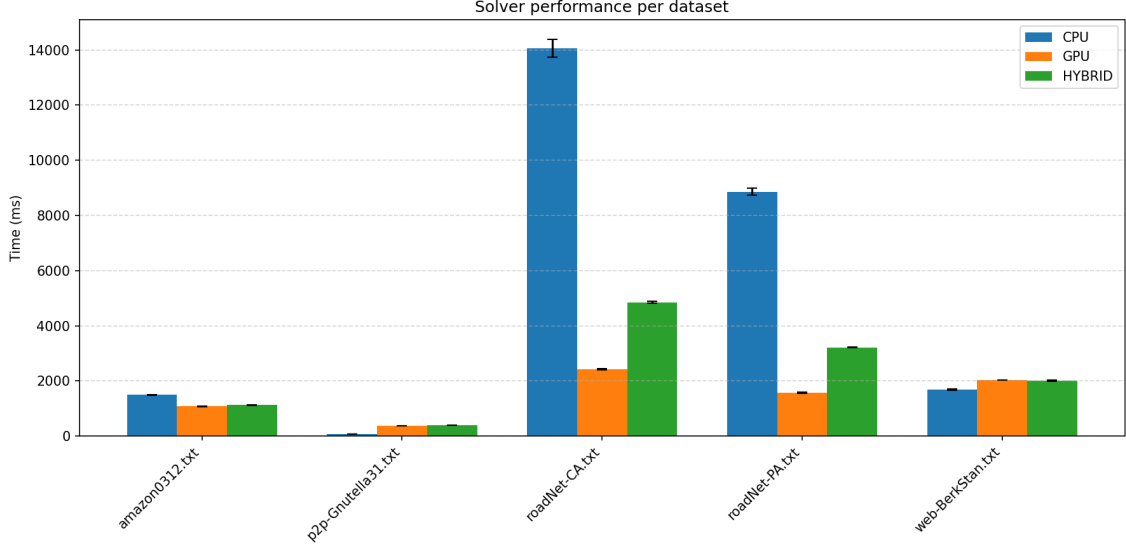


Figure 2: Performance against larger datasets

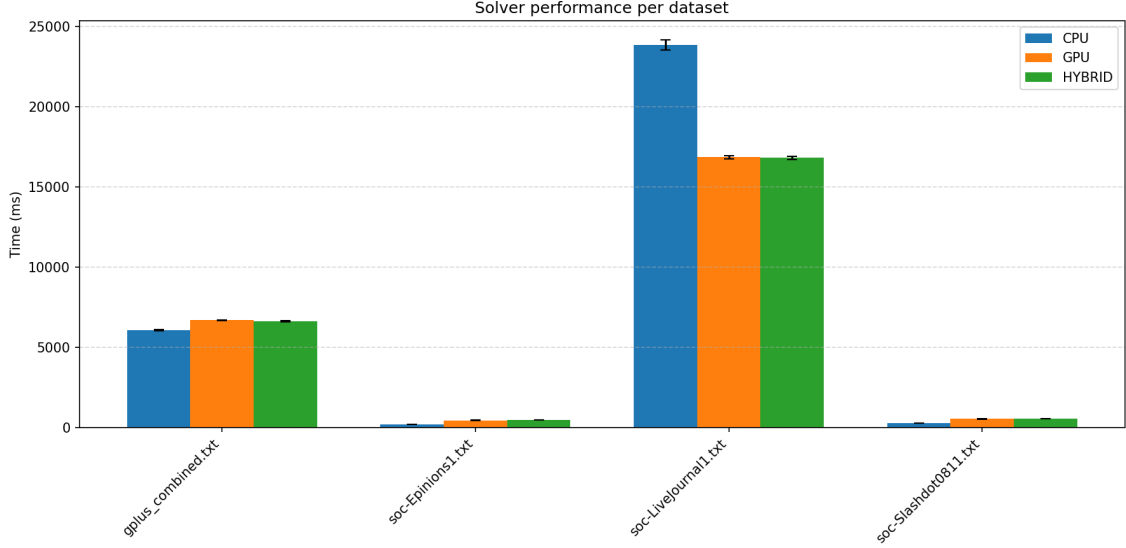


Figure 3: Performance against social graphs

## 6 Conclusion

For small enough graphs basic CPU solver outperforms GPU solvers solely on data locality and startup time of CUDA. Hybrid solver was a waste of an effort, consistently being worse than the GPU solver. And there is noticeable performance difference between social graphs and geometrical graphs *roadNet*. This is the expected consequence of the work sweep algorithm.

### 6.1 Limitations

- **Load Imbalance and Warp Divergence:** The performance of the sweep is highly sensitive to the graph's degree distribution. In power-law graphs, a single thread might process thousands of neighbors while neighboring threads in the same

warp remain idle. This creates a bottleneck where the execution time of a warp is dictated by its most heavily loaded thread, leading to poor hardware utilization.

- **Atomic Contention and Serialization:** The algorithm relies heavily on `atomicMin()`, which is essentially a serialized operation at the hardware level when multiple threads target the same memory address. In high-degree "hub" nodes, hundreds of threads may stall while waiting for their turn to update a single distance value, resulting in significant pipeline stalls and wasted clock cycles.
- **Host-Device data communication :** having to update data with the main thread after each iteration causes a lot of host-device data communication, especially in the hybrid implementation with unified memory, where we do not control those transfers. This may lead to time wasted by waiting for the data to be transferred.

## 6.2 Future Work

- Much more detailed performance analytics, comparing specifically performance against average metrics of graphs (diameter, number of triangles, triangle inequality constraints etc).
- Considering other methods such as near+far Pile which balances the workload better (useful for social network graphs for example).
- Implementing own versions of GPU reduction, binary search , sort etc ... instead of using external libraries that may use both GPU and CPU operations and add passive waiting clock cycles.
- Consider a better offloading condition that can lead to higher execution speed for the hybrid implementation.
- Give advises to unified memory to optimize host-device data transfers.

## References

- [1] Grace Morgan. *Introducing Bare Metal Systems Built on NVIDIA HGX H200*. DigitalOcean. Feb. 18, 2025. URL: <https://www.digitalocean.com/blog/now-available-bare-metal-nvidia-hgx-h200-gpus> (visited on 01/07/2026).
- [2] *perf(1) — Performance analysis tools for Linux*. Linux manual page (man7.org HTML rendering). perf (Linux source tree) project. June 20, 2024. URL: <https://man7.org/linux/man-pages/man1/perf.1.html> (visited on 01/07/2026).
- [3] Jure Leskovec and Andrej Krevl. *SNAP Datasets: Stanford Large Network Dataset Collection*. Stanford Network Analysis Project (SNAP). June 2014. URL: <https://snap.stanford.edu/data/index.html> (visited on 01/07/2026).
- [4] Pawan Harish and P. J. Narayanan. "Accelerating Large Graph Algorithms on the GPU Using CUDA". In: *High Performance Computing (HiPC 2007)*. Vol. 4873. Lecture Notes in Computer Science. Springer, 2007, pp. 197–208. DOI: 10.1007/978-3-540-77220-0\_21. URL: [https://www.nvidia.com/content/cudazone/CUDABrowser/downloads/Accelerate\\_Large\\_Graph\\_Algorithms/HiPC.pdf](https://www.nvidia.com/content/cudazone/CUDABrowser/downloads/Accelerate_Large_Graph_Algorithms/HiPC.pdf).

- [5] Andrew Davidson et al. “Work-Efficient Parallel GPU Methods for Single-Source Shortest Paths”. In: *2014 IEEE 28th International Parallel & Distributed Processing Symposium (IPDPS)*. IEEE, 2014, pp. 349–359. DOI: 10.1109/IPDPS.2014.45. URL: <https://readingxtra.github.io/docs/gpu-graph/DavidsonIPDPS2014.pdf>.
- [6] Kai Wang, Don Fussell, and Calvin Lin. “A Fast Work-Efficient SSSP Algorithm for GPUs”. In: *Proceedings of the 26th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP ’21)*. ACM, 2021, pp. 133–146. DOI: 10.1145/3437801.3441605. URL: <https://www.cs.utexas.edu/~lin/papers/ppopp21.pdf>.
- [7] *HAL preprint (see URL for full bibliographic metadata)*. HAL/Inria open archive. HAL Id: hal-01760642v1. Accessed 2026-01-07. URL: <https://inria.hal.science/hal-01760642v1>.
- [8] NVIDIA Corporation. *Thrust: A Parallel Algorithms Library*. NVIDIA Developer Documentation. URL: <https://developer.nvidia.com/thrust>.