Parallel Topological Sorting

Ezra Fu
erzhengf@andrew.cmu.edu

Robert Benjamin Osborne
rbo@andrew.cmu.edu

https://codeplay0314.com/parallel-topological-sorting/

Project Report for Carnegie Mellon University 15-618: Parallel Computer Architecture and Programming, Fall 2024

Abstract

In this project, we explored innovative ways, serial on CPU and CUDA and DSL on GPU, to accelerate topological sorting of directed acyclic graphs through parallel computing for efficient implementation and conducted comprehensive experiments and analysis.

1 Introduction

This project focuses on improving the performance of topological sorting. Later in this section we introduce the definition of the problem and our motivation. After reviewing existing research on this well-studied problem (see Section 2), we highlight the significance of efficient solutions and emphasize the potential of parallel computing to further enhance performance. We then present three distinct approaches: a traditional serial algorithm, a CUDA-based parallel implementation, and a pair of domain-specific language approaches using NetworkX and Boost Graph (see Section 3). Next, we describe the datasets used for evaluation, including both randomly generated graphs and real-world examples (see Section 4). Our experimental results, including correctness validation and performance benchmarking, are reported in Section 5. We conclude our study and summarize key findings in Section 6, as well as additional insights, potential improvements, and future research directions. Appendix A provides a breakdown of the contributions of each team member. The source code for this project is available on GitHub[1].

1.1 Problem Definition

With a Directed Acyclic Graph (DAG) as the input, our goal is to generate depth layers of nodes. Each depth layer (excluding the first) is composed of nodes that strictly depend on the nodes from the previous depth layer.

Input Format: The input describes a DAG:

- The first line contains an integer n representing the number of nodes in the DAG.
- The following n lines describe the dependencies of each node i (where i ranges from 1 to n):
  - Line i starts with an integer d_i, the number of dependencies of node i.

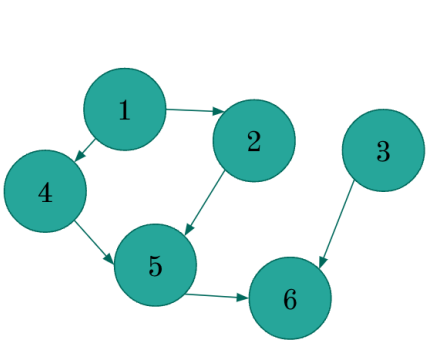
[1] https://github.com/codeplay0314/parallel-topological-sorting/

Table 1: Example Input (example.in)

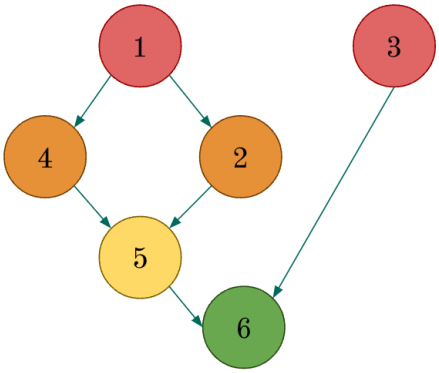
6
0
1 1
0
1 1
2 2 4
2 3 5

Table 2: Example Output (example.out)

4
2 1 3
2 2 4
1 5
1 6



(a) A sample DAG.



(b) The same DAG with nodes grouped by depth.

Figure 1: An example DAG and its node partitioning into depth layers.

  - Following d_i are d_i integers, each representing a node that i depends on.

It is guaranteed that the nodes and their dependencies form a valid DAG.

Output Format: The output describes the depth layers of the DAG:

- The first line contains an integer b indicating the number of depth layers.
- The following b lines describe each depth layer:
  - Line i starts with an integer k_i, the number of nodes in the i-th depth layer.
  - Following k_i are k_i integers representing the nodes in that depth layer.

As shown in Table 1 and Table 2, our input format and the corresponding output layers are clearly illustrated with a simple 6-node DAG. In particular, the input DAG structure (see Figure 1a) provides the relationships between nodes, while the depth-layer partitioning (Figure 1b) visually demonstrates how nodes are grouped according to their dependencies.

1.2 Motivation

This project work was initially inspired by our efforts in a previous course assignment[2], where we were tasked on rendering circles for an image. We attempted to determine the dependencies between circles using topological sorting, then form "batches" of circles that could be rendered concurrently. To accelerate this process, we employed a CUDA-based approach, aiming to leverage parallelism.

[2]https://github.com/cmu15418f24/asst2/blob/main/asst2_handout.pdf

However, we ultimately decided not to adopt this technique for the circle-rendering scenario. The primary reason is that the dependency graph for the circles tends to be quite deep, resulting in a large number of levels. Such a structure reduced the effectiveness of parallelization, and the CUDA solution did not provide the expected performance improvements under these conditions. This experience, however, guided our understanding and informed the current exploration of more effective parallel topological sorting techniques under different circumstances.

2 Background

2.1 Topological Sorting

Topological sorting is a fundamental operation for ordering elements that have defined precedence constraints, making it significant in a wide range of applications such as scheduling and dependency resolution. Its importance has grown with deep learning, where the order of computational steps can significantly impact efficiency and throughput.

Foundational work in topological sorting, such as Kahn's classic BFS-based algorithm [4], has established a baseline for both clarity and performance. More recent research has explored ways to leverage parallel computing platforms, such as GPUs, to further enhance performance. For instance, parallelizing BFS on CUDA has shown considerable promise in accelerating similar graph-based computations [3]. Additionally, comparative studies between CPU-based and GPU-based implementations have demonstrated the nuanced trade-offs in performance and resource utilization [9].

2.2 Directed Acyclic Graph

A Directed Acyclic Graph (DAG) is a directed graph with no cycles. Due to this property, DAGs serve as a natural representation of precedence constraints. The absence of cycles guarantees that topological sorting can be applied, providing a linear ordering of nodes that respects all dependencies.

2.3 CUDA

CUDA, developed by NVIDIA, is a parallel computing platform and programming model that allows researchers and engineers to harness the power of GPUs for general-purpose computation [6]. CUDA can accelerate algorithms such as parallel BFS, contributing directly to performance improvements with multiple GPU threads in topological sorting.

2.4 Domain-Specific Languages

Domain-Specific Languages (DSLs) offer a powerful, quick and performant method of implementing algorithms related to a specific domain. Our team implemented two DSLs for this project, including both NetworkX [2] and Boost Graph [8]. Both solutions provide developers with a flexible graph structure and builtin methods to simplify developer work.

3 Methods

3.1 Serial Implementation

Our serial topological sorting approach is based on Kahn's algorithm [4], implemented in C++. This algorithm operates by maintaining and updating an in-degree array for each

node, systematically selecting those with zero in-degrees and removing their influence until all nodes are processed.

3.1.1 Kahn's Algorithm Pseudocode

Algorithm 1 Kahn's Algorithm for Batch-Based Topological Sorting

Require: Directed Acyclic Graph (DAG) with n nodes.
Ensure: Batch-wise topological order of the nodes.

```
Store the graph structure in an adjacency list adj[] and an in-degree array indeg[].
Initialize an empty list batches.
Let batch = { all nodes v where indeg[v] = 0 }.
while batch is not empty do
    Append batch to batches.
    Initialize next_batch = [].
    for each node i in batch do
        for each neighbor j in adj[i] do
            indeg[j] ← indeg[j] - 1
            if indeg[j] = 0 then
                Add j to next_batch.
            end if
        end for
    end for
    batch = next_batch.
end while
```

3.1.2 Key Data Structures

- **Adjacency List (`adj`)**: A vector of vectors storing the outgoing edges for each node.
- **In-Degree Array (`indeg`)**: A vector that tracks the number of incoming edges.
- **Batch Queue (`batch`)**: A temporary vector used to hold nodes with zero in-degrees.

3.1.3 Key Operations

A key operation involves decrementing the in-degree of neighboring nodes as each node is processed. When a node's in-degree reaches zero, it is added to the next batch, ensuring that all dependencies are satisfied before it is processed.

3.1.4 Complexity Analysis

Time Complexity: The overall time complexity is $\mathcal{O}(n + m)$, where n is the number of nodes and m is the number of edges. This complexity accounts for reading the graph and processing all nodes and edges during the topological sort.

Space Complexity: The space complexity is $\mathcal{O}(n + m)$, arising from the storage requirements of the adjacency list, in-degree array, and intermediate batch structures.

3.2 CUDA Implementation

Our CUDA-based implementation aims to accelerate the topological sorting process by leveraging the parallelism of GPUs. While the serial approach is efficient in a single-threaded environment, it can be parallelized to better handle large graphs with high connectivity.

3.2.1 Key Data Structures

We maintain several key data structures on the GPU to facilitate parallel computation:

- **In-Degree Array:** An integer array `d_inDegrees` that holds the current in-degree.
- **Adjacency Representation (CSR):** A Compressed Sparse Row (CSR) format using:
  - `d_rowPtr`: An array indicating the start and end indices of nodes's adjacency list.
  - `d_edges`: A contiguous array listing the targets of edges for all nodes.

  This structure ensures coalesced memory accesses when retrieving neighbors.
- **Flags Array:** A boolean-like array `flags` on the device to mark zero in-degree nodes.

3.2.2 Key Operations

- **Scattering Zero-Degree Nodes:** Using a prefix sum (scan) and a functor, we pack the zero-degree nodes into a contiguous list for processing.
- **Processing Nodes and Reducing In-Degrees:** For each zero-degree node, we iterate over its adjacency list and decrement the in-degree of its neighbors atomically.

3.2.3 Parallelization and Computational Bottlenecks

The computationally expensive part of the algorithm is the repeated scanning and updating of in-degrees, especially in graphs with many edges. This includes:

- Identifying zero in-degree nodes each iteration.
- Decrementing in-degrees of a large number of neighbors.

These operations are data-parallel: each node (and its adjacency list) can be processed independently. The key dependencies lie in the correctness of in-degree updates—no node can appear in the next batch until all its incoming edges are processed. However, since each node's neighbors are updated atomically and independently, we can exploit a high degree of parallelism across nodes.

3.2.4 Evolution of the Approach

Our implementation underwent several iterations before arriving at the current solution. Initially, we experimented with using an adjacency matrix to represent the graph. This approach avoided the need for atomic operations since updates to in-degrees could be made by scanning entire rows or columns. However, this design severely impacted performance and scalability: each iteration required $\mathcal{O}(n^2)$ operations rather than $\mathcal{O}(m)$, since we repeatedly processed every entry of the $n \times n$ matrix, regardless of how many edges were actually present.

Furthermore, representing large graphs using an adjacency matrix quickly led to prohibitive memory demands. For instance, when dealing with more than 100,000 nodes, the

matrix became extremely large, requiring 40 gigabytes of GPU memory. This not only degraded performance but also caused memory allocation failures and crashes.

To address these issues, we transitioned to a more memory-efficient representation using the Compressed Sparse Row (CSR) format. This shift reduced memory usage, ensured that we only processed existing edges, and allowed us to exploit data parallelism more effectively. Combined with atomic operations for in-degree updates, our refined approach significantly improved both performance and scalability.

3.3 DSL

Our DSL based approaches use both Khan's Algorithm and builtin topological sorting methods. Graph DSLs will create a graph structure that includes many nodes (vertices) and dependencies (edges) with much higher overhead than the serial approach, but allows for much more flexible developer querying. The two DSLs used as part of this effort are NetworkX and Graph Boost.

3.3.1 Key Data Structures

Both DSLs tested used the following data structures to maintain and work with the DAG:

- **Directional or Bidirectional Graph:** A type implemented by the DSL that contains all information in the graph and DAG. Directions enable "depends on" relationships.
- **Vertex:** A class that enables quick lookup and deletion of specific nodes or vertices.
- **Edge:** A structure that marks a dependency on another node. Please note that edges are always directed in our testing.

3.3.2 Key Operations

Both DSL implementations include two main methods of calculating topological order:

Khan's Algorithm:

- **Calculating Out Degree of All Nodes:** First, the out degree of every node is calculated. Nodes with 0 out degree are added to the next topological batch and marked for deletion. Please note this is a read-only operation and may be performed concurrently.
- **Removing O-degree Nodes:** All 0 degree nodes and their in dependencies will then be removed from the graph. This step includes constant writes to the underlying graph structure.

Builtin sorting: Both NetworkX and Boost include built-in functions to apply topological sorting. Although these generally offer the best performance, they do not allow adding parallelization or standardizing output without additional processing.

3.3.3 Parallelization and Computational Bottlenecks

Several different strategies were attempted to parallelize the DSLs computations. When Khan's algorithm is used, the main program's while loop is divided into two main parts: finding all nodes with no dependencies (or an out-degree of 0), then removing all these from the Graph. The first step may be done in parallel, but the last one not, since concurrent writes will corrupt the underlying data structure.

To provide custom parallelization requires implementing our own algorithm (like Kahn's) outside the library. We started with NetworkX so, this involved iterating through data structures in Python3 space, which is significantly slower than C++ and will likely negate any potential benefits to parallelization. To mitigate this issue, our team turned to the C++ Boost Graph library. This enabled the read-only portion of Khan's Algorithm to be parallelized with OpenMP. The most significant bottleneck we observed is the clear and remove vertex process, as it is slow and may not be performed concurrently.

3.3.4 Evolution of the Approach

A number of different approaches were implemented in order to both improve performance add parallelism. Our team starting by researching various DSLs including GraphLab (Turi Create) and atypical solutions like Apache Spark. We settled on NetworkX for two main reasons: it is a popular and well-supported library and it has the ability to support highly parallelizable back ends. One such example is the cuGraph which may improve performance by several orders of magnitude depending on the workload[7]. However, trying this back end as well as several others did not result in any noticeable improvement. Since our Kahn implementation involved working with Python directly, we came to the conclusion that we were not able to achieve any further meaningful speedup with a Python DSL and considered alternatives. We settled on Boost Graph, which has similar data structures but will allow us to write our own algorithms in performant C++ code.

4 Dataset

We evaluated our approach using four categories of datasets: Node, Edge, Depth, and Real-World Test. Each category highlights a particular aspect of performance and scalability.

4.1 Node Test

The Node Test dataset is designed to measure how performance scales with increasing numbers of nodes. For these experiments, we fixed the number of edges at 10,000 and varied the node count from 10^3 to 10^7. As the number of nodes grows, the depth of the graph (defined as the number of layers in the topological order) decreases from around 300 down to 4. This dataset is generated randomly: given a specified number of nodes and edges, we repeatedly choose two distinct nodes at random and form a dependency from the node with the larger index to the node with the smaller index. This ensures acyclicity by preventing edges from pointing "forward" in index order.

4.2 Edge Test

The Edge Test dataset explores how the algorithm scales with an increasing number of edges which stresses the performance of our approach when the graph becomes denser. We fix the number of nodes at 1 million and vary the number of edges from 10^3 up to 10^7. As the number of edges increases, the depth of the graph increases from about 3 to 60 layers. The generation method is same as that of the Node Test.

4.3 Depth Test

For the Depth Test, we focus on controlling the depth of the graph. We fix the node count at 10^5 and the edge count at 2×10^5, while varying the depth from 10^0 (i.e., no dependencies) up to 10^5 (a chain). We first assign each node a depth level randomly. Then, for each node,

Table 3: Summary of Real-World Datasets

Name	Nodes	Edges	Depth	Category & Description
soc-Epinions1	75,888	405,740	6,972	Who-trusts-whom (Epinions.com)
amazon0302	262,111	899,792	63,792	Product co-purchasing (Amazon, Mar 2 '03)
web-Stanford	281,904	1,992,636	11,135	Web graph of Stanford.edu
web-Google	916,428	4,322,051	65,668	Web graph from Google
wiki-Talk	2,394,385	4,659,565	15,580	Wikipedia talk (communication) network

we introduce edges from that node to nodes in the previous depth layer, ensuring that edges always flow from deeper to shallower depths and thus remain acyclic. If additional edges are needed after meeting the initial depth requirement, we generate them randomly while preserving the acyclicity rule. This dataset evaluates how our method performs when the graph's topology varies dramatically in terms of layering.

4.4 Real-World Test

To complement the synthetic datasets, we also evaluate our method on real-world graphs sourced from the Stanford SNAP library [5]. Most real-world networks contain cycles, so we processed them using GraphViz's `acyclic` tool to remove cycles. We then constructed standard input and output files (`.in` and `.out`) for each resulting DAG. Table 3 summarizes the five real-world datasets used. These datasets represent diverse domains, including product co-purchasing networks, web graphs, social networks, and communication networks.

5 Results

5.1 Experiment Setup

All tests were conducted on the Pittsburgh Supercomputing Center (PSC) machine. Experiments were run on Bridges-2 nodes equipped with dual-socket AMD EPYC 7742 64-core processors (128 CPU cores in total) and NVIDIA A100 GPUs.

5.2 Correctness

To ensure the correctness of our implementations, we conducted comprehensive verification by comparing their outputs against known, standardized results. We adopted a novel test generation method, as described in Section 4.3, where we first constructed a correct solution and then generated the input data that would produce these answers. This reverse-engineering approach guaranteed that we had a reliable ground truth without the need for an independently verified topological sorter.

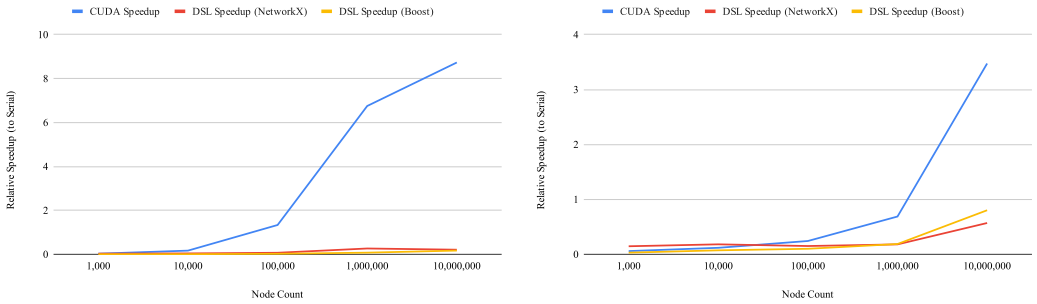
Using this technique, we produced a diverse set of test cases covering various scales and edge conditions. We then ran our serial, CUDA, and DSL implementations on these inputs and confirmed that all generated outputs matched the expected solutions. These experiments provided strong evidence of the correctness and robustness of our approaches under a wide range of scenarios.

5.3 Performance

We evaluated the performance of our Serial, CUDA, and DSL implementations across all four dataset categories (Node Test, Edge Test, Depth Test, and Real-World Test). For each implementation, we measured both the raw computation time and the total execution time,

Table 4: Computation Time (ms) by Varying Node Counts

Nodes	Edges	Depth	Serial	CUDA	NetworkX	Boost
10^3	10^5	301	1	36	34	372
10^4	10^5	54	1	6	25	245
10^5	10^5	12	8	6	116	369
10^6	10^5	5	54	8	205	732
10^7	10^5	4	454	52	2,178	2,701



(a) Computation Time

(b) Total Time

Figure 2: Node Test Speedup

including initialization and data transfer. We then calculated and plotted the computational and total speedups of the CUDA and DSL approaches relative to the serial baseline. These comparisons allowed us to identify the scalability benefits and efficiency gains of parallelization and DSL optimizations under varying graph sizes, densities, and depth distributions.

5.3.1 Node Test

The Node Test evaluates how the algorithms scale with increasing numbers of nodes while holding the number of edges fixed. As the node count grows, the depth of the graph decreases. Table 4 summarizes the computation times (in milliseconds) for different node sizes.

We observe varying performance outcomes for each approach, notable trends include:

- **[Figure 2a] CUDA is slow with less nodes, but increases significantly as more nodes are added:** CUDA parallelizes: finding zero in-degree nodes, updating their processed status, and decrementing the in-degrees of their neighbors.
- **[Figure 2b] DSL speed is slow to start, but slightly increases with more nodes:** Both DSLs keeps a Graph data structure to store all graph information, which is heaver weight than the Serial implementation's simple vector structure.

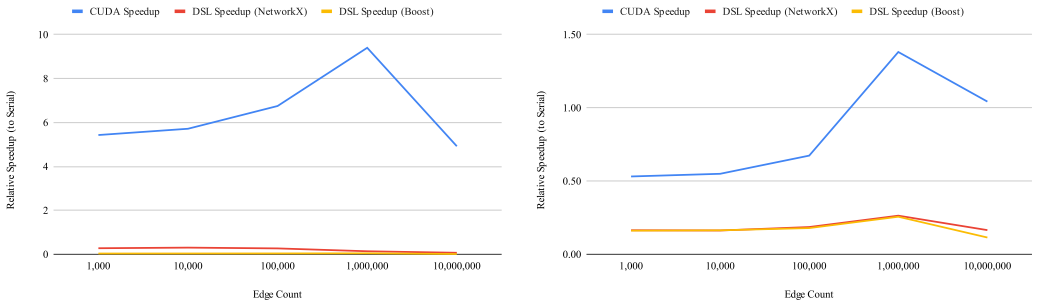
5.3.2 Edge Test

The Edge Test determines how well the algorithms handle more edges as the node count remains constant. Edges indicate dependencies on other nodes, and since node count remains constant, the average depth count will increase slightly with greater edge counts.

Notable Trends:

Table 5: Computation Time (ms) by Varying Edge Counts

Edges	Nodes	Depth	Serial Time	CUDA Time	NetworkX Time	Boost Time
10^4	10^6	3	38	7	138	604
10^5	10^6	3	40	7	132	619
10^6	10^6	5	54	8	202	800
10^7	10^6	13	188	20	1,366	2,696
10^8	10^6	60	364	74	5,316	21,343



(a) Computation Time

(b) Total Time

Figure 3: Edge Test Speedup

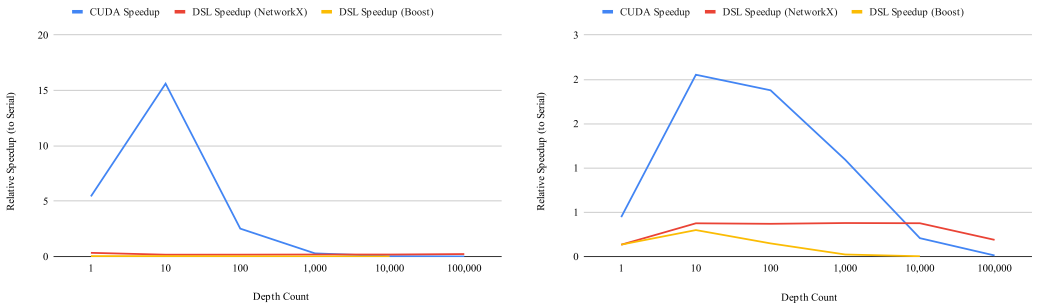
- **[Figure 3b] CUDA generally performs better as more edges are added:** CUDA parellelizes a number of steps in the sorting process. Datasets with more edges allow parallelization and more speedup as a result.
- **[Figure 3b] Relative CUDA performance drops from 10^6 to 10^7 edge counts** As part of the edge (dependency) count increasing, so does the depth count increase. More depth results in more kernel calls, slowing down the program.
- **[Figure 3b] NetworkX performance slightly drops as more edges are added:** NetworkX manages a more complicated directed graph data structure instead of a simple vector of integers. There is more variation with the serial implementation, but overall compute speedups should stay somewhat constant.

5.3.3 Depth Test

The Depth Test determines how well the algorithms are able to handle datasets with many layers of dependencies. Both nodes and edges will remain constant across all depth tests.

Table 6: Computation Time (ms) by Varying Average Depth Counts

Depth	Nodes	Edges	Serial Time	CUDA Time	NetworkX Time	Boost Time
1	10^6	2×10^5	38	7	115	584
10^1	10^6	2×10^5	281	18	1,722	5,406
10^2	10^6	2×10^5	297	118	1,795	13,906
10^3	10^6	2×10^5	315	1,119	1,770	109,150
10^4	10^6	2×10^5	319	11,030	1,803	937,414
10^5	10^6	2×10^5	425	110,190	1,935	DNF



(a) Computation Time (b) Total Time

Figure 4: Depth Test Speedup

Table 7: Computation Time (ms) with Different Real-World Datasets

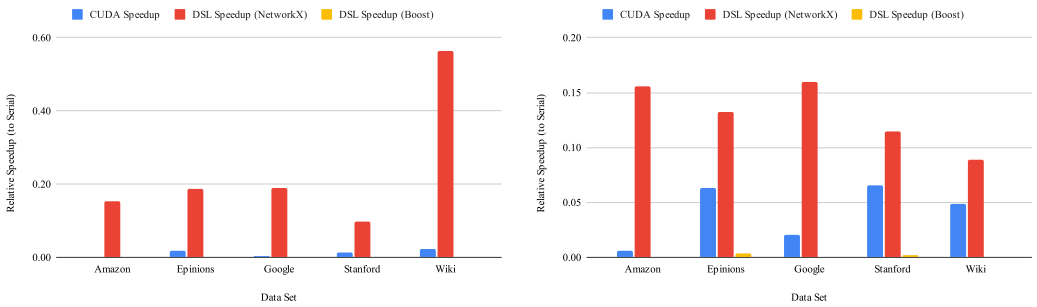
Dataset	Nodes	Edges	Depth	Serial	CUDA	NetworkX	Boost
soc-Epinions1	75,888	405,740	6,972	14	790	75	18,102
amazon0302	262,111	899,792	63,792	85	43,313	551	1,455,361
web-Stanford	281,904	1,992,636	11,135	290	64,632	1,538	DNF
web-Google	916,428	4,322,051	65,668	66	4,618	673	173,074
wiki-Talk	2,394,385	4,659,565	15,580	395	17,523	700	7,132

Notable Trends:

- **[Figure 4a] CUDA performance starts high, but rapidly drops as depth increases:** Every "depth" requires another layer and iteration. This includes several kernel calls and becomes significant when depth increases exponentially.
- **[Figure 4a] NetworkX performance stays relatively linear as depth increases:** The algorithm used in the NetworkX implementation is single threaded, so it is expected that it does not see any gains from parallelism on larger workloads.

5.3.4 Real-World Test

The real world tests apply the algorithms to large datasets of real-world user actions and behaviors. Node, edge and depth counts will all vary greatly depending on the dataset.



(a) Computation Time (b) Total Time

Figure 5: Real World Test Speedup

Notable Data points:

- **[Figure 5b Boost did not finish a test and was very slow for others:** Similar to the CUDA implementation, the Boost implementation struggles with higher depth datasets. This is due to both a recalculation of all vertices in the graph for every layer, and due to the high cost of clearing and removing vertices.
- **[Figure 5b Speedups were lower overall here:** The higher depths of these real-world datasets negatively impacted CUDA and Boost performance. This brings up an important consideration, that one must consider the dataset before deciding if these algorithms will improve their performance.

5.4 Universal Trends

Notable Trends:

- **[e.g. Figure 3b] DSL performance is poor - 0.15x - 0.4x serial implementation:** The serial application runs simple, highly performant loop that uses simple data structures. In contrast, DSLs support much more (relatively) powerful data structures like graphs, vertices and nodes. For large and complex work items, this will allow much more developer flexibility, providing them with powerful builtin methods. For simpler datasets however, the DSLs will incur a cost with the added complexity. Until greater parallelization is achieved, the DSLs may not be able to surpass the serial performance.
- **[Figure 3a vs 3b] CUDA initialization times are costly:** CUDA requires the dependency data to be loaded into several int arrays that enables the GPU to work in parallel. Furthermore, the data must be copied to the device. This critique is true for the DSL implementations as well, in part since they use and configure more complicated data structures than the simple int vector the serial solution uses.

6 Conclusion and Discussion

Our evaluation has demonstrated that the serial implementation provides a robust and adaptive baseline. Its execution time grows almost linearly as the number of nodes, edges, and graph depth increase, making it stable and predictable even for large real-world datasets.

Both NetworkX and Boost Graph were excellent to work with, although developers must consider if the extra weight of the library is worthwhile over a simple serial approach. NetworkX's Python interfaces makes it exceedingly quick and easy to get started writing graph algorithms. Boost Graph however seems to preferable DSL to use if speed and granularity are desired. In the future, two alternative approaches may be helpful to further improving Graph Boost's parallelization. The first is to implement a library like Boost.MPI, which uses MPI to distribute work across many servers [1]. The other is to explore using a breadth-first search (BFS), which may be performed in parallel, on remaining nodes as an alternative to clearing and removing vertices.

The CUDA-based approach shows significant promise, particularly as graph size grows in terms of node or edge count. By parallelizing key operations, we achieved substantial speedups over the serial implementation for large, wide graphs. For the largest synthetic tests, CUDA was able to leverage data parallelism effectively, distributing work across many GPU cores to accelerate the topological sort. However, the CUDA method does not scale as efficiently when graph depth is large. In these scenarios, the algorithm's approach—identifying and processing zero in-degree nodes layer by layer—leads to an increasing number of kernel

launches and additional per-layer overhead. Even though the total number of nodes and edges may remain constant, a deeper graph structure requires more sequential steps, limiting the potential speedup. This limitation became evident in real-world datasets, such as large social or web graphs, which often have substantial depth. Several factors limit the speedup are:

- **Depth-Induced Dependencies:** Increasing depth leads to more sequential GPU kernel launches, reducing effective parallelism.
- **Atomic Operations and Synchronization Overheads** Updating in-degrees often involves atomic operations, which can diminish parallel efficiency.
- **Data Transfers and Memory Bound Operations** Large data structures and irregular memory accesses can limit throughput on the GPU.

Future work will aim to bring the benefits of parallel topological sorting to a broader range of real-world applications. We can focus on exploring hybrid CPU-GPU strategies for uneven workloads, or investigate domain-specific optimizations, such as improved memory layouts and load balancing techniques, to achieve more consistent speedups.

References

[1] Douglas Gregor and Matthias Troyer. "Boost. mpi". In: *MPI, November* (2006).

[2] Aric A. Hagberg, Daniel A. Schult, and Pieter J. Swart. "Exploring Network Structure, Dynamics, and Function using NetworkX". In: *Proceedings of the 7th Python in Science Conference*. Ed. by Gaël Varoquaux, Travis Vaught, and Jarrod Millman. Pasadena, CA USA, 2008, pp. 11–15.

[3] Pawan Harish and Petter J Narayanan. "Accelerating large graph algorithms on the GPU using CUDA". In: *International conference on high-performance computing*. Springer. 2007, pp. 197–208.

[4] Arthur B Kahn. "Topological sorting of large networks". In: *Communications of the ACM* 5.11 (1962), pp. 558–562.

[5] Jure Leskovec and Rok Sosič. "Snap: A general-purpose network analysis and graph-mining library". In: *ACM Transactions on Intelligent Systems and Technology (TIST)* 8.1 (2016), pp. 1–20.

[6] NVIDIA Corporation. *CUDA Toolkit Documentation*. Accessed: 2024-12-15. 2024. URL: `https://developer.nvidia.com/cuda-toolkit`.

[7] Rick Ratzel and Jenn Yonemitsu. *NetworkX introduces zero code change acceleration using Nvidia Cugraph*. Oct. 2024. URL: `https://developer.nvidia.com/blog/networkx-introduces-zero-code-change-acceleration-using-nvidia-cugraph/`.

[8] Jeremy Siek, Lie-Quan Lee, and Andrew Lumsdaine. *The Boost Graph Library (BGL)*. URL: `https://www.boost.org/doc/libs/1_87_0/libs/graph/doc/index.html`.

[9] David Svantesson and Martin Eklund. *A naive implementation of Topological Sort on GPU: A comparative study between CPU and GPU performance*. 2016.

Appendix

A List of Work by Each Student

The overall workload was distributed evenly between the two students, with each contributing approximately 50%.

A.1 Ezra Fu

- Conducted literature review on topological sorting.
- Implemented the standard sequential topological sort.
- Developed the CUDA-based topological sort.
- Validated correctness using sample DAGs and established baseline performance metrics.
- Generated test data and converted real-world datasets.
- Wrote automated test scripts.
- Contributed to drafting sections of the proposal, milestone report, and final report.
- Collected portions of the experimental data.
- Built and set up the project website.
- Prepared materials for the poster session.

A.2 Robert Benjamin Osborne

- Researched and chose both DSLs.
- Implemented and optimized both DSLs.
- Generated "exponential" test data (unused).
- Contributed to the milestone and final report.
- Wrote test scripts for running on PSC machines.
- Performed testing and gathered final results.
- Assisted in poster session presentation.