CUDA-Based Cycle Detection in Directed Graphs Using Kahn's Algorithm

Francesco Granata 1000046547

July 24, 2025

Contents

1 Introduction

Graphs are foundational data structures used to represent relationships and dependencies across diverse domains such as scheduling, compiler optimization, social networks, and data pipelines. A particularly important problem in directed graphs is the detection of cycles, which is crucial for validating correctness in systems involving dependencies or execution order.

Kahn's algorithm is a well-known method for topological sorting and can be adapted to detect cycles by attempting to process all nodes in a graph. However, in its traditional sequential form, it becomes a bottleneck when applied to large-scale graphs. To overcome this limitation, parallel processing—especially using the highly parallel architecture of modern GPUs—offers a promising direction.

This project presents two parallelized implementations of Kahn's algorithm using the CUDA programming model: one designed for graphs represented by adjacency matrices, and another for graphs using the Compressed Sparse Row (CSR) format. These two versions take into account the different memory access patterns and storage layouts inherent to each representation, requiring distinct optimizations and data structures.

By leveraging GPU acceleration, both implementations aim to achieve high-throughput computation for in-degree calculation, prefix scans, and the core node-processing steps required for cycle detection. For each graph format, multiple CUDA kernel designs are explored to optimize key operations. For instance, the in-degree calculation is implemented through different strategies, from straightforward loops to more sophisticated tile-based or atomic-based approaches. Likewise, the identification and extraction of zero in-degree nodes employs compacting techniques inspired by stream compaction.

Through this exploration, we evaluate the trade-offs in warp efficiency, memory access patterns, and synchronization overhead specific to each graph representation. This report documents the design decisions, implementation strategies, and performance analysis of the two systems, highlighting both the challenges and opportunities involved in using CUDA for parallel graph cycle detection.

2 Background

2.1 Graphs and Cycle Detection

A graph is a fundamental data structure used to represent a collection of entities (called *nodes* or *vertices*) and the relationships (*edges*) between them. In a *directed graph* (or *digraph*), each edge has a direction, going from one node to another. These structures are widely used in various fields such as task scheduling, dependency resolution, and compiler design.

A graph is said to be *acyclic* if it does not contain any cycles — that is, there is no path that starts and ends at the same node by following the direction of the edges. Detecting whether a directed graph contains a cycle is a critical task in many applications, as cycles often represent logical errors or invalid dependencies.

2.2 Kahn's Algorithm

Kahn's algorithm is a classical method for performing a topological sort of a directed acyclic graph (DAG). It works by repeatedly removing nodes with zero in-degree (nodes that have no incoming edges) and decrementing the in-degrees of their neighbors. If at any point there are no nodes with zero in-degree and there are still nodes remaining, the graph contains a cycle.

This algorithm is particularly well-suited for cycle detection: if the number of processed nodes is less than the total number of nodes, then the graph is not acyclic. Its sequential implementation is simple and efficient for small graphs, but for large-scale graphs, especially in high-performance computing contexts, a parallel version becomes desirable.

2.3 CUDA Programming Model

CUDA (Compute Unified Device Architecture) is a parallel computing platform and programming model developed by NVIDIA. It enables developers to harness the computational power of NVIDIA GPUs for general-purpose computing. CUDA provides an extension to the C/C++ programming languages that allows code to be executed on the GPU in the form of *kernels*.

The CUDA execution model is based on a hierarchy of threads: individual threads are grouped into blocks, and blocks are organized into a grid. Each thread executes the same kernel code but can operate on different data. This model is particularly well-suited for data-parallel problems where the same operation is applied to many elements independently.

Memory hierarchy is another key feature of CUDA, with multiple levels of memory including global, shared, and local memory, each with different latency and access characteristics. Efficient use of memory and careful thread coordination are essential for achieving high performance.

In the context of graph algorithms, CUDA offers the potential to parallelize operations such as computing in-degrees, identifying nodes with certain properties, and performing prefix sums. However, challenges such as irregular memory access patterns, synchronization overhead, and load balancing must be addressed to fully exploit GPU capabilities.

3 Problem Statement

Given a directed graph $G = (V, E)$, where V is the set of vertices and $E \subseteq V \times V$ is the set of directed edges, the goal of this project is to determine whether G contains any cycles. If no cycles exist, the graph is classified as a Directed Acyclic Graph (DAG).

Formally, the problem can be defined as follows:

- **Input**: A directed graph represented either as an adjacency matrix or in Compressed Sparse Row (CSR) format.
- **Output**: A Boolean result indicating whether the graph contains a cycle.

This problem has critical applications in domains such as:

- Task scheduling, where a cycle in dependencies renders execution impossible.
- Build systems, where cyclic dependencies between modules must be detected and resolved.
- Compiler design, where topological ordering is used for evaluating expression trees or dependency graphs.

In a sequential setting, this problem can be solved efficiently using depth-first search (DFS) or Kahn's algorithm. However, with the increasing size of graph-structured data, particularly in data processing pipelines and dependency analysis systems, sequential approaches become inadequate in terms of performance.

The challenge, therefore, lies in designing a parallel solution suitable for GPU execution using CUDA. This includes addressing:

- Efficient parallel computation of node in-degrees.
- Fast identification and compaction of zero in-degree nodes.
- Updating the graph structure in parallel while maintaining correctness.
- Handling irregular memory access patterns inherent in graph traversal.

The objective of this project is to implement a parallelized version of Kahn's algorithm for cycle detection on the GPU, evaluate its performance, and explore multiple kernel-level implementations and optimizations for scalability and efficiency.

4 Graph Representations

Efficient graph representation is a fundamental design choice in any graph algorithm implementation, particularly when targeting performance-critical environments such as GPUs. This project supports two primary representations: the *adjacency matrix* and the *Compressed Sparse Row (CSR)* format. Each has advantages and limitations depending on the graph's density and the operations being performed.

4.1 Adjacency Matrix

An adjacency matrix is a 2D array A of size $n \times n$, where $n = |V|$ is the number of nodes in the graph. The element $A[i][j]$ is set to 1 if there is an edge from node i to node j, and 0 otherwise.

- **Advantages:**
  - Constant-time edge lookup ($O(1)$).
  - Simple indexing and straightforward parallel access patterns.
  - Well-suited for dense graphs where many edges are present.
- **Disadvantages:**
  - Poor memory efficiency for sparse graphs: requires $O(n^2)$ space regardless of the number of edges.
  - Inefficient traversal for graphs with low edge density.

In CUDA, the adjacency matrix allows for coalesced access when rows are processed in parallel, but also wastes memory and bandwidth when many entries are zero. It is primarily used in this project for testing and comparison purposes.

4.2 Compressed Sparse Row (CSR)

The Compressed Sparse Row (CSR) format is a memory-efficient representation well-suited for sparse graphs. It uses three arrays:

- `row_ptr` – An array of size $|V| + 1$, where `row_ptr`[i] gives the index in the `col_ind` array where the adjacency list of node i begins.
- `col_ind` – A flattened array of destination node indices for all outgoing edges.
- `val` (optional) – If edge weights are needed; omitted in this unweighted implementation.

Example:

$$\texttt{row\_ptr} = [0, 2, 3, 5], \quad \texttt{col\_ind} = [1, 2, 0, 0, 2]$$

This means:

- Node 0 connects to nodes 1 and 2,
- Node 1 connects to node 0,
- Node 2 connects to nodes 0 and 2.

- **Advantages:**
  - Efficient memory usage: requires $O(|V| + |E|)$ space.
  - Ideal for sparse graphs where $|E| \ll |V|^2$.

    - Allows compact traversal of outgoing edges.

- **Disadvantages:**
    - Indirect memory access patterns in CUDA can lead to lower coalescing and cache efficiency.
    - Edge lookup is $O(\text{degree})$, not constant-time.

CSR is used in this project to efficiently process large sparse graphs on the GPU, minimizing memory footprint while still supporting parallel traversal during cycle detection.

4.3 Comparison and Selection

The choice between adjacency matrix and CSR is determined by the graph's density and the intended operations:

- The adjacency matrix is easier to implement and parallelize for small or dense graphs, offering fast access at the cost of high memory usage.
- CSR offers significant performance and memory advantages for sparse graphs, which are common in real-world applications such as dependency trees or scheduling systems.

Both representations are implemented and tested in this project to allow direct comparison of performance, flexibility, and implementation complexity.

5 Algorithm Design

This section outlines the high-level structure of the parallel algorithm used to detect cycles in a directed graph, based on a modified version of Kahn's topological sorting algorithm. The algorithm is designed to run efficiently on a GPU using CUDA, taking advantage of massive parallelism while addressing data dependency and synchronization challenges inherent in graph traversal.

5.1 Overview of Parallel Kahn's Algorithm

Kahn's algorithm for topological sorting proceeds by iteratively removing all nodes with zero in-degree and updating the in-degrees of their neighbors. If all nodes can be removed in this way, the graph is acyclic. Otherwise, a cycle must be present.

To parallelize this process, we divide the algorithm into distinct phases, each of which can be executed in parallel across GPU threads. A key modification is the use of scan (prefix sum) and compaction operations to manage dynamic lists of active nodes.

1. Compute the in-degree of each node.
2. Identify all nodes with in-degree zero.
3. Use a parallel prefix sum to compact the list of active nodes.
4. Remove the active nodes and decrement the in-degrees of their neighbors.
5. Repeat steps 2–4 until no nodes with in-degree zero remain.
6. If all nodes have been processed, the graph is acyclic. Otherwise, a cycle exists.

5.2 Step 1: In-Degree Calculation

Each edge (u, v) contributes $+1$ to the in-degree of node v. On the GPU, this can be parallelized by launching one thread per edge. The main challenge is ensuring atomic updates to the in-degree array, especially when multiple threads try to update the same node concurrently.

5.3 Step 2: Node Extraction (Zero In-Degree Detection)

Once in-degrees are known, we need to identify all nodes with in-degree zero. This is done in parallel using a flag array, where each thread checks whether the in-degree of its assigned node is zero and writes a 1 (true) or 0 (false) accordingly.

5.4 Step 3: Prefix Sum (Scan)

A prefix sum (also called scan) is used to compact the flag array into a list of active node indices. This step is crucial for efficiently creating a dense list of nodes to process next.

5.5 Step 4: Graph Update (In-Degree Reduction)

For each node extracted in the previous step, we iterate over its outgoing neighbors and decrement their in-degrees by 1. Again, this step must be implemented carefully using atomic operations to avoid race conditions. Threads are typically launched per outgoing edge or per active node, depending on the representation (adjacency matrix or CSR).

5.6 Step 5: Cycle Detection

After each iteration, we keep track of how many nodes have been successfully processed. If the total number of processed nodes equals the number of nodes in the graph, then the graph is acyclic. Otherwise, one or more nodes are unreachable due to a cycle, and the algorithm terminates with a positive cycle detection.

6 CUDA Implementations and Design Choices

6.1 Adjacency Matrix Version

6.1.1 General Clarifications

- **Matrix Padding**: The adjacency matrix is padded to the next power of two. This is necessary because the scan operation implemented in this project works only with power-of-two sizes. Although this padding increases memory usage, it simplifies indexing and often improves performance due to aligned memory accesses.

- **Transposed Matrix**: Using a transposed version of the adjacency matrix would improve memory coalescing in the `calculate_in_degree` kernel. However, this introduces complications in the logic of the `remove_active_nodes` kernel, causing a performance bottleneck. Allocating both the original and transposed matrices was considered but ultimately rejected due to memory limitations on the GPU. For further details, refer to the individual kernel implementations.

- **CPU Fallback**: The task of removing and updating active nodes is executed also on the CPU. This decision was based on the observation that, in each iteration, the number of active nodes is typically very small. Executing this step on the GPU would introduce unnecessary overhead due to kernel launch latency and limited parallelism. In the experimental section there are the results.

6.1.2 Kernel: `calculate_in_degree`

The in-degree of a node in a directed graph is the number of incoming edges it receives. In the adjacency matrix representation, this corresponds to counting how many 1s appear in each column of the matrix.

In this implementation, two CUDA-based strategies were developed and evaluated: a simple direct approach and a more complex version using shared memory and partial reduction.

Naïve Kernel: `calculate_in_degree` This is a straightforward implementation where each thread is responsible for computing the in-degree of one node by scanning the corresponding column of the adjacency matrix.

- Each thread reads across all rows of a single column in the matrix.
- This results in non-coalesced memory accesses, as threads within the same warp access different rows in the same column.
- Despite its memory access inefficiency, this version is surprisingly the fastest in practice for medium-sized graphs due to its simplicity and minimal synchronization overhead.

Pseudo-code:

```
for node idx in parallel:
    in_degree[idx] = 0
    for row j in 0..n-1:
        if adj[j][idx] == 1:
            in_degree[idx] += 1
```

Shared Memory Kernel: `calculate_in_degree_partial` and `...reduce_final` This version aims to improve global memory access by:

- Dividing the computation into blocks, where each block processes one column (node).
- Using multiple blocks along the y-dimension to parallelize partial summations of the column elements.
- Accumulating partial results in shared memory and reducing them within the block.
- Finally, the results from each block are summed on the GPU in a separate reduction kernel.

Rationale:

- Global memory access patterns are more regular.
- Shared memory is used to aggregate local results, which is faster than global atomic additions.
- This version is expected to scale better with very large matrices or high degrees of sparsity.

Trade-offs:

- More complex launch configuration (2D grid).
- Requires additional global memory allocation for storing partial results.
- Performance benefits only emerge on large-scale graphs where memory access latency becomes a bottleneck.

Pseudo-code:

```
// Kernel 1: each block computes partial sum
for thread tid in block:
    for row j assigned to thread:
        if adj[j][node] == 1:
            sum[tid] += 1
reduce sum[tid] using shared memory
store result in partial[node][block]

// Kernel 2: sum all partials
for node in parallel:
    in_degree[node] = sum(partial[node][b] for all b)
```

Kernel Launch Strategy The naïve kernel was launched using:

```
<<< (Npad + lws - 1) / lws, lws >>>
```

Where `lws` is the number of threads per block and `Npad` is the padded graph size (next power of two).

The shared memory version used:

```
dim3 grid_dim(n, num_blocks_y);
dim3 block_dim(lws);
```

with `num_blocks_y` tuned based on available GPU resources.

Final Choice After benchmarking both implementations, the naïve version was selected as the default due to its superior performance on graphs of practical size (hundreds to thousands of nodes), lower memory usage, and easier integration.

Transposed Matrix Using a transposed version of the adjacency matrix greatly improved memory coalescing for this kernel, as threads accessing adjacent memory locations could read from contiguous global memory. However, this optimization introduced complications in other kernels—particularly `remove_active_nodes`—which required row-wise access. Additionally, due to GPU memory constraints, storing both the original and transposed matrices simultaneously was not feasible, leading to the decision to retain only the original layout despite the trade-off in access efficiency for this kernel.

6.1.3 Kernel: `flag_zero_in_degree`

This kernel identifies nodes with an in-degree of zero, marking them as active candidates for the next iteration of the topological sorting process. It operates in parallel, with each thread responsible for examining a single node.

Functionality Each thread checks whether the in-degree of its assigned node is zero. If so, it sets the corresponding position in the `flags` array to 1; otherwise, it sets it to 0. This `flags` array is later used in a stream compaction step to extract the list of active nodes.

```
for each node idx in parallel:
    if in_degree[idx] == 0:
        flags[idx] = 1
    else:
        flags[idx] = 0
```

Launch Configuration The kernel is launched with a 1D grid, sized to cover all padded nodes:

```
flag_zero_in_degree<<<(Npad + lws - 1) / lws, lws>>>(d_in_degree, d_flags, Npad);
```

Design Considerations This kernel is highly efficient due to its simplicity and data-parallel nature. Memory access to the `in_degree` and `flags` arrays is naturally coalesced, as each thread accesses a contiguous index.

Padding is preserved here to keep alignment with other arrays such as the adjacency matrix and to maintain consistent indexing throughout the pipeline.

Performance This kernel is not a bottleneck in the pipeline due to its low arithmetic intensity and high degree of parallelism. Profiling showed negligible execution time relative to more compute- or memory-intensive phases such as in-degree calculation and active node removal.

6.1.4 Parallel Prefix Sum (Scan)

The prefix sum (also called a scan) is a critical step in compacting the list of active nodes (i.e., those flagged with in-degree zero). A parallel implementation is necessary to preserve GPU performance and avoid CPU-GPU synchronization overhead.

Motivation After flagging nodes with in-degree zero, we need to pack these active nodes into a contiguous array. A parallel scan converts the binary flag array into an array of write indices for this compaction, enabling the construction of the next active front.

Approach The scan is implemented using the Blelloch algorithm, designed to work in parallel with exclusive scan semantics. Our implementation handles inputs of arbitrary size by padding to the nearest power of two and splitting work across multiple thread blocks.

We divide the scan into three stages:

- **Local Block Scan (Stage 1):** Each thread block performs an exclusive scan on a chunk of the input using shared memory. This is handled by the `scanKernelMultiBlock`, which stores partial block sums.
- **Block Sum Scan (Stage 2):** A separate kernel `scanKernel` computes a scan of the block sums, preparing the offset each block needs to apply to its local results.
- **Add Offsets (Stage 3):** The `addScannedBlockSums` kernel adds the scanned block offsets to each local result from Stage 1, producing the global scan.

Implementation Notes

- **Memory Padding:** The scan operates on a padded array whose length is the next power of two greater than or equal to the number of nodes, to simplify index calculations during the up-sweep and down-sweep phases.
- **Shared Memory Use:** Shared memory is used within each block to reduce latency and enable fast synchronization during prefix computation.
- **Synchronization:** Synchronization is enforced at each level of up-sweep and down-sweep to maintain correctness.
- **Two-Pass Structure:** The multi-block scan requires coordination between multiple passes to maintain a globally consistent ordering of indices.

Performance Considerations The scan is not the dominant kernel in terms of execution time, but its efficiency directly impacts how quickly new active nodes can be identified and processed. Our approach scales with the number of blocks and maintains coalesced accesses through well-aligned shared memory layouts.

Output Interpretation The final scanned output is used to populate the new active node list. The total number of active nodes is computed as:

$$\texttt{active\_count} = \texttt{output}[n-1] + \texttt{input}[n-1]$$

where `n` is the original (unpadded) input size.

6.1.5 Kernel: `compact_active_nodes`

This kernel performs the compaction step in the topological sorting pipeline. After identifying which nodes have zero in-degree (via a flag array) and performing an exclusive prefix sum (scan) on that flag array, this kernel maps the flagged nodes into a compact output array, producing the list of active nodes to process in the next iteration.

Functionality Each thread checks if its assigned node is flagged (i.e., has `flags[idx] == 1`). If so, the thread uses the corresponding value in the `scan` array to determine the index where the node should be written in the `active_nodes` output array.

```
for each node idx in parallel:
    if flags[idx] == 1:
        active_nodes[scan[idx]] = idx
```

This operation is a classic scatter-gather step, where each flagged item is written to its final compacted position determined by a prefix sum.

Launch Configuration The kernel is launched with a 1D grid, sufficient to cover the padded size of the node array:

```
compact_active_nodes<<<(Npad + lws - 1) / lws, lws>>>(d_flags, d_output, d_active_nodes, Np
```

Design Considerations This kernel leverages the results of the scan step to enable a deterministic and conflict-free write pattern. There is no need for atomic operations, since the exclusive scan guarantees unique output positions for each flagged node.

Memory access is largely coalesced:

- Reads from `flags` and `scan` are sequential across threads,
- Writes to `active_nodes` are scattered but non-overlapping.

The kernel assumes that the prefix sum has already been computed correctly and that the number of active nodes is less than or equal to `Npad`.

Performance This kernel is efficient and lightweight, with minimal computational overhead. It benefits from high parallelism and good memory throughput due to coalesced reads. The only conditional branch (checking the flag) is predictable and does not cause significant divergence.

In practice, this step contributes negligibly to the total runtime of the algorithm. Its primary importance lies in enabling efficient load balancing for the next phase, where only active nodes are processed.

6.1.6 Kernel: `remove_active_nodes`

This fundamental kernel stage removes the currently active nodes from the graph and updates the `in_degree` array by subtracting one from the in-degree of each of their neighbors. It is essential for the iterative topological sorting procedure and represents one of the most computationally intensive phases of the algorithm.

Purpose For each node marked as active (i.e., with in-degree zero), this step:

- marks the node as processed (e.g., setting its `in_degree` to `-1`),
- scans its outgoing edges to detect its neighbors,
- decrements the `in_degree` of each neighbor.

Because this phase involves potentially many updates to shared neighbors across active nodes, careful kernel design is required to ensure correctness and performance.

Host Fallback (CPU version) For small numbers of active nodes (below a threshold), the logic is executed on the host using:

```
remove_active_nodes_cpu_dense(adj, in_degree, active_nodes, active_count, n, Npad);
```

This version performs well on small inputs or when the number of active nodes is minimal, avoiding the overhead of launching CUDA kernels. It directly operates on the padded adjacency matrix and uses simple nested loops to perform the updates. I controlled the number of active nodes every cycle and they were always less than 5, this because of the way the acyclic graph is displayed.

CUDA Implementation #1: `remove_active_nodes_2D` This 2D CUDA kernel uses atomic operations to ensure correctness when multiple threads attempt to update the same destination node's in-degree. Each thread handles a pair:

- one source node from the active list,
- one destination node (matrix column).

```
dim3 block(lws2, lws2);
dim3 grid((active_count + lws2 - 1) / lws2, (n + lws2 - 1) / lws2);
remove_active_nodes_2D<<<grid, block>>>(...);
```

Drawbacks: While simple and correct, the performance is hindered by the use of many `atomicSub` operations to update the same memory locations in the `in_degree` array, which can create contention and serialization.

CUDA Implementation #2: `remove_active_nodes_2D_sm` This optimized kernel reduces global memory contention by first accumulating updates in shared memory. Threads in a block collaboratively compute partial in-degree decrements for each destination node and then commit these changes with fewer atomic operations.

```
remove_active_nodes_2D_sm<<<grid, block, sharedMemSize>>>(...);
```

Design Highlights:

- Each thread block uses shared memory to buffer per-column contributions.

- Only a single thread per destination node within a block performs the atomic update.
- This greatly reduces the number of atomic operations at the global level.

Shared Memory Layout: The kernel uses a 2D conceptual buffer in shared memory, linearized as a 1D array with size `blockDim.x * blockDim.y`, storing binary flags indicating whether each `source_node` has an outgoing edge to a `dest_node`.

Kernel Logic (Simplified)

```
for each source_node in parallel:
    for each destination_node in parallel:
        if edge exists:
            local_sum[destination_node] += 1

for each destination_node:
    if local_sum > 0:
        atomicSub(in_degree[destination_node], local_sum)

for each source_node:
    mark as processed: in_degree[source_node] = -1
```

Launch Configuration The kernel is launched with a 2D grid layout: one dimension for active nodes, one for all possible destination nodes.

```
dim3 block(lws2, lws2);
dim3 grid((active_count + lws2 - 1) / lws2, (n + lws2 - 1) / lws2);
size_t sharedMemSize = lws2 * lws2 * sizeof(int);
remove_active_nodes_2D_sm<<<grid, block, sharedMemSize>>>(...);
```

Design Considerations

- The sparse memory access pattern (adjacency lookup) limits coalescing potential.
- Shared memory significantly reduces global atomic contention.
- Setting `in_degree[source_node] = -1` is necessary to prevent reprocessing and is handled by a dedicated thread per source node.
- Careful shared memory sizing and synchronization (`__syncthreads`) are critical for correctness.

Performance Notes This phase represents one of the most time-consuming parts of the algorithm due to its high memory bandwidth demands and potential contention. Empirical results revealed the following:

- The `remove_active_nodes_2D_sm` kernel, which uses shared memory, is slightly slower than the naive global memory version in several cases.
- The CPU fallback, initially expected to perform better with a small number of active nodes, does not provide performance advantages. This is primarily due to the overhead introduced by repeated `cudaMemcpy` operations between device and host.
- Performance improvements from using shared memory are more evident with denser graphs and larger batches of active nodes per iteration.

Summary of Alternatives

- **CPU fallback**: Not effective in this representation, due to data transfer overheads outweighing the benefits of CPU-side execution.
- **Naive 2D GPU kernel**: Straightforward and works well, but suffers from global memory atomics and lacks locality optimization.
- **Optimized 2D GPU kernel with shared memory**: More efficient for dense graphs and when the active set is large; however, benefits are less pronounced for sparse or irregular topologies.

6.2 CSR Version

6.2.1 General Clarification

- **CSC for the in-degree kernel**: To improve global memory coalescing during access, the kernel responsible for calculating the `in_degree` uses the CSC (Compressed Sparse Column) representation of the adjacency matrix. Since the sparse format (CSC/CSR) is significantly more memory-efficient than a dense one, it is feasible to store both the CSR and CSC versions in memory simultaneously.
- **Why not use CSC everywhere?** Although CSC is ideal for column-wise traversal (as in the in-degree computation), it is suboptimal for removing active nodes, where we need to iterate over the *outgoing edges* of a node—that is, row-wise access. For this reason, the `remove_active_nodes` kernel uses the CSR (Compressed Sparse Row) format, which allows efficient access to all neighbors (out-edges) of a given node.
- **Shared kernels across representations**: Several parts of the pipeline remain unchanged between the dense and CSR versions. In particular:
  - The `flag_zero_in_degree` kernel (which scans for nodes with in-degree 0),
  - the parallel prefix sum (scan),
  - and the `compact_active_nodes` kernel (which performs stream compaction),

  are identical, as they operate on flat arrays and are agnostic to the underlying graph storage format.
- **CPU Fallback**: The task of removing and updating active nodes is executed also on the CPU. This decision was based on the observation that, in each iteration, the number of active nodes is typically very small. Executing this step on the GPU would introduce unnecessary overhead due to kernel launch latency and limited parallelism. In the experimental section there are the results.

6.2.2 Kernel: `calculate_in_degree`

This section presents the sparse implementations of the kernel responsible for computing the in-degree of each node. Three variants are implemented to support different data structures and performance trade-offs: one using the CSC format and two using the CSR format.

Variant 1: `calculate_in_degree_csc` This version operates on a graph stored in Compressed Sparse Column (CSC) format. Each thread computes the in-degree of a single node by inspecting the difference between consecutive elements in the `col_ptr` array, which identifies the range of incoming edges for each node.

```
for each node i in parallel:
    in_degree[i] = col_ptr[i + 1] - col_ptr[i]
```

Launch Configuration

```
calculate_in_degree_csc<<<(n + lws - 1) / lws, lws>>>(d_col_ptr, d_in_degree, n);
```

Design Considerations This kernel is the most efficient among the three. Thanks to the CSC format, each node's in-degree can be computed independently with only two memory accesses, both coalesced. This implementation avoids atomic operations entirely and is well-suited for large graphs, as long as a CSC representation is available.

Variant 2: `calculate_in_degree_csr_atomic` This variant computes in-degrees using the CSR (Compressed Sparse Row) format. Each thread processes a single node (row) and iterates over its outgoing edges, atomically incrementing the in-degree of each target node.

```
for each node i in parallel:
    for each neighbor j in row i:
        atomicAdd(&in_degree[j], 1)
```

Launch Configuration

```
calculate_in_degree_csr_atomic<<<(n + lws - 1) / lws, lws>>>(d_row_ptr, d_col_idx, d_in_deg
```

Design Considerations This kernel avoids the need for a CSC representation and operates directly on CSR data. However, its performance can degrade due to contention on `atomicAdd` operations when multiple threads attempt to increment the same in-degree entry. This may be significant in graphs with high fan-in (many incoming edges to a few nodes).

6.2.3 Kernel: `remove_active_nodes`

This section presents the sparse implementations of the kernel responsible for removing active nodes and updating the in-degree of their neighbors. Two main strategies are implemented using the CSR format to balance simplicity and performance: a direct approach using atomic operations, and an optimized two-phase method that reduces atomic contention by aggregating updates. Additionally, a CPU fallback version is used for small active frontiers to avoid inefficient GPU launches.

Variant 1: `remove_active_nodes_csr` This kernel removes active nodes and decrements the in-degrees of their neighbors directly using atomic operations. Each thread handles one active node.

Functionality For every active node, the kernel sets its in-degree to `-1` to mark it as removed. It then iterates over all outgoing edges (using CSR format) and atomically decrements the in-degrees of the destination nodes.

```
for each active node in parallel:
    in_degree[node] = -1
    for each neighbor of node:
        atomicSub(in_degree[neighbor], 1)
```

Launch Configuration

```
remove_active_nodes_csr<<<(active_count + blockSize - 1) / blockSize, blockSize>>>(...);
```

Design Considerations This version is simple and works well for small to medium workloads. However, atomic contention becomes a bottleneck when many threads attempt to decrement the in-degree of the same node, especially in high-degree graphs.

Variant 2: No Atomic

Kernel: `remove_active_nodes_csr_reduce_global` This kernel is part of a two-phase approach to reduce atomic contention. Instead of modifying the `in_degree` array directly, it builds a `d_decrement` array that accumulates how many times each node's in-degree should be decremented.

Functionality Each thread processes one active node and its outgoing edges. It atomically increments a counter in the `d_decrement` array for every destination node it finds.

```
for each active node in parallel:
    for each neighbor of node:
        atomicAdd(decrement[neighbor], 1)
```

Launch Configuration

```
remove_active_nodes_csr_reduce_global<<<(active_count + blockSize - 1) / blockSize, blockSiz
```

Design Considerations While still using atomics, this kernel avoids updating the in-degree directly. It reduces contention in the critical array (`in_degree`) and allows for more efficient sequential updates later.

Kernel: `apply_decrement_to_in_degree` This kernel is the second phase of the two-pass reduction strategy. It applies the values accumulated in `d_decrement` to the `in_degree` array.

Functionality Each thread processes one node. If a decrement is present, it subtracts it from the in-degree.

```
for each node in parallel:
    if decrement[node] > 0:
        in_degree[node] -= decrement[node]
```

Launch Configuration

```
apply_decrement_to_in_degree<<<(n + blockSize - 1) / blockSize, blockSize>>>(...);
```

Design Considerations This phase avoids atomics entirely, as each thread writes to a unique position. It benefits from coalesced memory access and offers significantly better performance under high contention.

Kernel: `mark_removed_nodes` This kernel finalizes the removal of active nodes by marking their in-degree as `-1`, signaling that they have already been processed.

Functionality Each thread sets the in-degree of one active node to `-1`.

```
for each active node in parallel:
    in_degree[node] = -1
```

Launch Configuration

```
mark_removed_nodes<<<(active_count + blockSize - 1) / blockSize, blockSize>>>(...);
```

Design Considerations This kernel is separated from the decrement phase to ensure safe writes and to eliminate atomic operations. It enables precise profiling and improves performance on large-scale graphs.

Variant 3: Function: `remove_active_nodes_cpu` This CPU-side fallback is used when the number of active nodes is below a threshold. It performs the same logic as the GPU version, using sequential updates.

Functionality For each active node, it marks the node as removed and decrements the in-degree of all its neighbors (if they haven't been removed already).

```
for each active node:
    in_degree[node] = -1
    for each neighbor of node:
        if in_degree[neighbor] != -1:
            in_degree[neighbor] -= 1
```

Design Considerations Using the CPU avoids launching small kernels with poor occupancy. The memory is copied to/from the device before and after execution. It's an effective strategy when dealing with small active frontiers.

7 Experimental Evaluation

This section presents the performance evaluation of the proposed topological sorting implementations using both dense and sparse graph representations. The goal is to assess the efficiency, scalability, and memory access behavior of different kernel variants across various graph sizes and formats.

To provide a comprehensive analysis, we compare multiple versions of key CUDA kernels implemented using both dense (adjacency matrix) and sparse (CSR/CSC) data structures. Additionally, we compare GPU-based approaches against a CPU baseline to highlight the potential speedup and trade-offs.

The primary performance metric considered is the **execution time**, measured in milliseconds (ms). This choice is motivated by the observation that, for the considered graph datasets, the GPU does not reach saturation — making execution time a more direct and reliable indicator of kernel efficiency.

Metrics. Performance is reported using the following key metrics:

- **Execution Time (ms)**: The total time spent in kernel execution, measured using CUDA events. This reflects the raw computational performance of each kernel or step.

- **Memory Bandwidth (GB/s)**: The effective memory throughput, computed as the number of bytes read and written divided by execution time. It serves as an indicator of memory subsystem efficiency.
- **Elements Processed per Second (GE/s)**: This metric reflects the number of *nodes* processed per second, normalized in billions. It is calculated by dividing the total number of nodes involved in the operation by the kernel execution time (in seconds). As such, it provides insight into the throughput of node-based operations.

Parameters After testing a lot of times, here the default values of the blocksize:

- `lws`: Local Work Size: 256
- `lws2`: Local Work Size in 2D Kernels: 16

Synthetic graphs of various sizes were used to evaluate each configuration.

7.1 Benchmark Design

7.1.1 Graph Generation and Validation

To evaluate the performance of our topological sorting algorithms, we generated a collection of synthetic graphs of controlled size and structure. The graph generation process was implemented in C and produced two sets of graphs: one in dense adjacency matrix format and one in sparse CSR format.

Each graph consists of n nodes. Two types of graphs were generated:

- **Cyclic graphs:** fully random binary matrices or CSR representations with no structural guarantees.
- **Acyclic graphs:** upper-triangular binary matrices or CSR with edges only from lower to higher indices ($j > i$), ensuring the absence of cycles.

This separation allows testing not only performance but also correctness of the topological sort—since only acyclic graphs should produce a complete topological order.

7.1.2 Input File Structure

Adjacency Format. Dense graphs were generated as $n \times n$ binary matrices, written to plain-text files with the following format:

- First line: number of nodes n.
- Next n lines: rows of the adjacency matrix, with 0s and 1s.
- Last line: a string identifying the graph type (`ciclica` or `aciclica`).

CSR Format. For CSR graphs the output format includes:

- First line: number of nodes n and number of edges (non-zero entries) nnz.
- Second line: CSR row pointer array of size $n + 1$.
- Third line: column indices array of size nnz.
- Last line: graph type (`ciclica` or `aciclica`).

The files were stored in the directories `../data/` (dense) and `../data_csr/` (CSR), with filenames of the form `sample_graphX.txt` or `sample_graph_csrX.txt` respectively.

7.2 Hardware and Configuration

All experiments were conducted on a laptop running **Microsoft Windows 11 Home**, version `10.0.26100`, equipped with an **Intel Core i7 (12th Gen)** processor (Model 154, 8 performance and efficiency cores) with a base clock of 2.3 GHz and **16 GB of RAM**. The GPU used for CUDA execution was an **NVIDIA GeForce RTX 2050 Laptop GPU** with **4 GB of GDDR6 memory**. The system used **CUDA Toolkit version 12.8** and **driver version 572.61**.

The CUDA code was compiled using **Visual Studio 2022** and the NVIDIA `nvcc` compiler (release `12.8`, build `V12.8.93`).

Timing was measured using CUDA events (`cudaEventRecord`) to accurately capture device-side execution times.

This hardware and configuration ensured reproducible and consistent measurements across all kernel variants.

7.3 Timing: Dense vs Dense (Kernel Variants)

This subsection compares the performance of different CUDA kernel variants using dense adjacency matrix representations. Two main kernels are evaluated: `calculate_in_degree` and `remove_active_nodes`. For each, both the standard global memory version and a shared memory optimized variant are tested. Additionally, for the removal kernel, a CPU version is also included as a reference.

7.3.1 Kernel: `calculate_in_degree`

The standard global memory implementation achieves higher throughput, especially on acyclic graphs. As seen in Table 1, performance drops significantly for the cyclic graph, likely due to denser connections and increased memory traffic.

Table 1: Performance comparison of `calculate_in_degree` kernel (Dense)

Graph Size	Type	Kernel	Time (ms)	GB/s / GE/s
10000	Acyclic	Global Mem	**4.815**	83.08 / 20.77
10000	Acyclic	Shared Mem	12.998	30.78 / 7.69
20000	Acyclic	Global Mem	19.393	82.51 / 20.63
20000	Acyclic	Global Mem	**17.514**	91.36 / 22.84
20000	Acyclic	Shared Mem	48.712	32.85 / 8.21
20000	Acyclic	Shared Mem	51.290	31.20 / 7.80
20000	Cyclic	Global Mem	92.873	17.23 / 4.31
20000	Cyclic	Shared Mem	**46.188**	34.64 / 8.66

Observations

- The global memory kernel consistently outperforms the shared memory version in terms of throughput (GB/s and GE/s), particularly for acyclic graphs. Notably, the global kernel approaches the memory bandwidth limits of the hardware, reaching up to 93 GB/s.
- The performance gap narrows on cyclic graphs, likely due to the denser connectivity leading to more uniform memory access patterns that benefit shared memory reuse.
- The shared memory kernel exhibits higher latency for sparse topologies, such as acyclic graphs, where its benefits are less pronounced, probably for the small number of operations.

- In general, the shared memory version suffers from overhead caused by frequent `__syncthreads()` calls, which introduce synchronization delays after each operation.

7.3.2 Kernel: `remove_active_nodes`

This kernel is extremely fast in both its global and shared memory versions. As shown in Table 2, all GPU implementations outperform the CPU variant by a large margin.

Table 2: Performance comparison of `remove_active_nodes` kernel (Dense)

Graph Size	Type	Kernel Variant	Time (ms)
10000	Acyclic	GPU Global	**0.015**
10000	Acyclic	GPU Shared	0.023
10000	Acyclic	CPU Host	0.113
20000	Acyclic	GPU Global	0.028
20000	Acyclic	GPU Global	**0.025**
20000	Acyclic	GPU Shared	0.039
20000	Acyclic	GPU Shared	0.039
20000	Acyclic	CPU Host	0.207
20000	Acyclic	CPU Host	0.228
20000	Cyclic	GPU Global	0.000
20000	Cyclic	GPU Shared	0.000
20000	Cyclic	CPU Host	0.000

Observations

- GPU-based implementations run nearly instantaneously, with sub-millisecond execution across all graph types.
- The CPU version is approximately 4–8× slower, despite the simplicity of the task.
- For the cyclic graph, the kernel effectively does no work as there are no nodes with in-degree zero—hence because of the random structure of cyclic graphs.

7.4 Timing: CSR vs CSR (Kernel Variants)

7.4.1 Kernel: `calculate_in_degree`

This section compares three implementations of the in-degree calculation using a CSR-based graph representation:

- `calculate_in_degree_csr_atomic`: uses atomic operations on global memory to update degrees.
- `calculate_in_degree_csc`: leverages a CSC-formatted representation derived from CSR to allow coalesced access and avoid atomics.

Table 3: Performance comparison of `calculate_in_degree` kernel (CSR)

Graph Size	**Type**	**Kernel**	**Time (ms)**	**GB/s / GE/s**
10000	Acyclic	`csr_atomic`	4.74	21.13 / 5.28
10000	Acyclic	`csc`	**0.80**	0.10 / 0.0125
20000	Acyclic	`csr_atomic`	41.83	9.57 / 2.39
20000	Acyclic	`csc`	**0.02**	6.51 / 0.81
20000	Cyclic	`csr_atomic`	148.55	5.39 / 1.35
20000	Cyclic	`csc`	**0.01**	14.20 / 1.78
30000	Acyclic	`csr_atomic`	164.61	5.47 / 1.37
30000	Acyclic	`csc`	**0.03**	8.68 / 1.09
40000	Cyclic	`csr_atomic`	587.52	5.45 / 1.36
40000	Cyclic	`csc`	**4.20**	0.076 / 0.009
40000	Acyclic	`csr_atomic`	326.52	4.90 / 1.23
40000	Acyclic	`csc`	**0.02**	18.25 / 2.28

Observations

- The atomic kernel (`csr_atomic`) suffers from contention on global memory writes, especially as graph density increases.
- The CSC-based kernel consistently delivers the best performance for most graphs, achieving up to 18.2 GB/s and over 2 GE/s. This is thanks to fully coalesced memory access and the absence of atomics.
- A significant degradation is seen in the CSC kernel for the 800M-edge cyclic graph, likely due to either hardware resource exhaustion or inefficient access caused by extreme edge distribution.

7.4.2 Kernel: `remove_active_nodes`

This phase is responsible for removing active nodes from the graph by updating the in-degree of their neighbors. Three approaches were compared:

- `remove_active_nodes_cpu`: executes the removal phase on the CPU if the number of active nodes is below a threshold.
- `remove_active_nodes_csr`: baseline CUDA implementation using atomic operations on global memory.
- `no_atomic` (triple kernel): optimized version without atomics, leveraging shared memory and cooperative thread processing.

Table 4: Performance comparison of `remove_active_nodes` kernel (CSR)

Graph Size	**Type**	**Kernel**	**Time (ms)**
10000	Acyclic	`remove_cpu`	**0.073**
10000	Acyclic	`csr`	0.263
10000	Acyclic	`no_atomic`	0.467
20000	Acyclic	`remove_cpu`	**0.172**
20000	Acyclic	`csr`	0.520
20000	Acyclic	`no_atomic`	0.867
20000	Cyclic	`remove_cpu`	0.000
20000	Cyclic	`csr`	0.000
20000	Cyclic	`no_atomic`	0.000
30000	Acyclic	`remove_cpu`	**0.116**
30000	Acyclic	`csr`	0.741
30000	Acyclic	`no_atomic`	1.097
40000	Cyclic	`remove_cpu`	0.000
40000	Cyclic	`csr`	0.000
40000	Cyclic	`no_atomic`	0.000
40000	Acyclic	`remove_cpu`	**0.134**
40000	Acyclic	`csr`	0.955
40000	Acyclic	`no_atomic`	1.405

Observations

- `remove_active_nodes_cpu` is the fastest method in all valid test cases, confirming that CPU fallback is efficient when the number of active nodes is small and memory transfer overhead is minimal in this type of representation.
- `remove_active_nodes_csr` offers a balanced approach, leveraging GPU parallelism but paying the price of global atomic contention.
- The `no_atomic` variant performs worse than the atomic one, likely due to the cost of synchronization and memory coordination among threads.
- For cyclic graphs, timings are reported as zero because of the random structure of the cyclic graphs.

7.5 Dense Final Version

This section presents the final implementation using the dense representation, optimized for GPU execution. It includes all kernel stages: in-degree computation, flagging zero in-degree nodes, exclusive scan, compaction of active nodes, and node removal. The kernels used are:

- `calculate_in_degree`: **Global Memory** version
- `flag_zero_in_degree`, `scan`, `compact_active_nodes`: optimized parallel versions using standard CUDA operations
- `remove_active_nodes`: **Global Memory 2D** kernel for dense matrices

Table 5: Performance summary of the final dense implementation

Graph Size	Type	In-Degree (ms)	Iterations	Total Time (ms)	Cyclic?
10000	Acyclic	10.093	5772	1910.036	No
20000	Acyclic	129.286	11597	5402.327	No
20000	Acyclic	36.139	11560	5281.660	No
20000	Cyclic	150.888	1	156.799	Yes

Table 6: Average kernel times and throughput per iteration

Kernel	Avg Time (ms)	Bandwidth (GB/s)	Ops (GOps/s)
`flag_zero_in_degree`	0.027 – 0.049	up to 3.93	up to 0.49
`scan`	0.092 – 0.194	up to 1.34	up to 0.167
`compact_active_nodes`	0.047 – 0.360	up to 3.41	up to 0.426

7.6 CSR Final Version

This section presents the final implementation using the Compressed Sparse Row (CSR) representation. This version is optimized for sparse graphs, efficiently using memory and maintaining performance across large-scale datasets. The following kernels were used:

- `calculate_in_degree_csc`: **CSC version**
- `flag_zero_in_degree`, `scan`, `compact_active_nodes`: parallel CUDA kernels adapted for sparse input
- `remove_active_nodes`: **CPU version**

Table 7: Performance summary of the final CSR implementation

Graph Size	Type	Iterations	Total Time (ms)	Cyclic?
10000	Acyclic	5758	2417.917	No
20000	Acyclic	11553	7712.533	No
20000	Acyclic	11596	5688.361	No
20000	Cyclic	1	90.878	Yes
30000	Acyclic	17386	9803.709	No
40000	Cyclic	1	592.401	Yes
40000	Acyclic	23115	13119.766	No

Table 8: Average kernel times and throughput per iteration (CSR)

Kernel	Avg Time (ms)	Bandwidth (GB/s)	Ops (GOps/s)
`flag_zero_in_degree`	0.022 – 0.057	up to 14.20	up to 1.77
`scan`	0.087 – 1.305	up to 3.00	up to 0.375
`compact_active_nodes`	0.000 – 0.059	up to 9.03	up to 0.752

7.7 Timing: Dense CUDA vs Host CPU

This subsection presents a performance comparison between the CUDA dense implementation and the sequential CPU-based host implementation. Both versions were tested on the same graph inputs, ensuring consistent conditions for a fair evaluation.

The GPU implementation uses optimized CUDA kernels for each major operation: in-degree computation, node flagging, exclusive scan, active node compaction, and node removal. In contrast, the host version follows a classic topological sort logic with a single-threaded approach.

Table 9: Dense GPU vs Host CPU Execution Time (same inputs)

Graph	Type	GPU Time (ms)	Host Time (ms)
sample_graph1 (10k nodes)	Acyclic	**1910.036**	2084.000
sample_graph2 (20k nodes)	Acyclic	**5402.327**	8973.000
sample_graph3 (20k nodes)	Acyclic	**5281.660**	9033.000
sample_graph4 (20k nodes)	Cyclic	**156.799**	10897.000

Observations

- The GPU implementation consistently outperforms the CPU version on all acyclic graphs, achieving speedups of up to **1.7x** on large graphs.
- The performance gain becomes more significant as the graph size and number of iterations increase.
- In the case of a cyclic graph, the GPU version terminates almost instantly after one iteration, correctly detecting the cycle. The CPU version still completes the detection but with significantly higher overhead.
- Despite GPU overhead from kernel launches and memory copies, the highly parallel structure of the dense algorithm compensates by handling thousands of active nodes per iteration efficiently.

Conclusion The dense CUDA implementation is clearly more scalable and better suited for large graphs with high connectivity. The host implementation, although simpler and easier to debug, does not scale well with input size and is only competitive on very small graphs.

7.8 Timing: Dense vs CSR (CUDA)

This section compares the Dense and CSR-based CUDA implementations of the topological sort. While both are designed to exploit GPU parallelism, they differ significantly in memory layout, access patterns, and performance trade-offs. The input graphs used in this comparison are not identical, so the analysis focuses on qualitative differences and trends rather than direct numerical comparisons.

Data Structures and Kernel Strategies

- **Dense Representation**: Uses an adjacency matrix of size $n \times n$, which offers fast indexing at the cost of memory overhead. Best suited for smaller or dense graphs where memory capacity is not a constraint.
- **CSR Representation**: Uses Compressed Sparse Row (CSR) format, significantly reducing memory usage for sparse graphs. Kernel execution involves more pointer dereferencing but maintains better scalability.

Table 10: Overview of CUDA Execution Time and Iteration Count

Graph	Type	Nodes	Dense Time (ms)	CSR Time (ms)
sample_graph1 / csr1	Acyclic	10000	1910.036	2417.917
sample_graph2 / csr2	Acyclic	20000	5402.327	7712.533
sample_graph3 / csr3	Acyclic	20000	5281.660	5688.361
sample_graph4 / csr4	Cyclic	20000	156.799	90.878
– / csr5	Acyclic	30000	–	9803.709
– / csr6	Cyclic	40000	–	592.401
– / csr7	Acyclic	40000	–	13119.766

Performance Summary

Observations

- **Execution Time:** For smaller graphs (10k–20k nodes), the dense implementation is competitive or slightly faster due to its simple indexing and minimal pointer overhead. However, as graph size increases, CSR scales better, while dense becomes infeasible due to memory limitations.
- **Kernel Performance:**
  - `calculate_in_degree`: The dense kernel exhibits very high throughput (up to 44 GB/s), while the CSR version varies significantly depending on graph sparsity and access locality.
  - `remove_active_nodes`: This kernel is slower in the CSR implementation (e.g., up to 0.23 ms per iteration), mainly due to irregular memory access patterns and pointer-based traversal. To mitigate this, a CPU fallback was introduced in the CSR version for certain cases. In contrast, the final Dense implementation retains a fully GPU-based kernel, which remains efficient in practice due to coalesced memory access and shared-memory optimizations.
- **Scalability:** CSR supports graphs with up to 40,000 nodes and hundreds of millions of edges, while the dense representation is limited by quadratic memory complexity.

8 Conclusion

Throughout this project, the hardware limitations played a significant role in shaping design choices. The GPU available was not particularly powerful, and the nature of the task—especially operations like in-degree calculation and active node management—made heavy use of memory bandwidth. Because of this, rather than focusing solely on raw performance, the project aimed to explore and compare multiple implementation strategies to understand the trade-offs involved.

Three main approaches were developed and tested: a Host (CPU-based) implementation, a Dense adjacency matrix GPU version, and a CSR (Compressed Sparse Row) GPU version. Each of these representations came with its own strengths and challenges, and a good part of the effort went into optimizing their specific bottlenecks.

For the dense matrix implementation, optimizations included padding for scan efficiency, experimenting with memory coalescing through transposed matrices, and the use of shared memory in kernels. Meanwhile, the CSR version required different strategies due to its irregular memory patterns, such as parallel traversal of sparse structures and avoiding warp divergence. In both cases, the computation of the in-degrees and the removal of active nodes were areas of particular focus, as they were called thousands of times during execution and directly influenced iteration speed.

The host version, while conceptually straightforward, performed significantly worse on larger graphs, making it unsuitable for real-world use beyond prototyping or validation. In contrast, the dense representation showed competitive performance on small-to-medium graphs, especially when the graph was dense enough to justify its memory footprint. The CSR version, although more complex, scaled better with graph size and sparsity—making it more versatile overall.

From these experiments, some key takeaways emerged. Dense matrices benefit from simplicity and straightforward parallelization, but become inefficient in memory usage as graph size grows. CSR is more memory-efficient and performs better on sparse graphs, though it requires more careful handling to achieve good performance. Additionally, kernel optimizations like shared memory usage, and decisions such as CPU fallbacks, need to be evaluated on a case-by-case basis—what helps in one representation may hurt in another.

Lastly, although the implementations were functional and performed reasonably well, there is room for future improvement. Exploring even more scalable solutions such as multi-GPU execution, evaluating high-level libraries, or improving load balancing in CSR kernels could further enhance performance and robustness.