# PARALLEL TOPOLOGICAL SORT FOR COMPUTATIONAL GRAPH

**Yizhou Wang** **Andy Tang**
{yizhouw3, andyt}@andrew.cmu.edu

## 1 ABSTRACT

We present a parallel framework for executing computations on directed acyclic graphs (DAGs) using topological sorting. Such graphs naturally represent complex workflows with nontrivial dependencies, but sequential execution often underutilizes modern hardware. To address this limitation, we design parallel approaches that expose concurrency by identifying and processing nodes whose dependencies are resolved. Our implementations leverage both CPU and GPU architectures, using OpenMP for multicore parallelism and CUDA for massively parallel execution.

Both approaches follow a frontier-based strategy inspired by Kahn's algorithm, enabling dynamic scheduling and efficient utilization of computational resources. We evaluate performance across a range of graph structures, varying graph depth, width, sparsity, and regularity. We identify performance bottlenecks by measuring scalability, synchronization overhead, and load balancing. The results show substantial speedups over sequential baselines, with the GPU implementation delivering the strongest speedup (close to 40x) on highly parallel workloads and the CPU implementation achieving high efficiency (10x on 32 cores). Overall, this work demonstrates that parallel topological sorting is an effective and practical technique for accelerating dependency-driven computations across heterogeneous systems.

## 2 BACKGROUND

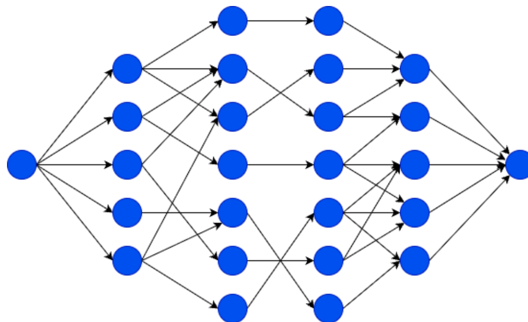

Figure 1: Example computation graph.

The application accelerates execution of an arbitrary computational graph, represented as a directed acyclic graph (DAG) where each node corresponds to a computation and edges represent data dependencies as in Figure 1. Unlike simple pipelines, this graph can have complex branching and merging structure, capturing general-purpose computations such as scientific workflows, compiler intermediate representations, or differentiable programs.

The topological sort algorithm performs the computation while respecting data dependencies. In brief, the algorithm maintains a set of "ready" nodes with no unmet dependencies, processes them, and updates dependent nodes until the entire graph is evaluated. An example sequential implementation is shown below.

```
void topo_compute(int n, vector<vector<int>> &adj, vector<int>
 ↪  &value) {
```

```
    queue<int> q;
    vector<int> indeg(n);

    while (!q.empty()) {
        int u = q.front();
        q.pop();
        for (int v: adj[u]) {
            value[v] += value[u];
            if (--indeg[v] == 0) {
                q.push(v);
            }
        }
    }
}
```

The application can benefit from parallelism because the DAG defines only a partial order, not a total order. At any point in time, multiple nodes may have all dependencies satisfied and can therefore execute simultaneously. This creates a dynamic "frontier" of ready nodes that can be processed in parallel. In addition, parallelism exists across the outgoing edges of each ready node, where value propagation and indegree updates to different neighbors can be performed concurrently.

Nonetheless, parallelizing the algorithm is challenging due to complex data dependencies and massive contention to shared data structures. As mentioned above, the computational graphs in this application have complex dependencies. A node cannot execute until all of its predecessors have completed, which limits parallelism along the graph's critical path and can lead to load imbalance in irregular graphs. Furthermore, frontier sizes can vary significantly, and differences in average outdegree can either limit or amplify the available parallelism.

Shared data structures further introduce contention. For example, the ready queue (q) used in the sequential implementation becomes a bottleneck when multiple threads attempt to push and pop nodes simultaneously. Similarly, updating the in-degree array (in_deg[v]) requires synchronization (e.g., atomic decrement operations), and many threads may contend when multiple predecessors finish at the same time. Speedup would be undermined without techniques such as work-stealing queues or batching.

Finally, latency presents another challenge. Graph workloads typically have low arithmetic intensity and poor memory locality, which can degrade performance without optimizations such as compressed row storage or graph partitioning.

## 3 Approach

### 3.1 Graph Generation Implementation

For evaluation, we generate synthetic DAGs rather than relying on fixed datasets. The generator constructs layered graphs, where vertices are partitioned into levels and edges primarily connect nodes from earlier layers to later ones. This structure guarantees acyclicity while allowing us to control the overall shape of the graph, such as its depth and width.

Several input parameters are used to control the graph structure. The primary parameter is -n (number of vertices), which determines the overall size of the graph. The parameters -l (number of layers) and -w (average layer width) control how vertices are distributed across layers, effectively shaping the graph to be either deeper (more sequential) or wider (more parallel). If these values are not provided, they are inferred automatically based on the total number of vertices.

The -wv (width variation) parameter, which we later call $\alpha$ in the report, controls how evenly nodes are distributed across layers. Higher values produce more uniform layer sizes, while lower values introduce greater imbalance. This parameter is important because it directly influences the size of each frontier and therefore the amount of available parallelism at each level.

Edge connectivity is governed by -d (edge density), which specifies the average out-degree per vertex. Additional constraints are imposed by -fo (maximum fan-out) and -fi (maximum fan-in) to

avoid extreme cases with highly skewed degree distributions. These parameters affect both the total number of edges and the computational workload per node.

To introduce more realistic graph structures, we include -s (skip probability) and -sk (maximum skip distance), which allow edges to connect nodes across non-adjacent layers. This reduces the strictly layered structure and creates longer-range dependencies. As a result, the number of levels may decrease while irregularity in the graph increases.

Finally, a random seed (-r) ensures reproducibility, and the implementation also supports loading a graph from an external file via -f, which bypasses generation entirely.

Overall, these parameters allow us to systematically vary graph characteristics and evaluate how different structural properties, such as frontier size, depth, and degree distribution, affect the performance of our parallel implementations.

### 3.2 OpenMP Implementation

Our OpenMP topological sort traverses the graph in a level-synchronous fashion. The input graph is stored in compressed sparse row (CSR) format, where each vertex's outgoing edges are stored contiguously in a flat array indexed by a row-offset array. The algorithm processes one frontier (set of zero in-degree vertices) at a time, repeatedly generating the next frontier until all vertices are processed.

We implemented two OpenMP variants with increasing levels of optimization.

The **baseline OpenMP version** follows a straightforward parallelization strategy. For each frontier level, we launch an OpenMP `parallel for` loop over all vertices in the current frontier. Each thread processes one vertex, iterating over its outgoing edges. For each neighbor, the thread atomically accumulates contributions to the neighbor's value and atomically decrements its in-degree. If a neighbor's in-degree reaches zero, it is inserted into the next frontier using an atomic increment on a shared counter to reserve a position in the frontier array. While this approach is simple and ensures correctness, it suffers from contention on shared memory locations, especially the global frontier counter and frequently updated in-degree entries. This limits scalability as parallelism increases.

To address this bottleneck, we implement a **thread-local OpenMP version** that reduces contention on the shared frontier structure. Instead of inserting newly ready vertices directly into a global frontier using atomic operations, each thread maintains a private buffer (local queue). The main computation phase remains parallel, with threads still using atomic operations for value accumulation and in-degree updates to ensure correctness. However, newly discovered frontier vertices are appended only to thread-local storage, eliminating contention on the global frontier counter during this phase. After all threads complete processing the current frontier (enforced by the implicit barrier at the end of the parallel region), the thread-local buffers are merged into the global frontier in a sequential pass. This design changes the number of synchronization-heavy operations on the frontier from being proportional to the number of discovered vertices to being proportional to the number of threads, significantly reducing contention and improving performance.

Both versions use dynamic scheduling with a fixed chunk size to balance workload across threads. Vertex degrees may vary significantly and would have lead to load imbalance under static scheduling. Additionally, atomic capture operations are used when decrementing in-degrees to ensure that the check for zero is performed consistently with the update.

### 3.3 CUDA Implementation

Our CUDA implementation performs a parallel topological sort using a level-by-level (BFS-style) approach based on Kahn's algorithm. The graph is stored in CSR format, where each node's outgoing edges are stored as a contiguous chunk in a flat array, indexed by a row offset array. At a high level, the algorithm processes one frontier at a time: for each level, we launch a CUDA kernel, then synchronize on the host before moving on to the next level.

We implemented three versions of the kernel with increasing levels of optimization.

The **baseline CUDA version** assigns one thread to each node in the current frontier. Each thread loops over its node's outgoing edges, adds its value to its neighbors using atomicAdd, and decre-

ments their in-degrees with atomicSub. If a neighbor's in-degree reaches zero, it gets added to the next frontier using another atomicAdd on a global counter. This version is simple and correct, but it doesn't scale well because many threads end up contending on that single global counter when pushing nodes into the next frontier.

To improve this, the **block-local version** reduces contention on the global counter. Instead of writing directly to global memory, each block first collects newly ready nodes into a shared-memory queue. Since shared memory is much faster and only involves threads within the block, this significantly reduces contention. After all threads in the block finish, one thread performs a single atomicAdd to reserve space in the global frontier, and then the block writes its results out in parallel. This changes the number of global atomics from being proportional to the number of new nodes to being proportional to the number of blocks, which helps a lot on graphs with high parallelism.

Finally, the **warp-per-node version** increases parallelism within each node. Instead of assigning one thread per node, we assign an entire warp (32 threads) to process a single node's edges. Each thread in the warp handles a different subset of edges in a strided way, so memory accesses become coalesced. This improves memory bandwidth usage compared to the baseline, where each thread accesses edges sequentially. This version still uses the same block-local buffering strategy to reduce contention when adding nodes to the next frontier. We also use a grid-stride loop so that all warps stay busy even if the frontier size doesn't perfectly match the number of available warps.

Overall, each optimization targets a different bottleneck: the baseline is limited by global atomic contention, the block-local version reduces that contention, and the warp-per-node version improves memory access efficiency while keeping the benefits of block-local buffering.

# 4 RESULTS

## 4.1 RESULTS ON OPENMP

### 4.1.1 GHC (8 PROCESSORS)

We measure running time of different strategies (sequential, omp basic, omp local) while varying input space parameters.

Number of vertices: $n$
Number of layers: $L$
Width variance across layers: $\alpha$
Average out-degree: $d$

We measure how performance varies with each input parameter. We also give speculative analysis on why performance varies this way. In section 4.1.3 we use hardware profiling to provide evidence to some parts of the analysis.

Fix $n$ = 1e7, $\alpha$ = 10, $d$ = 3. Vary $L$. Run on 8 processors.

| $L$ | Seq. Time (ms) | Omp Basic (ms) | Omp Local (ms) | Basic Speedup | Local Speedup |
|---|---|---|---|---|---|
| 1e2 | 2534 | 853 | 668 | 3.0 | 3.8 |
| 1e3 | 2409 | 913 | 571 | 2.6 | 4.2 |
| 1e4 | 2324 | 981 | 644 | 2.4 | 3.6 |
| 1e5 | 2190 | 2101 | 1869 | 1.0 | 1.2 |

Table 1: Speedups for different graph depths

Analysis: As the number of layers increases, the number of vertices in each layer decreases. This increases the chance of two threads discovering the same node at the same time. Since the code requires the accesses to `in_degree[]` to be atomic, this causes sync stalls. Therefore, the basic parallel strategy's speedup decreases. When $L$ = 1e5, there are on average 100 vertices on the

frontier, and at any given time the 8 processors visit 3 vertices each on average. The chance of collision is high, sync stalls become drastic and speedup is severely compromised (1.2x).

Fix $n$ = 1e7, $L$ = 1e3, $\alpha$ = 10. Vary $d$. Run on 8 processors.

| $d$ | Seq. Time (ms) | Omp Basic (ms) | Omp Local (ms) | Basic Speedup | Local Speedup |
|---|---|---|---|---|---|
| 1 | 2272 | 617 | 479 | 3.7 | 4.7 |
| 3 | 2409 | 913 | 571 | 2.6 | 4.2 |
| 5 | 2518 | 984 | 640 | 2.6 | 3.9 |
| 10 | 2714 | 893 | 785 | 3.0 | 3.5 |

Table 2: Speedups for different out degrees

Analysis: At any given time, the 8 threads each visit $d$ nodes on average, for a total of $8d$ nodes. The accesses to `in_degree[]` need to be atomic, so when two threads visit the same node, we have sync stalls. The chance of collision increases with $d$. Therefore, the speedup of local parallel strategy increases up to 4.7x as $d$ decreases to 1.

Fix $n$ = 1e7, $L$ = 1e3, $d$ = 3. Vary $\alpha$. Run on 8 processors.

| $\alpha$ | Seq. Time (ms) | Omp Basic (ms) | Omp Local (ms) | Basic Speedup | Local Speedup |
|---|---|---|---|---|---|
| 0.3 | 1688 | 578 | 477 | 2.9 | 3.5 |
| 1 | 2173 | 765 | 552 | 2.8 | 3.9 |
| 3 | 2368 | 672 | 570 | 3.5 | 4.2 |
| 10 | 2409 | 913 | 571 | 2.6 | 4.2 |

Table 3: Speedups for different width variances across layers

Analysis: We quantify the "evenness" of the graph's shape using the Dirichlet distribution. Indeed, we generated graphs such that the layer widths follow a symmetric Dirichlet distribution with $\alpha$ as parameter. Large $\alpha$ means that all the layers tend to have the same number of vertices, whereas small $\alpha$ means that the number of vertices tend to vary largely from one layer to another. For small $\alpha$, the layer widths are unevenly distributed, meaning that some layers will have significantly fewer vertices. This causes more contention when two threads visit the same vertex. For a local parallel strategy, the speedup decreases as $\alpha$ decreases.

#### 4.1.2 PSC (Up to 64 Processors)

We measure how different strategies (sequential, omp basic, omp local) scale as the number of available processors increases. We test on 2 separate input configurations, whose parameter values are shown in table 4.

| Config Index | $n$ | $L$ | $\alpha$ | $d$ |
|---|---|---|---|---|
| 1 | 1e7 | 1e3 | 10 | 5 |
| 2 | 1e7 | 1e2 | 3 | 5 |

Table 4: Input configurations for PSC experiments

Config 1:
Sequential time: 2160 ms

| Num. Proc. | Omp Basic (ms) | Omp Local (ms) | Basic Speedup | Local Speedup |
|---|---|---|---|---|
| 2 | 2085 | 1907 | 1.0 | 1.1 |
| 4 | 1616 | 1151 | 1.3 | 1.9 |
| 8 | 1411 | 682 | 1.5 | 3.2 |
| 16 | 1293 | 338 | 1.7 | 6.4 |
| 32 | 747 | 206 | 3.0 | 10.4 |
| 64 | 535 | 164 | 4.0 | 13.1 |

Table 5: Scalability on input configuration 1

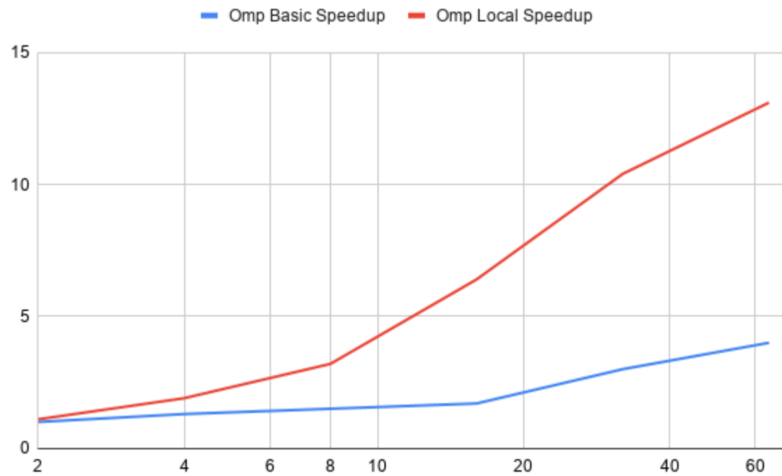


Figure 2: Speedup scalability on input configuration 1

Analysis: The Omp local parallel strategy scales much better than the Omp basic parallel strategy. This was not as evident on GHC with 8 processors. When the number of processors is large, there is more contention for the queue. Therefore, the local parallel strategy is faster since each thread fills up its local queue first. However, the local parallel strategy still suffers from contention on `in_deg[]`, which also becomes a greater issue when the number of processors gets large. The maximal speedup was 13.1x on 64 processors. An efficient choice would be 10.4x speedup on 32 processors.

Config 2:
Sequential time: 1963 ms

| Num. Proc. | Omp Basic (ms) | Omp Local (ms) | Basic Speedup | Local Speedup |
|---|---|---|---|---|
| 2 | 2056 | 1818 | 1.0 | 1.1 |
| 4 | 11435 | 1035 | 1.4 | 1.9 |
| 8 | 1281 | 621 | 1.5 | 3.2 |
| 16 | 1276 | 327 | 1.6 | 6.0 |
| 32 | 668 | 169 | 3.0 | 11.7 |
| 64 | 442 | 132 | 4.4 | 14.9 |

Table 6: Scalability on input configuration 2

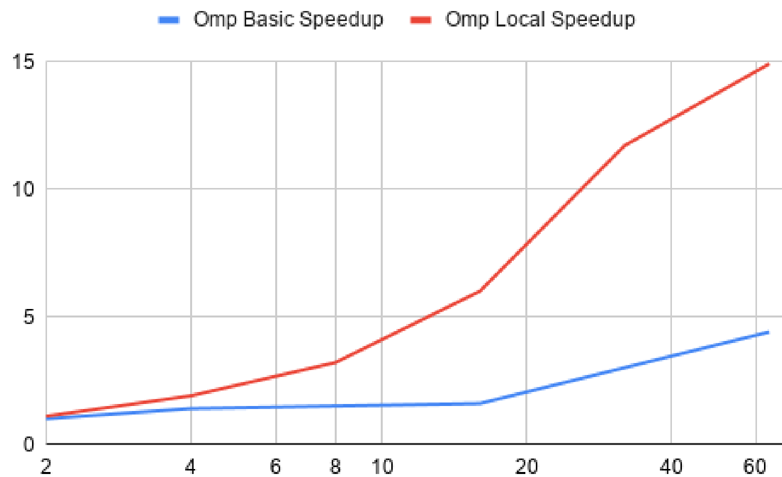


Figure 3: Speedup scalability on input configuration 2

Analysis: As for input config 1, the omp local parallel strategy scales much better than the omp basic parallel strategy. The maximal speedup was 14.9x on 64 processors. An efficient choice would be 11.7x speedup on 32 processors.

#### 4.1.3 OpenMP Profiling

We use Linux profiling tool `perf` on GHC to analyze runtime behavior at a lower level. We focused on identifying synchronization stalls due to contention on `in_deg[]` and `value[]`. By inspecting the call graph via `perf report`, we were able to measure the percentage of time lost due to synchronization overhead. An example of how we obtain the percentage is presented in figure 4.

```
Percent │
        │     #pragma omp atomic
   0.27 │10f:   movd    %edx,%xmm1
   0.01 │       mov     %edx,%eax
   0.35 │       addss   %xmm0,%xmm1
   0.25 │       movd    %xmm1,%edi
  10.23 │       lock    cmpxchg %edi,(%rsi)
   0.01 │     ↓ jne     1d5
        │     return *(this->_M_impl._M_start + __n);
   0.19 │       mov     0x18(%rbx),%rax
        │     values[v] += val_u;
        │
        │     int32_t new_deg;
        │     #pragma omp atomic capture
        │     new_deg = --in_deg[v];
        │       mov     0x34(%rsp),%edx
        │     #pragma omp atomic capture
        │       mov     (%rax),%rax
  13.41 │       lock    subl    $0x1,(%rax,%rdx,4)
```

Figure 4: Profiling shows that in the current run, the omp local strategy spent $10.23\%$ of the time sync stalling on the values array and $13.41\%$ on the in-degree array, for a total of $23.64\%$ sync stall.

In the OpenMP strategies, the writes to `in_deg[]` and `value[]` have to be atomic. When multiple processors discover the same vertex $v$, contention occurs. `perf` shows that this phenomenon is one of the primary reasons for sub-ideal speedup. The profiling results also sheds light on how different graph attributes affects access patterns,

We find that as the average out-degree parameter, $d$, increases, contention increases. The result is shown in table 7 and figure 5, . This confirms the speculative analysis conducted in section 4.1.1.

| d | Omp Local (ms) | Sync Proport. | Sync Time (ms) | Compute/Load Time (ms) |
|---|---|---|---|---|
| 1 | 479 | 0.16 | 77 | 402 |
| 3 | 571 | 0.24 | 137 | 434 |
| 5 | 640 | 0.32 | 205 | 435 |
| 10 | 785 | 0.39 | 306 | 479 |

Table 7: Sync proportion of local parallel strategy for different graph out-degrees

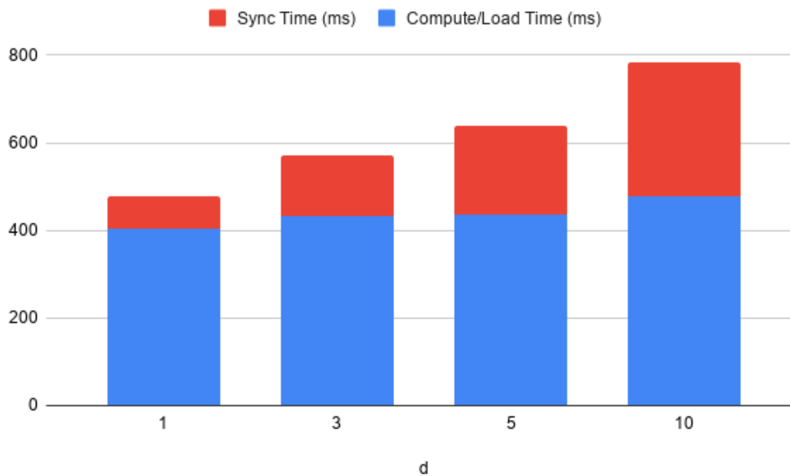


Figure 5: Runtime breakdown of omp local parallel strategy for different graph out-degrees

### 4.2 Results on CUDA

Similar to the OpenMP results, we measure running time of different strategies (CUDA Baseline, CUDA Block-Local, and CUDA Warp) while varying input space parameters. Our baseline is the sequential runtime of the topological sort algorithm with Kahn's algorithm. We measure the runtimes of all CUDA versions as well as their speedup relative to the sequential version.

We vary the following parameters:

- Number of layers: $L$
- Width variance across layers: $\alpha$
- Average out-degree: $d$

and provide analysis on performance.

#### 4.2.1 Varying the Number of Layers

First, we vary $L$, the number of layers, while fixing $n$ = 1e7, $\alpha$ = 10, $d$ = 3.

From Table 8, performance degrades significantly as $L$ increases. When $L$ is small ($10^2$ to $10^3$), each layer contains many vertices, resulting in large frontiers and high available parallelism. In this regime, all three CUDA implementations achieve strong performance, with speedups exceeding $30\times$. The warp-per-node kernel performs best, benefiting from both coalesced memory accesses and additional parallelism across edges.

| $L$ | Sequential (ms) | Base (ms) | Block-Local (ms) | Warp (ms) | Base Speedup | Block-Local Speedup | Warp Speedup |
|---|---|---|---|---|---|---|---|
| $10^2$ | 2598 | 79 | 88 | 75 | 32.81 | 29.62 | 34.84 |
| $10^3$ | 2630 | 74 | 74 | 68 | 35.63 | 35.70 | **38.81** |
| $10^4$ | 2464 | 307 | 247 | 183 | 8.04 | 9.99 | 13.49 |
| $10^5$ | 2291 | 2489 | 2052 | 1502 | 0.92 | 1.12 | 1.52 |

Table 8: CUDA Performance vs. $L$ with $n = 10^7$, $\alpha = 10$, $d = 3$.

As $L$ increases to $10^4$, the frontier size shrinks, reducing the number of active threads and leading to lower GPU utilization. Although the warp kernel still outperforms the other variants, overall speedup drops substantially.

At $L = 10^5$, the frontier becomes very small, and parallelism is severely limited. In this case, kernel launch overhead and host-device synchronization dominate execution time. As a result, the CUDA implementations provide little to no speedup over the sequential baseline, with the baseline kernel even performing worse than sequential. This demonstrates that level-synchronous approaches are not well suited for very deep graphs with limited frontier parallelism.

#### 4.2.2 Varying the Average Outdegree

Fix $n$ = 1e6, $L$ = 1000, $\alpha$ = 10. Vary $d$.

| $d$ | Sequential (ms) | Base (ms) | Block-Local (ms) | Warp (ms) | Base Speedup | Block-Local Speedup | Warp Speedup |
|---|---|---|---|---|---|---|---|
| 1 | 141 | 23 | 21 | 17 | 6.06 | 6.78 | 8.52 |
| 10 | 203 | 39 | 36 | 20 | 5.14 | 5.72 | 10.41 |
| 100 | 699 | 142 | 168 | 44 | 4.93 | 4.16 | 15.76 |
| 1000 | 5242 | 1080 | 1404 | 293 | 4.86 | 3.73 | **17.92** |

Table 9: CUDA Performance vs. $d$ with $n = 10^6$, $L = 1000$, $\alpha = 10$.

From Table 9, increasing $d$ increases the amount of work per node, which affects the three implementations differently. For the baseline and block-local kernels, higher $d$ leads to more atomic operations on `values` and `in_deg`, increasing contention and limiting scalability. As a result, their speedup remains relatively flat or decreases slightly as $d$ grows.

In contrast, the warp-per-node kernel benefits from higher $d$. Since a full warp is assigned to each node, increasing the number of edges increases the amount of parallel work that can be distributed across threads within the warp. This improves utilization and amortizes memory and synchronization costs. As a result, the warp kernel achieves increasing speedup with larger $d$, reaching up to $17.9\times$ at $d = 1000$.

Overall, this shows that the warp-based approach is more effective at exploiting edge-level parallelism, while the other implementations are more sensitive to contention from atomic operations.

#### 4.2.3 Varying the Width Variance

Fix $n$ = 1e7, $L$ = 1e3, $d$ = 3. Vary $\alpha$.

| $\alpha$ | Sequential (ms) | Base (ms) | Block-Local (ms) | Warp (ms) | Base Speedup | Block-Local Speedup | Warp Speedup |
|---|---|---|---|---|---|---|---|
| 0.3 | 1796 | 239 | 264 | 62 | 7.50 | 6.80 | 29.20 |
| 1 | 2313 | 107 | 106 | 69 | 21.54 | 21.77 | 33.59 |
| 3 | 2514 | 75 | 76 | 68 | 33.38 | 33.28 | 36.87 |
| 10 | 2584 | 74 | 73 | 68 | 34.99 | 35.31 | **38.10** |

Table 10: CUDA Performance vs. $\alpha$ with $n = 10^7$, $L = 10^3$, $d = 3$.

From Table 10, performance improves as $\alpha$ increases. Larger $\alpha$ values correspond to more uniform layer widths, which leads to consistently large frontiers and stable parallelism across levels. In this setting, all CUDA implementations achieve high speedups, with the warp kernel again performing best. When $\alpha$ is small, the graph becomes highly imbalanced, with some layers containing very few vertices. These small frontiers reduce available parallelism and lead to underutilization of the GPU. The baseline and block-local kernels are particularly affected, showing a significant drop in speedup at $\alpha = 0.3$. The warp-per-node kernel is more robust in this regime. Even when the number of frontier nodes is small, it can still exploit parallelism across edges within each node. This allows it to maintain relatively strong performance compared to the other variants, although some degradation is still observed. Overall, these results show that balanced graphs with large and consistent frontiers are the most favorable for GPU execution, while irregular graphs expose limitations in parallelism and increase the impact of overheads.

#### 4.2.4 Performance Bottleneck Analysis

We conducted a performance bottleneck analysis by running our kernel with Nvidia Nsight Compute (NCU).

| **Metric** | **Small Frontier** | **Large Frontier** |
|---|---|---|
| Achieved Occupancy | 24.6% | 98% |
| Eligible Warps / Scheduler | 0.05 | 0.05 |
| No Eligible Cycles | 95.9% | 96.3–96.6% |
| L1TEX Stall | 52.0% | 74.6–77.9% |
| DRAM Utilization (of peak) | 3.5% | 23.5% |
| Compute Utilization (of peak) | 1.7% | 5.5% |
| Avg Active Threads / Warp | 21/32 | 20–21/32 |

Table 11: Key NCU metrics for CUDA warp kernel under small and large frontier configurations

As shown in Table 11, the performance bottleneck differs depending on the frontier size.

For the small frontier case, performance is primarily limited by insufficient parallelism. The achieved occupancy is only 24.6%, and NCU reports that the kernel launches do not generate enough waves to fully utilize the GPU. This indicates that the available work per frontier is too small to keep the hardware busy, so speedup is limited by algorithmic dependencies.

For the large frontier case, parallelism is no longer the limiting factor, as occupancy reaches approximately 98%. However, performance remains poor due to extremely low scheduler efficiency: only 0.05 warps per scheduler are eligible on average, and approximately 96% of cycles have no eligible warp. This indicates that most warps are stalled and unable to issue instructions.

NCU identifies the dominant stall reason as L1TEX dependency stalls, accounting for approximately 75–78% of cycles. This suggests that execution is limited by latency from dependent memory operations rather than compute or memory bandwidth. This is consistent with the low DRAM utilization (only 23.5% of peak), which shows that the kernel is not bandwidth-bound.

From the implementation, the most likely source of these dependency stalls is the use of global atomic operations in the inner loop, particularly:

```
if (atomicSub(&in_deg[v], 1) == 1) { ... }
```

This operation introduces a true dependency because the return value of the atomic is immediately used to determine control flow. As a result, the warp must wait for the atomic operation to complete before proceeding, leading to frequent scoreboard stalls.

Overall, the CUDA implementation is limited by lack of parallelism for small frontiers and by latency from dependent global atomic operations for large frontiers, rather than by memory bandwidth or compute throughput.

### 4.3 COMPARISON OF CPU VS. GPU

We compare the CPU (OpenMP) and GPU (CUDA) implementations in terms of how they exploit parallelism and how sensitive they are to graph structure. Both approaches take advantage of the fact that a DAG defines only a partial order, but in practice they behave quite differently.

On the CPU side, the implementation uses a relatively small number of threads, up to 64 in our experiments, and performs reasonably well even when the available parallelism is not very large. The thread-local optimization helps reduce contention on the global frontier, which would otherwise be a bottleneck. In our experiments, the CPU achieves around $13\times$ to $15\times$ speedup on larger graphs. As the frontier becomes smaller, for example when $L$ is large, or when the out-degree $d$ increases, contention from atomic updates grows, but the slowdown is gradual rather than abrupt.

The GPU shows a different behavior. It can reach much higher peak speedups, around $30\times$ to $38\times$ in our results, but only when there is enough parallel work to keep the hardware busy. When the frontier is large, the CUDA kernels perform well, especially the warp-per-node version which can exploit parallelism across edges. However, when the graph becomes deep and the frontier size shrinks, the GPU becomes underutilized. In cases such as $L = 10^5$, performance drops significantly and can get close to the sequential baseline because of kernel launch overheads and limited parallel work.

The two implementations also differ in how they respond to irregular graph structure. The CPU is relatively robust even when the graph is imbalanced, for example when $\alpha$ is small, although contention still increases. The GPU is more sensitive in this setting, since small or uneven frontiers lead to poor utilization and load imbalance across threads. One exception is for very high-degree graphs, where the warp-per-node strategy benefits from having more edges per node, since it exposes additional parallelism within each node.

The main bottlenecks are also different. On the CPU, profiling shows that a significant fraction of execution time, sometimes around $30\%$ to $40\%$, is spent on synchronization, especially atomic updates to `in_deg[]` and `value[]`. On the GPU, the issue is more related to latency. Nsight Compute shows that a large fraction of cycles, around $75\%$, are stalled due to memory dependencies from atomic operations whose results are used immediately. This means that even with high occupancy, performance can still be limited by these stalls.

Overall, the GPU achieves higher peak throughput when the graph has enough parallelism, such as wide graphs with large frontiers. The CPU provides more stable performance across different inputs and handles deep or irregular graphs better. In practice, the better choice depends more on the structure of the graph than on its size alone.

## 5 REFERENCES

Our references include the OpenMP and CUDA reference guides:


- **OpenMP Reference Guide:** https://www.openmp.org/resources/refguides/
- **CUDA Reference Guide:** https://docs.nvidia.com/cuda/cuda-c-programming-guide/


## 6 TEAM MEMBER CONTRIBUTIONS

- **Andy** primarily worked on the CUDA implementation, benchmarking and analysis, as well as the graph generator.
- **Yizhou** primarily worked on the OpenMP implementation, benchmarking and analysis.
- **Both** members worked on the report and presentation equally.
- The total work was a $50\% - 50\%$ split.