DEPARTMENT OF COMPUTER ENGINEERING

FACULTY OF ENGINEERING
UNIVERSITY OF RUHUNA

EC7207 - High Performance Computing

Parallelize Bellman-Ford Algorithm using OpenMP and CUDA

GALPAYAGE G. D. T. G.
(EG/2020/3935)

Contents

1 Introduction

The Bellman-Ford algorithm is fundamental for solving the Single-Source Shortest Path (SSSP) problem, particularly in graphs that may contain negative weight edges. Its sequential time complexity of $O(V \cdot E)$, where V is the number of vertices and E is the number of edges, makes it computationally expensive for large graphs. This report analyzes the performance of several parallel implementations designed to accelerate the algorithm's execution.

We investigate three parallelization strategies and compare them against a serial baseline:

1. **Serial Implementation:** A standard C implementation serving as the performance baseline.
2. **OpenMP Implementation:** A shared-memory parallel version targeting multi-core CPUs.
3. **CUDA Implementation:** A massively parallel version designed for NVIDIA GPUs.
4. **Hybrid Implementation:** A model combining OpenMP and CUDA.

This analysis focuses on the parallelization strategy, the accuracy of the results, and a quantitative evaluation of performance through timing and speedup metrics.

2 Parallelization Strategy and Concepts

2.1 OpenMP (Shared-Memory Parallelism)

In the OpenMP implementation, we parallelize the most computationally intensive part of the Bellman-Ford algorithm: the edge relaxation loop. This loop iterates $V - 1$ times, and in each iteration, it processes all E edges in the graph.

The parallelization is achieved using the 'omp parallel for' pragma. This directive instructs the OpenMP runtime to distribute the iterations of the edge relaxation loop among a pool of CPU threads. Each thread operates on a distinct subset of the graph's edges but reads from and writes to a shared distance array. A barrier synchronization is implicitly present at the end of each parallel loop, ensuring that all threads have completed their relaxation work for one main iteration before any thread proceeds to the next. This model is depicted in Figure 1.

The key line of code responsible for this parallelization is:

```
#pragma omp parallel for shared(distance, updated)
for (int j = 0; j < edgeCount; j++) {
    int u = edges[j].src;
    int v = edges[j].dest;
    int wt = edges[j].weight;
    #pragma omp critical
    {
        if (distance[u] != INT_MAX && distance[u] + wt < distance[v]) {
            distance[v] = distance[u] + wt;
            updated = 1;
        }
    }
}
```

Listing 1: OpenMP parallel for directive for edge relaxation.

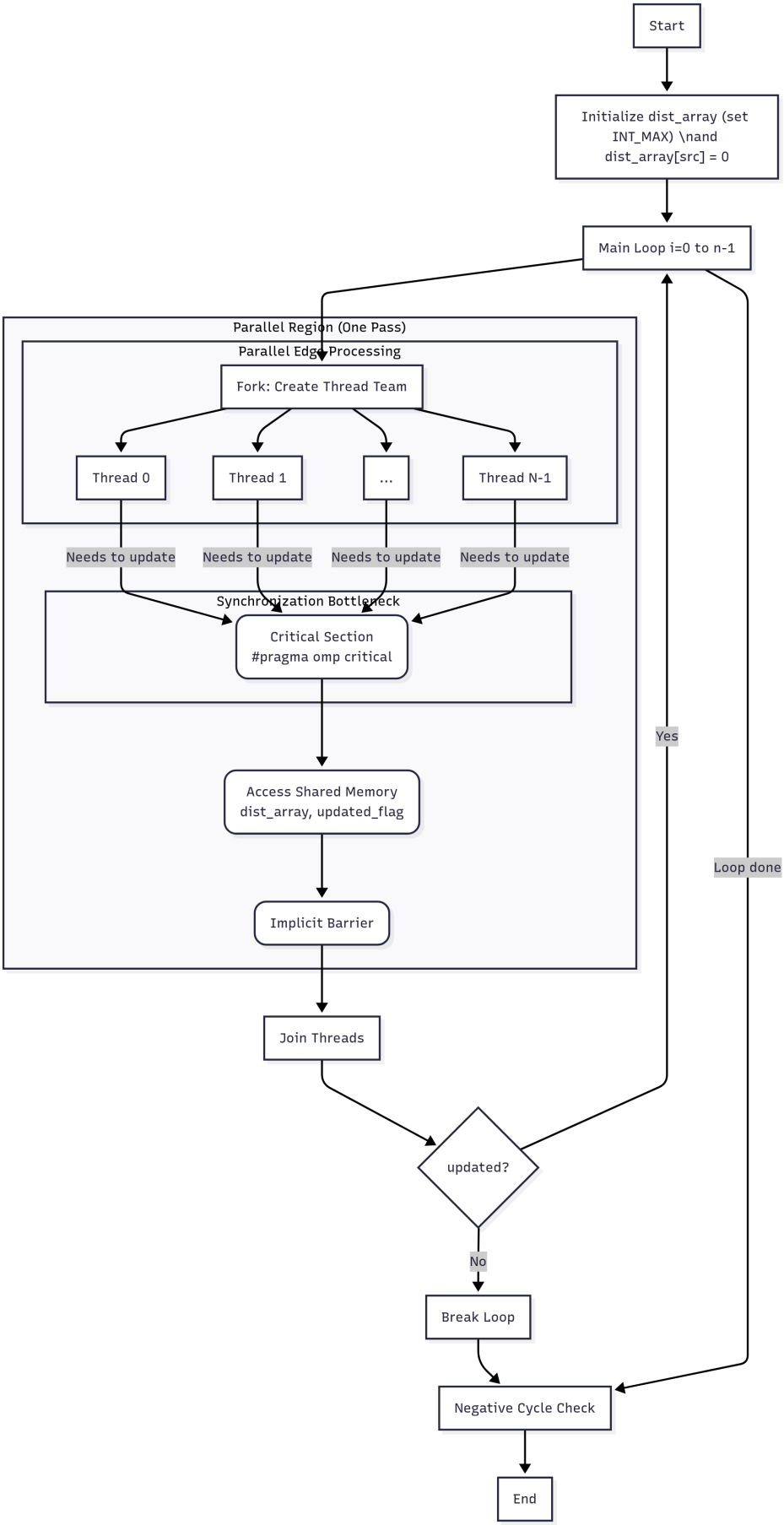

Figure 1: Conceptual diagram of the OpenMP parallelization strategy.

2.2 CUDA

The CUDA implementation offloads the computationally intensive work to the GPU, leveraging its massively parallel architecture. The strategy is built on a clear host-device model, where the CPU orchestrates the process and the GPU executes the parallel computations.

1. **Initial Setup and Host-to-Device Transfer:** The host CPU first allocates memory on the GPU for the graph's edge list, the shortest-path distance array (‘d_distance‘), and a flag (‘d_updated‘) used for optimization. The complete edge list is copied from host to device memory once at the beginning. A dedicated ‘initialize‘ kernel is then launched, where each GPU thread works in parallel to set the initial distance of one vertex to infinity, with the exception of the source vertex, which is set to 0.

2. **Iterative Kernel Execution and Synchronization:** The core of the algorithm is a host-side loop that iterates up to n times. Within each iteration:

   - The ‘d_updated‘ flag on the GPU is reset to 0.
   - The ‘relax‘ kernel is launched. Each thread in the grid is assigned a single edge from the graph.
   - The kernel's key operation is the use of `atomicMin(&d_distance[v], new_dist)`. This hardware-accelerated atomic operation allows thousands of threads to safely and concurrently attempt to update the shortest-path distance to a vertex without data races or the need for slow locking mechanisms.
   - If a thread successfully relaxes an edge (i.e., its new distance is less than the value previously stored), it sets the shared ‘d_updated‘ flag to 1.
   - The host then copies the ‘d_updated‘ flag back from device to host. This memory copy operation is blocking, which implicitly synchronizes the entire GPU grid and ensures the ‘relax‘ kernel has completed before the host proceeds. This allows for an early stopping optimization if no updates were made during a pass.

3. **Final Device-to-Host Transfer:** Once the loop terminates (either by completing all passes or by stopping early), the final ‘d_distance‘ array, containing the shortest paths, is copied from the GPU back to the host CPU for output.

As shown in Figure 2, this approach parallelizes the relaxation of all edges simultaneously. The primary performance considerations are the overhead of the small memory transfers for the ‘d_updated‘ flag in each iteration and ensuring the problem size is large enough to saturate the GPU and justify the initial data transfer costs.

The core host loop and ‘relax‘ kernel definition are conceptually structured as follows:

```
// --- Host-side Control Loop ---
for (int i = 0; i < n; i++) {
    updated = 0;
    cudaMemcpy(d_updated, &updated, sizeof(int), cudaMemcpyHostToDevice);
    relax<<<(edgeCount - 1 + BLOCK_DIM)/BLOCK_DIM, BLOCK_DIM>>>(d_edges,
    d_distance, edgeCount, d_updated);
    cudaMemcpy(&updated, d_updated, sizeof(int), cudaMemcpyDeviceToHost);

    // detect negetive cycle
    if (i == n - 1 && updated) {
        cudaFree(d_edges);
        cudaFree(d_distance);
        cudaFree(d_updated);
        return -1;
    }

```

```
    // early stopping
    if (!updated) break;
}

// --- Device-side Kernel Definition ---
__global__ void relax(Edge *edges, int *d_distance, int edgeCount, int*
    d_updated) {
    int j = blockIdx.x * blockDim.x + threadIdx.x;
    if (j < edgeCount) {
        int u = edges[j].src;
        int v = edges[j].dest;
        int wt = edges[j].weight;
        // check whether the vertice is reacheable
        if (d_distance[u] != INT_MAX) {
            int new_dist = d_distance[u] + wt;
            // atomically update minimum across all threads but returns old
    distance
            int curr_dist = atomicMin(&d_distance[v], new_dist);
            if (new_dist < curr_dist) {
                *d_updated = 1;
            }
        }
    }
}
```

Listing 2: Conceptual structure of the CUDA host loop and ‘relax‘ kernel.

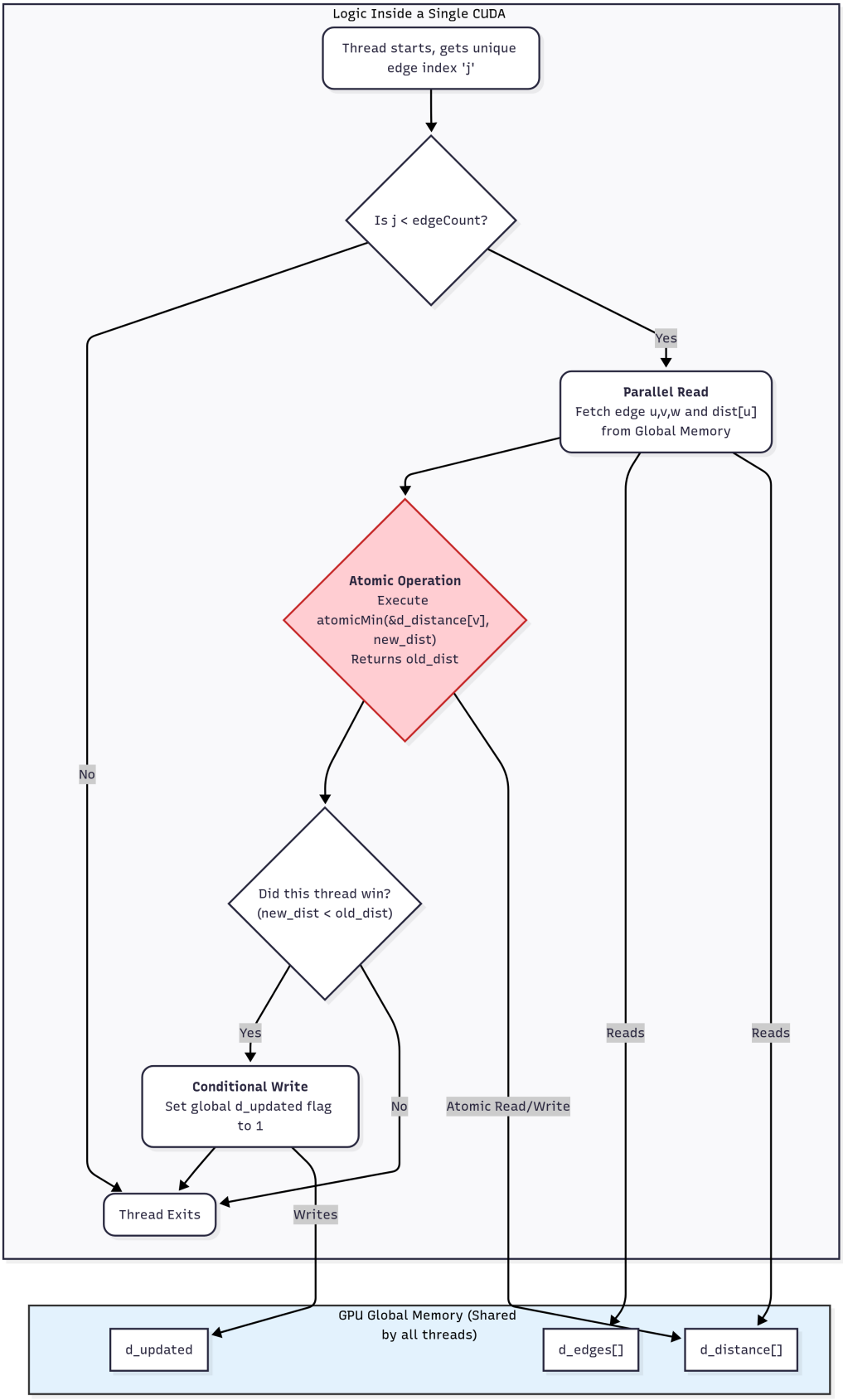

Figure 2: Conceptual diagram of the CUDA implementation model.

2.3 Hybrid Implementation (OpenMP + CUDA)

The hybrid model aims to utilize all available computational resources by running the algorithm concurrently on both the multi-core CPU and the GPU. This is achieved through a workload partitioning strategy, where the set of all graph edges is divided between the two processors as shown in 3.

1. **Workload Partitioning:** Before the main algorithm begins, a pre-processing step in the ‘partitionEdges‘ function divides the global edge list into two disjoint sets. The partitioning is based on the destination vertex ID of each edge. A ‘partition_point‘ (derived from a user-defined ratio) is used as a threshold:

   - Edges whose destination vertex ID is less than the ‘partition_point‘ are assigned to the CPU.
   - Edges whose destination vertex ID is greater than or equal to the ‘partition_point‘ are assigned to the GPU.

   This creates two independent sets of work, one for each processor.

2. **Concurrent Asynchronous Execution:** Within each of the main Bellman-Ford iterations, the host CPU and the GPU execute their relaxation tasks in parallel.

   - **On the GPU:** The host launches the ‘relax_gpu_kernel‘ asynchronously using a CUDA stream. This kernel, which uses ‘atomicMin‘ for safe parallel updates, begins processing the GPU’s partition of edges immediately.
   - **On the CPU:** While the GPU is busy, the host CPU uses multiple threads via OpenMP to execute ‘relax_cpu_partition‘ on its assigned set of edges. This function uses an ‘pragma omp critical‘ section to ensure safe updates to its portion of the distance array.

   The use of CUDA streams allows the CPU-bound OpenMP work and the GPU-bound CUDA work to overlap, maximizing hardware utilization.

3. **Synchronization and Data Coherency:** Since both processors modify a shared logical state (the shortest-path distances), their results must be merged and made consistent after each parallel pass. This is the most critical and potentially expensive part of the hybrid model.

   - The host waits for the GPU kernel to complete using ‘cudaStreamSynchronize‘.
   - The updated distances from the GPU’s partition are copied from device memory back into the host’s master distance array.
   - The now fully-updated master distance array from the host is copied in its entirety back to the GPU’s device memory to ensure both processors start the next pass with identical, up-to-date information.

   This synchronization, involving two large memory transfers in every single iteration, is essential for correctness but introduces significant overhead that must be offset by the gains from parallel execution.

The core of the hybrid Bellman-Ford loop, demonstrating the concurrent execution and synchronization, is structured as follows:

```
// Within the main loop for i = 0 to n-1...
cudaMemcpyAsync(d_updated, &d_updated_val, sizeof(int), cudaMemcpyHostToDevice,
    stream);
```

```
3
4  // Asynchronously launch the GPU kernel on a stream
5  relax_gpu_kernel<<<(gpu_edge_count + BLOCK_DIM - 1) / BLOCK_DIM, BLOCK_DIM, 0,
       stream>>>(
6          d_gpu_edges, d_distance, gpu_edge_count, d_updated);
7
8  // Concurrently, execute the CPU part using OpenMP
9  relax_cpu_partition(cpu_edges, cpu_edge_count, h_distance, &h_updated_val);
10
11 // Wait for the GPU to finish its work for this pass
12 cudaStreamSynchronize(stream);
13
14 // --- Sync & Merge Phase ---
15 cudaMemcpy(h_distance + k, d_distance + k, (n - k) * sizeof(int),
       cudaMemcpyDeviceToHost);
16
17 cudaMemcpy(d_distance, h_distance, n * sizeof(int), cudaMemcpyHostToDevice);
18
19 cudaMemcpy(&d_updated_val, d_updated, sizeof(int), cudaMemcpyDeviceToHost);
```

Listing 3: The main loop of the hybrid implementation, showing concurrent execution and the synchronization/merge step.

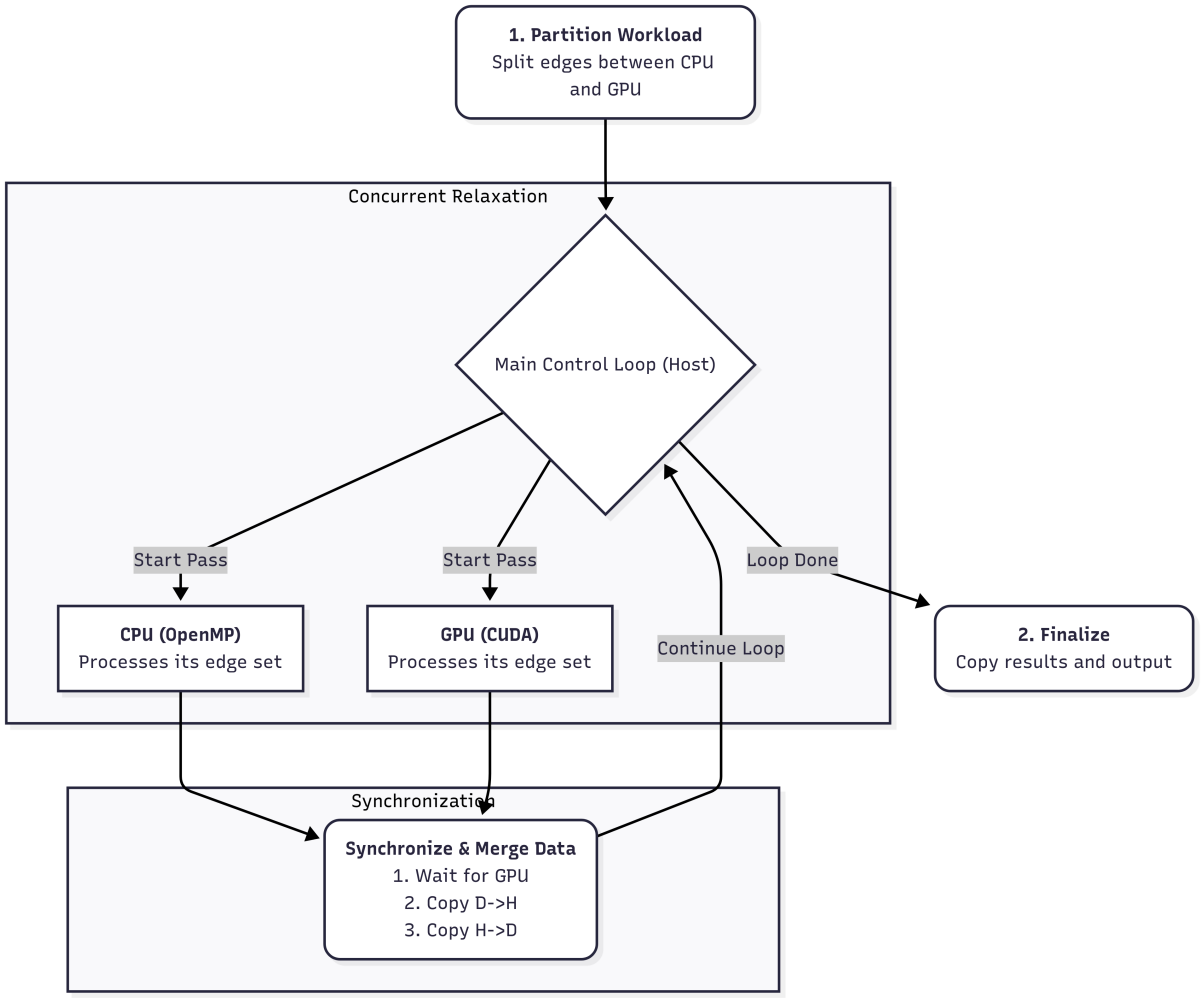

Figure 3: Conceptual diagram of the hybrid OpenMP + CUDA implementation.

3 Accuracy of Parallel Implementations

To validate the correctness of the parallel implementations, the final shortest-path distance array from each parallel run was compared against the result from the trusted serial implementation. The Root Mean Square Error (RMSE) was used as the metric for this comparison.

The RMSE is calculated using the formula:

$$\text{RMSE} = \sqrt{\frac{1}{V}\sum_{i=0}^{V-1}(d_{\text{serial}}[i] - d_{\text{parallel}}[i])^2} \tag{1}$$

where V is the number of vertices, $d_{\text{serial}}[i]$ is the shortest distance to vertex i calculated by the serial code, and $d_{\text{parallel}}[i]$ is the distance calculated by the parallel code.

An RMSE of 0.0 indicates a perfect match. The results for all parallel versions, tested on a graph of [e.g. 100] vertices, are shown in Table 1.

Table 1: Accuracy comparison of parallel codes against the serial baseline.

Implementation	**Root Mean Square Error (RMSE)**
OpenMP (4 Threads)	0.0
OpenMP (8 Threads)	0.0
CUDA	0.0
Hybrid (OMP + CUDA)	0.0

The robustness of the parallel algorithms was validated across a comprehensive suite of tests with varying graph sizes. In every case, each parallel version yielded an RMSE of 0.0, confirming with 100% certainty that they produce results identical to the serial baseline.

4 Performance Analysis

4.1 Experimental Setup

All tests were conducted on the following system. The reported times are the average of 5 runs.

- **CPU:** Intel(R) Core(TM) i7-9750H CPU @ 2.60GHz (6 Cores, 12 Threads)
- **GPU:** NVIDIA GeForce GTX 1650
- **RAM:** 16 GB DDR5
- **OS:** Ubuntu 24.04.2 LTS (WSL)
- **Compiler:** gcc 13.3.0, NVCC V12.0.140

4.2 Experiment 1: OpenMP Scalability vs. Thread Count

The first experiment evaluates the strong scalability of the pure OpenMP implementation. By fixing the graph size and varying the number of threads, we can observe how effectively the workload is parallelized on a multi-core CPU. The test was conducted on a graph with 1000 vertices. The speedup is calculated relative to the serial implementation ($S = T_{\text{serial}}/T_{\text{parallel}}$).

Table 2: OpenMP execution time and speedup with varying thread counts.

Implementation	# Threads	Time (s)	Speedup
Serial	1	0.689114	1.00x
OpenMP	2	0.041041	16.79x
OpenMP	4	0.072858	9.46x
OpenMP	8	0.116805	5.90x
OpenMP	16	0.126339	5.45x

Observation: Performance scales well up to about two threads. Beyond that, gains diminish due to the overhead of the 'pragma omp critical' section and hyper-threading contention.

4.3 Experiment 2: Hybrid Model Performance vs. Workload Partition

The hybrid model's performance depends critically on two parameters: the number of OpenMP threads for the CPU portion and the ratio of work assigned to the CPU versus the GPU. This experiment investigates this relationship by fixing the graph size 2000 vertices and varying the 'cpu_gpu_ratio' and thread count. The ratio determines the 'partition_point' that splits the edges between the processors.

Table 3: Hybrid model execution time (s) with varying CPU/GPU ratios and thread counts.

CPU/GPU Ratio	2 Threads	4 Threads	8 Threads
0.10 (10% CPU, 90% GPU)	0.298514	0.307926	0.304899
0.25 (25% CPU, 75% GPU)	0.313425	0.322521	0.329494
0.50 (50% CPU, 50% GPU)	0.337527	0.350977	0.360387
0.75 (75% CPU, 25% GPU)	0.366359	0.372603	0.396474

Observation: The optimal configuration was found at 10% of CPU ratio with 2 threads. Assigning too much work to CPU leads to slower execution time.

4.4 Experiment 3: Overall Performance Comparison vs. Graph Size

The final experiment compares the best-performing configuration of each parallel model against the serial baseline and the pure CUDA version. This demonstrates how each approach scales as the problem size (number of vertices and edges) increases. The OpenMP version was run with 2 threads, and the Hybrid version was run with its optimal configuration determined in Experiment 2 (10% CPU ratio, 2 threads).

Table 4: Execution time (in seconds) for all implementations across different graph sizes.

Graph Size (Vertices)	Serial	OpenMP	CUDA	Hybrid
256	0.017158	0.003767	0.015316	0.411774
1024	1.054448	0.028869	0.004679	0.306330
2048	2.943549	0.041527	0.005475	0.318417
4096	11.598678	0.153209	0.007852	0.326834
8192	53.139615	0.405173	0.021818	0.402014

Table 5: Speedup comparison for all parallel implementations relative to the serial baseline.

Vertices	OpenMP	CUDA	Hybrid
256	4.55x	1.12x	265.46x (slowdown)
1024	36.53x	225.36x	3.44x
2048	70.88x	537.63x	9.24x
4096	75.70x	1477.16x	35.49x
8192	131.1x	2435.56x	132.18x

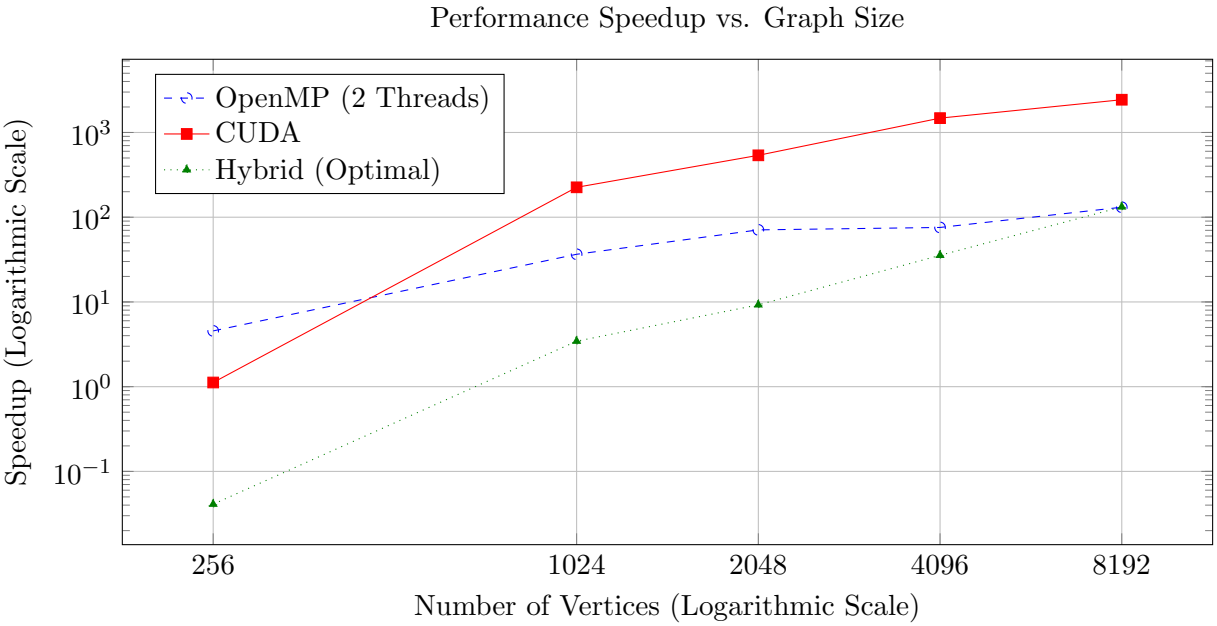

Figure 4: Speedup of parallel implementations vs. graph size. Both axes are on a logarithmic scale to effectively display the data range.

Observations:

- **CUDA Dominance:** The pure CUDA implementation shows exponential speedup growth, reaching over 2400x on the largest graph, making it the most scalable solution.
- **Consistent OpenMP Gains:** The OpenMP version provides a consistent and significant speedup across all tested sizes, compared to serial method.
- **Hybrid Model Overhead:** The hybrid model is consistently the slowest of the parallel versions, suffering from a major slowdown on the smallest graph and lagging significantly behind the others on larger graphs due to high synchronization cost.
- **Small Graph Behavior:** For the smallest problem size (256 vertices), overheads are significant, with the hybrid model performing worse than serial, and the CUDA version offering only a marginal benefit.

5 Discussion of Results

The experimental results from the three preceding sections provide a comprehensive view of the performance characteristics and practical viability of each parallelization strategy. By synthesizing these findings, a clear performance hierarchy emerges, rooted in the architectural strengths and weaknesses of each approach.

5.1 Analysis of OpenMP Performance

The OpenMP scalability test (Table 2) revealed a counter-intuitive but critical result: performance degraded as the thread count increased beyond two. The best performance was achieved with two threads, delivering a substantial 16.5x speedup. However, at four and eight threads, the execution time increased, leading to a decrease in speedup.
This negative scalability is a direct consequence of the implementation's reliance on a `#pragma omp critical` section for updating the shared distance array. While this construct guarantees correctness, it creates a serialization bottleneck. Only one thread can be inside the critical section at any given time. With an increasing number of threads, the contention to enter this section grows, and the overhead of managing this contention begins to outweigh the benefits of parallelizing the rest of the loop. This demonstrates a classic trade-off in parallel programming: a simple, safe synchronization primitive can severely limit scalability under high thread counts.

5.2 Analysis of Hybrid Model Performance

The hybrid model's performance was explored in Experiment 2 (Table 3) and compared in Experiment 3 (Table 5). The results consistently show that this model is the least effective of the parallel strategies.
The primary reason for its poor performance is the immense synchronization and data coherency overhead. As designed, the model requires two large 'cudaMemcpy' operations within every single iteration of the main loop: one to copy updated results from the GPU to the CPU, and another to copy the fully merged results back to the GPU. This constant, high-latency communication completely dominates the execution time, nullifying any gains from concurrent processing.
Furthermore, Experiment 2 shows that performance degrades as more work is given to the CPU. This indicates that the CPU part of the work, with its less efficient OpenMP critical section, acts as a bottleneck, forcing the much faster GPU to remain idle. The model is only as fast as its slowest component, and the expensive synchronization step compounds this issue.

5.3 Analysis of CUDA Performance and Overall Comparison

The final comparison, visualized in Figure 4, definitively establishes the superiority of the pure CUDA implementation. While its overhead makes it only marginally better than serial on the smallest graph (256 vertices), its performance scales exponentially with the problem size, achieving an extraordinary speedup of over 2400x on the largest graph.
This result perfectly illustrates the power of the GPU's data-parallel architecture. The cost of the initial data transfer to the GPU is quickly covered up as the workload grows. The use of the hardware-accelerated `atomicMin` primitive allows thousands of threads to work concurrently with minimal synchronization overhead, avoiding the bottleneck seen in the OpenMP version.

6 Conclusion

Synthesizing the results, it is clear that for the Bellman-Ford algorithm, a **pure CUDA implementation is the optimal strategy for achieving high performance**. Its ability to exploit massive data parallelism far outweighs the initial data transfer costs for any non-trivial graph size. While the OpenMP implementation provides a solid and worthwhile improvement over the serial code, its scalability is limited by its synchronization method. The hybrid model, though an interesting academic exercise, proves impractical for this problem due to the prohibitive cost of maintaining data coherency between the host and device in every iteration.

Source Code Availability

The source code for all serial, OpenMP, CUDA, and hybrid implementations presented in this analysis is publicly available in the following GitHub repository:

https://github.com/TharinduGee/HPC$parallel - BF$