Heterogeneous CPU-GPU Orchestration for Repeated Minimum Spanning Tree Computation

Linxuan Ma

linxuanm

Joseph Wan

zhilianw

1 Summary

We implemented a parallel graph-processing program for minimum spanning tree computation, focusing on workloads where the graph topology remains fixed while edge weights change across rounds. We evaluated multiple implementations, in OpenMP and CUDA, on graph snapshots with varying structural properties to understand where each approach performs well and where it encounters bottlenecks. Based on these findings, we designed a hybrid heterogeneous strategy that orchestrates the CPU and GPU implementations in parallel to hide GPU latency and match algorithmic choices to graph characteristics. On CMU GHC machines using 8 CPU cores and CUDA, our Hybrid strategy amortizes its preprocessing cost after ≈ 6 MST rounds and achieves about half the per-round time of the one-shot MP and CU strategies on our mixed sparse/dense R-MAT workload.

2 Background

The Minimum Spanning Tree (MST) is a classical graph theory problem with critical applications in network design, taxonomy, and clustering. Given a connected, undirected graph with weighted edges (input), the objective is to find the subset of edges that connects all vertices together without any cycles, such that the total edge weight is minimized (output). While sequential algorithms like Kruskal's or Prim's efficiently solve this for smaller datasets, massive modern graphs require parallel processing to compute the MST within a reasonable timeframe. Borůvka's algorithm is historically the most amenable to parallelization, and understanding its workload characteristics is key to mapping this problem to multi-core CPUs and GPUs.

When computing the MST on large-scale graphs, a significant bottleneck is the sheer volume of edges that must be processed. In Kruskal's, sorting the edges takes $O(E \log E)$ time, forcing a massive sequential bottleneck. In Borůvka's algorithm, the most expensive phase is the neighborhood search—scanning all edges attached to every component to find the absolute minimum outgoing edge. Because components can independently search for their respective minimum edges without altering the graph topology, this search phase is highly computationally expensive but incredibly ripe for parallelization.

Graph traversals inherently suffer from extremely poor spatial locality. Because vertices possess varying degrees and connecting edges span essentially arbitrary memory addresses, traversing adjacency lists or chasing pointers up a Union-Find tree results in scattered, non-contiguous memory accesses that routinely thrash the CPU/GPU caches. Moreover, the control flow of sparse graph processing is not particularly amenable to SIMD execution. For example, different threads executing the Find operation will traverse Union-Find trees of vastly different depths, causing threads within the same SIMD lane/warp to serialize and idle while waiting for the deepest traversal to complete.

Fixing Graph Topology

To make pre-processing worthwhile, we target repeated MST computations where the vertex and edge set remain fixed, but edge weights change between queries. This computation pattern appears in applications where the underlying network structure changes slowly, but the cost of using each connection changes frequently. For example, in transportation and logistics networks, the available roads or routes may remain fixed while travel times or congestion levels vary throughout the day. In communication and sensor networks, the set of possible links may stay the same while dynamic properties like latency, bandwidth or energy costs fluctuate over time. This premise enable us to pre-process the graph structure and perform scheduling accordingly.

Borůvka's Algorithm

Borůvka's algorithm is a highly parallel MST algorithm that builds the spanning tree through round-based graph contraction. It uses the theorem that "given a disjoint component C from a partition of the graph, the minimum out-going edge from C will be in the final MST".

This theorem allows us to parallelize on the edges of the graph during a contraction round if we have an efficient way to perform the graph contractions. We use **union-find** to accomplish this. A union-find is a data structure that efficiently maintains a collection of disjoint sets and supports two main operations: finding which set an element belongs to, and merging two sets together.

A high-level illustration of the Borůvka's algorithm is as follows:

Algorithm 1 Borůvka's Minimum Spanning Tree Algorithm

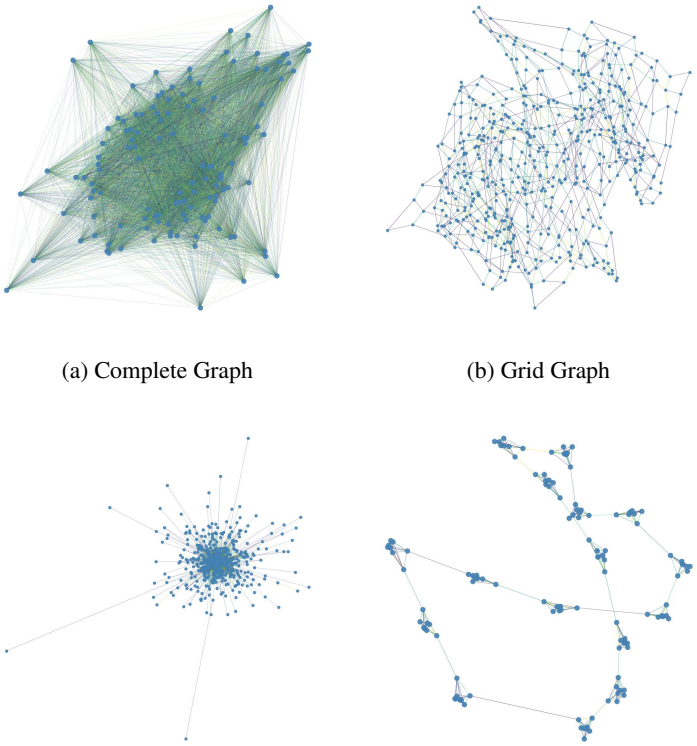
Require: Graph $G = (V, E)$
Ensure: Minimum Spanning Tree T
$T \leftarrow \emptyset$
Initialize each vertex as its own component
while there is more than one component **do**
    For each component, find its minimum-weight outgoing edge
    if no outgoing edge is found for any component **then**
        break
    end if
    for each selected minimum outgoing edge e **do**
        if e connects two different components **then**
            Add e to T
            Merge the two components connected by e
        end if
    end for
end while
return T

Graph Features

To ensure that our benchmark fairly represents the graphs encountered in a real-life scenario, we wrote a script `generate_graphs.py` to generate graphs with varying features. We created the following types of graphs.

- **Circulant:** Each vertex is connected to its k nearest neighbors in both directions along a ring topology, which creates a $2k$-regular graph. This generates a highly structured, strictly regular graph where every vertex has an identical degree of 2k. Used to test the parallel algorithm's performance on uniform, predictable workloads with strong local connectivity.

- **Complete:** Every pair of vertices is connected, resulting in $m = n(n-1)/2$ edges. This is the densest possible simple graph. The degree is uniform at $n-1$ for all vertices. Evaluates the effectiveness of the algorithm's edge elimination strategies.
- **Grid:** A $\sqrt{n} \times \sqrt{n}$ grid with 4-connectivity. The graph is extremely sparse (roughly $m \approx 2n$) and highly structured with strong spatial locality. Measures performance on regular, structured workloads with good spatial locality. It tests cache behavior when the graph structure maps naturally to the memory layout, serving as a baseline to compare against irregular graphs.
- **RMAT:** Generates graphs with severely skewed degree distributions using the recursive matrix model. A small number of hub vertices have a very high degree, while the vast majority of vertices have a low degree. This models real-world networks like social graphs and web link structures. Evaluates how the system handles load imbalance across threads due to the skewed degree distribution.
- **Caveman:** Consists of a ring of cliques. Evaluates the algorithm's component handling and cross-cluster edge processing abilities.



(a) Complete Graph (b) Grid Graph

(c) RMAT Graph (d) Caveman Graph

Figure 1: Visualization of each type of graph ($n = 128$).

We created these graphs at the following sizes to test for scalability:

Size	Tiny	Small	Medium	Large	Huge
n	16	64	1,024	65,536	262,144

Table 1: Graph sizes used in our benchmark.

"Round" Clarification

Our project report uses the term "round" for many different things. For clarification, In this report, we use the term "round" in two different ways.

We use **Borůvka round** to refer to one contraction iteration inside a single MST computation. We use **MST round** to refer to one complete MST computation on the fixed graph topology with a new set of edge weights. MP and CU are one-shot strategies because they compute one MST round from scratch. Hybrid is a multi-shot strategy because it performs pre-processing once and reuses the resulting partition across many MST rounds.

3 Method

One major obstacle we encountered in this project initially was that for regular MST generation, its complexity is $O(n + m)$ (n denotes the number of vertices, and m the number of edges). The linearity implies that we cannot perform a linear-time preprocessing of the graph for scheduling purposes without potentially dominating the cost of the MST computation itself, unlike in the wire routing assignments where preprocessing could be amortized over a more expensive workload. Hence, we moved our initial goal from simple MST generation to repeated MST generation on graphs with fixed topology but dynamically changing edge weights, where structure preprocessing and strategy selection can be amortized across many rounds.

Our approach is done in two parts. First, we implemented two strategies, **MP** and **CU**, using OpenMP and CUDA respectively, to parallelize Borůvka's algorithm under different backends. **MP** serves as a CPU-based shared-memory implementation that is flexible and less sensitive to irregular graph structure, while **CU** is capable of massive parallelism but is easily affected by graph irregularities. We evaluated these implementations on graph snapshots with varying structural properties, such as density and degree distribution, to identify when each backend performs well and when it becomes inefficient. We refer to MP and CU as **one-shot** strategies, since they compute each MST query independently, without reusing topology-specific preprocessing across queries.

Second, we used these observations to design a hybrid, heterogeneous strategy for repeated MST computation. Since the graph topology is fixed across rounds, we can analyze structural features once and use them to guide future scheduling decisions, rather than treating every round as an independent one-shot MST computation.

3.1 One-Shot Strategies

We implemented two one-shot strategies (CU and MP) to compute the MST of a subgraph. Both MP and CU are self-contained Borůvka implementations. We call them one-shot strategies because each invocation computes an MST from scratch on the given graph or subgraph, without using persistent structural information from previous MST computations.

3.1.1 OpenMP Strategy (MP)

The OpenMP implementation is parameterized over the union-find type, so that the same Boruvka skeleton can be paired with any of three disjoint-set implementations. The graph is held as a flat array of weighted edges and is never rebuilt between rounds; contraction is expressed implicitly through the union-find representative of each endpoint, which means no costly compaction or rewiring of the edge list is required between iterations.

We implemented MP with sequential Boruvka (contraction) rounds. Each Boruvka round runs inside a single OpenMP parallel region and proceeds in three phases, separated by the implicit barriers at the end of each iteration:

- **Reset:** A parallel loop initializes a per-supervertex *best edge* slot to a sentinel value indicating "no candidate yet." Each slot is an atomic word that holds the minimum-weight outgoing edge found so far for that supervertex, with edge indices as tie-breakers so that the comparison defines a total order on edges and the result is deterministic regardless of thread schedule.
- **Min-edge selection (edge-parallel):** A parallel loop over all m edges computes the current root of each endpoint via the union-find. If the edge connects two disjoint components, it performs a lock-free atomic minimum against the slot of each endpoint's root. The atomic minimum is implemented as a compare-and-swap loop.
- **Contraction:** A parallel loop over supervertices reads each winning candidate edge and attempts to merge its endpoints in the union-find. Edges whose merge succeeds are pushed onto a thread-local output buffer (avoiding contention on a shared MST list). After the parallel region exits, these per-thread buffers are concatenated into the global MST.

The outer loop terminates when a round produces no successful merges, indicating that the MST is complete.

Synchronization Points

To eliminate contention, we refactored Boruvka's algorithm into the above three phases so that each phase accesses a disjoint set of memory locations and the few unavoidable conflicts are pushed onto narrow, word-sized atomic slots rather than shared data structures guarded by locks. The reset phase performs only independent stores and races against nothing. The selection phase restricts all inter-thread communication to the per-supervertex best edge slots. Updates to those are serialized by a single compare-and-swap on an atomic word (packed edge data) and no critical section around the edge classification. The contraction phase reads each slot from only one thread that owns the corresponding supervertex, so the per-vertex reads are race-free, and the only remaining cross-thread interaction is on the union-find, where the lock-free variants reduce contention to retried CAS attempts on a small number of root nodes per round.

Union-Find Implementation

We implemented three variants of union-find with different synchronization strategies to explore how disjoint-set design choices interact with the contention pattern produced by edge-parallel Boruvka:

- **Coarse-Grained Locking** `locked`: Ensures correctness by wrapping operations in a global lock. This implementation serializes every disjoint-set operation and is expected to perform poorly under high thread counts.
- **Atomic UF with Half Path-Compression** (`atomic-half`): Each `find` operation reads the current node's parent and grandparent and attempts a compare-and-swap to redirect the node directly to its grandparent before advancing upward. Each lookup has depth$(T)/2$ CAS operations.
- **Atomic UF with Full Path-Compression** (`atomic-full`): Each `find` operation directs all node on the path to root to the supervertex root directly. Each lookup has depth(T) CAS operations.

Misc Optimizations

We made some small optimizing designs to reduce the fixed overhead of launching and synchronizing OpenMP threads:

- **Edge Packing**: Each edge is packed into a `uint64_t` with the edge weight as the upper 32 bits so weight comparison on edges works directly. This restricts an edge write to a one-value CAS operation, along with reducing memory footprint for more effective caching.
- **Structure of Arrays**: We used structure of arrays on data structures that performs pointer chasing to improve cache locality by storing frequently accessed fields contiguously and reducing unnecessary memory loads.

3.1.2 CUDA Strategy (CU)

Our CUDA implementation (CU) computes the minimum spanning tree using a modified Boruvka's algorithm. The graph is represented as a dense adjacency matrix on the GPU, where a positive edge weight indicates the presence of an edge. The algorithm maintains a union-find structure over vertices to track the current connected components. In contrast to MP, CU uses a vertex-oriented contraction strategy to minimize warp divergence.

At initialization, each vertex forms its own singleton component. Inside each MST computation, CU performs multiple Borůvka contraction rounds. Each contraction round consists of three GPU kernels:

1. **Find the cheapest outgoing edge for each component**

   The `find_min` kernel searches for the cheapest edge leaving each component. One warp is assigned to each vertex, allowing the 32 lanes in a warp can cooperatively scan that vertex's outgoing edges in parallel. Each lane checks a subset of neighbors (interleaved) and ignores edges that stay inside the same component.

2. **Component Contraction**

   The `add_edge` kernel processes the chosen edge for each component root. If a component has a valid outgoing edge in `min_edge`, the kernel commits the edge to the result MST and attempts to merge the two endpoint components in the union-find structure.

3. **Rebuild Component Labels**

   The `rebuild_states` kernel updates the component information after the merge phase. It finds the current (union-find) root for each vertex, performs path compression and writes the root back into component. This commits the contraction done in that round and prepares for the next.

Synchronization Points

CU implicitly synchronizes all threads at the end of each of the three steps described previously. Other synchronizations include CAS operations (most notably the union-find structure) and warp-level synchronization to agree on the smallest outgoing edge value (done with `__shfl_down_sync`; cheap in-warp message passing).

Optimizations

We faced more challenges and considerations when designing CU. Unlike OpenMP, the CU implementation must minimize warp divergence, shared memory access (memory stalls) and atomic operations (CAS retries). After tuning our algorithm to benchmark results, we made the following design decisions to achieve an optimal speedup:

- **Warp-level work assignment**: We assigned one warp to each vertex in `find_min` so the code keeps the edge scan within a small group of 32 lanes. This avoids block-wide coordination and reduces the amount of time spend at synchronization points. To perform a `min`-reduction on the minimum edge across the 32 lanes, we used `__shfl_down_sync` (warp-level sync primitive) which is much faster than using shared memory.
- **Per-vertex edge scanning**: Assigning one warp to each vertex ensures that the work within a warp is centered around contractions involving that vertex's component. This gives the warp a more uniform and localized

memory access pattern, while the similarity of the edge checks performed by its lanes helps reduce warp divergence. The inspiration for this design is from [1].

- **Packed edge representation**: The code stores weight and endpoints together in a single `p_edge`. This allows `atomicMin` to compare whole candidate edges directly, reducing the need for multiple memory operations or custom synchronization.

3.2 Hybrid Strategy

After establishing one-shot strategies MP and CU, we implemented a hybrid strategy (Hybrid) that performs multi-shot MST generation on a graph with fixed structure, but varying weights across repeated MST queries, which we call MST rounds. Given a fixed graph, we partition it into a set of disjoint subgraphs D and store this partition structure. The Hybrid strategy uses heuristics to determine whether to assign MP or CU to that graph, based on graph features like size and density. Then, during every MST round with new edge weight input, we iterate through D and use either MP or CU to process each graph to get $|C|$ super-vertices, which we then stitch together to form the final MST.

The rationale behind this design is to amortize structure-dependent work across many MST rounds. Since the graph topology is fixed, the partition D only needs to be computed once. Later rounds only receive new edge weights, so Hybrid can reuse the same subgraph layout and avoid repeated structure analysis. This makes the approach different from the one-shot MP and CU strategies, which process the full graph directly without using persistent information about its structure.

Partitioning also allows Hybrid to match each subgraph with the implementation that is better suited for it. MP tends to work well on smaller or sparser subgraphs because it has lower overhead and avoids the fixed costs of GPU execution. CU is more favorable on larger and denser subgraphs, where the GPU has enough regular parallel work to amortize kernel launch and memory transfer overheads. By choosing between MP and CU per subgraph, Hybrid avoids forcing one implementation strategy onto the entire graph.

A high-level, conceptual description for our algorithm is as follows (assume graph partition D is created, and W is the new weights on that round):

Algorithm 2 HYBRIDCOMPUTE(D, W)

Require: Partitioned subgraphs D, edge weights W
Ensure: MST weight or edge set for the full graph
1: **for all** $d \in D$ **in parallel do**
2: $s \leftarrow$ HEURISTIC(d) ▷ $s \in \{\text{MP}, \text{CU}\}$
3: $G_d \leftarrow$ SUBGRAPH(d, W)
4: $C_d \leftarrow$ RUNSTRATEGY(s, G_d)
5: **end for**
6: Wait for all subgraph computations to finish
7: $C \leftarrow \bigcup_{d \in D} C_d$
8: **return** SEQUENTIALMST(C)

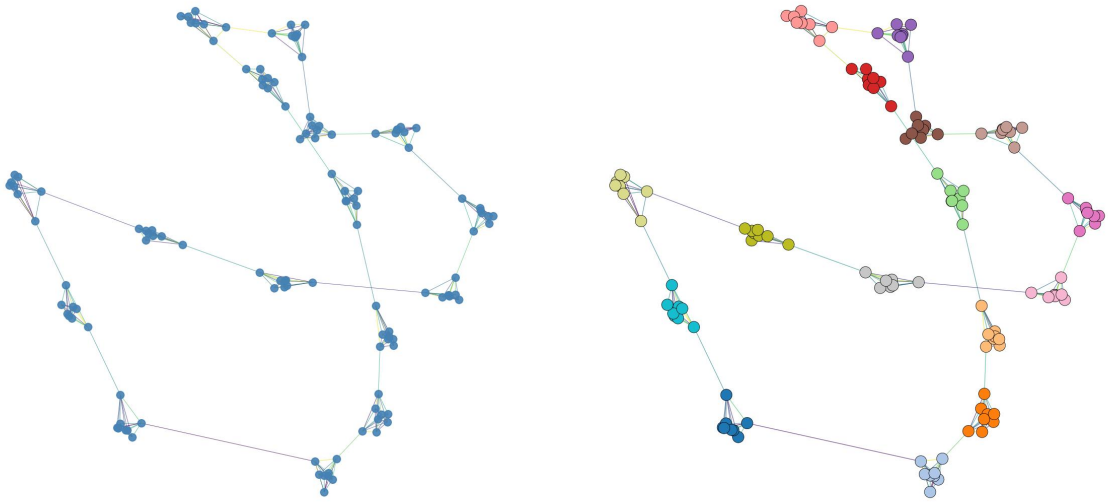


Figure 2: Ideal partition of a graph into dense subgraphs

For each partitioned subgraph, MP or CU computes a local MST and contracts that subgraph into a supervertex. Hybrid then builds a smaller contracted graph containing these supervertices, along with the original edges that cross partition boundaries. A final sequential MST pass on this contracted graph builds the local results into the full MST.

K-Core Decomposition

For the graph partitioning algorithm during pre-process, we implemented a k-core decomposition algorithm through vertex peeling. A k-core is a maximal subgraph in which every vertex has degree at least k within that subgraph. This decomposition separates the graph by local connectivity: vertices in higher cores belong to denser and more strongly connected regions, and can be grouped together into denser subgraphs. We use k-core decomposition because it gives a lightweight way to identify dense subgraphs before repeated MST computation (i.e., as a way of clustering). Separating the dense and sparse regions of a graph is beneficial for local processing in future rounds, as the Hybrid heuristic can utilize their structural feature to assign strategies to them.

At a high level, the vertex peeling process repeatedly removes vertices whose current degree is below the active threshold. When a vertex is removed, the degrees of its remaining neighbors are updated, which may cause more vertices to become removable due to falling under the threshold. This process continues until no more vertices can be removed for the current k. We use the resulting cluster labels to group vertices into disjoint subgraphs for Hybrid. This pre-processing is done once for the fixed graph topology, and the resulting partition is reused across MST rounds with different edge weights.

Heterogeneous Orchestration

In theory, subgraphs in the partition should be parallelized. However this is impractical on real hardware as cores and GPUs are limited resources. Instead, we take a heterogeneous approach through OpenMP. Each round, we launch T threads. Thread 0 is reserved for CUDA orchestration, while the other $T - 1$ threads participate in parallelizing the sparse parts of the graph with the MP strategy. We used OpenMP's parallel sections to execute different and non-iterative blocks of code simultaneously on multiple threads. We wrapped the two strategies in sections, which spawns two threads initially. One goes to work on CU, while the other spawns an additional `num_cores - 2` threads and work on MP cooperatively.

Our reason for choosing the `section` pragma is that it enables blocking within the same thread group. MP have to use barriers to sync across a few places within a contraction round, and `section` achieves this without interfering with the CU thread, as all barrier operations are section-local.

Latency Hiding

Heterogeneous orchestration also allows latency hiding for CU. Normally, the CPU stalls when copying memory from or to the GPU. With the heterogeneous approach, only one thread (the thread responsible for CUDA orchestration) is stalled, while other CPU threads continue to make progress in the MP strategy, hiding the GPU's fixed cost.

4 Results

Unless otherwise stated, all CPU experiments were run on CMU GHC machines using 8 OpenMP threads. We report wall-clock execution time and speedup relative to sequential Kruskal. We vary graph family, graph size and number of repeated MST rounds in our input. For repeated-round experiments, graph topology is fixed and edge weights are regenerated each round.

4.1 OpenMP

Speedup

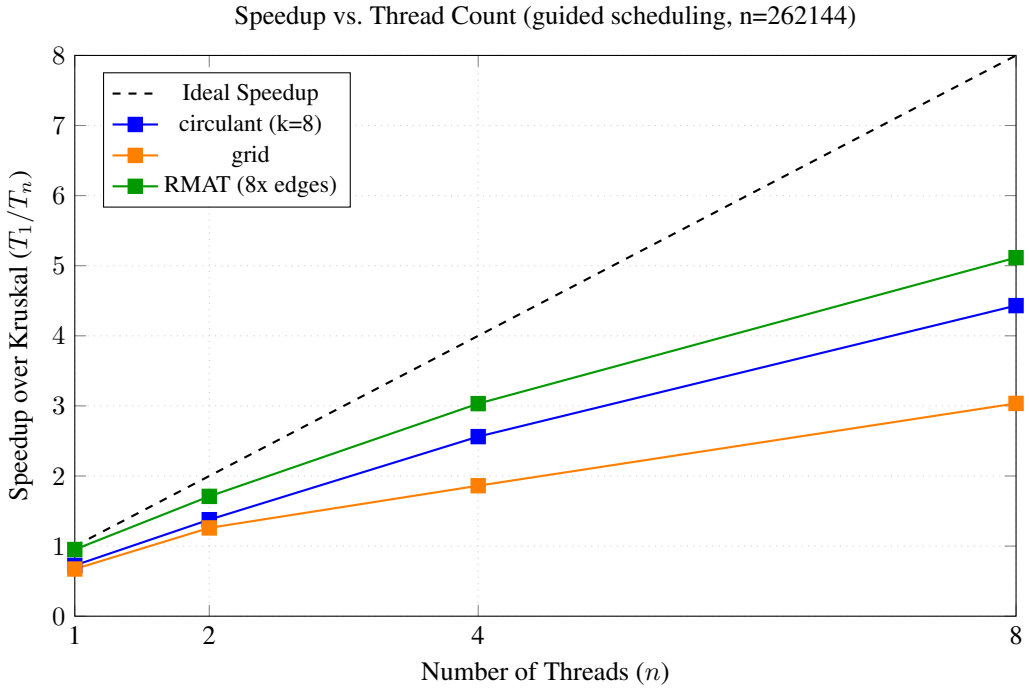


Figure 3: Speedup of OpenMP Borůvka (guided scheduling) over sequential Kruskal ($n = 262144$)

The performance of the edge-parallel OpenMP Borůvka implementation varies significantly based on graph topology, with RMAT scaling the best (5.12× at 8 threads), followed by Circulant (4.43×), and Grid scaling the worst (3.04×). This difference is tied to how component sizes grow during the algorithm. RMAT graphs has a highly skewed, power-law degree distribution; massive "hub" vertices causes many individual vertices to rapidly collapse into a few large components in the very first rounds. This drastically reduces the number of active components and valid edges, minimizing the number of contraction rounds and global synchronization barriers. Circulant graphs are strictly regular and offers good initial load balancing, but their localized connectivity means components grow at a steadier and more uniform rate. Grid graphs scale poorly because they are sparse (degree of 4), and its long diameter results in a

higher upper-limit of rounds. Consequently, the Grid graph requires more Borůvka contraction rounds, inducing more synchronization sites and denser CAS attempts in the union find (leads to high contention).

Synchronization

As synchronization stall is the bottleneck for MP, we used VTune to benchmark the time spent on synchronization (lock acquisition, atomic operations, etc.).

The results demonstrate that union-find with half-path compression has marginally less sync time at high thread count. We chose half-path compression as the default option used in our MP implementation from this benchmark result.

Atomic (Half Path Compression)				Atomic (Full Path Compression)			
T	Total (s)	Sync (s)	Sync %	T	Total (s)	Sync (s)	Sync %
1	15.19	3.19	21.0%	1	17.77	3.84	21.6%
2	8.28	3.57	21.5%	2	10.05	4.18	20.8%
4	4.37	3.86	22.1%	4	5.35	4.26	19.9%
8	2.81	4.58	20.4%	8	3.58	6.15	21.5%
16	2.05	6.29	19.1%	16	2.29	7.56	20.6%
32	1.80	6.81	11.8%	32	1.98	8.85	13.9%

Table 2: Sync Time of Path Compression Strategies over Thread Count (T)

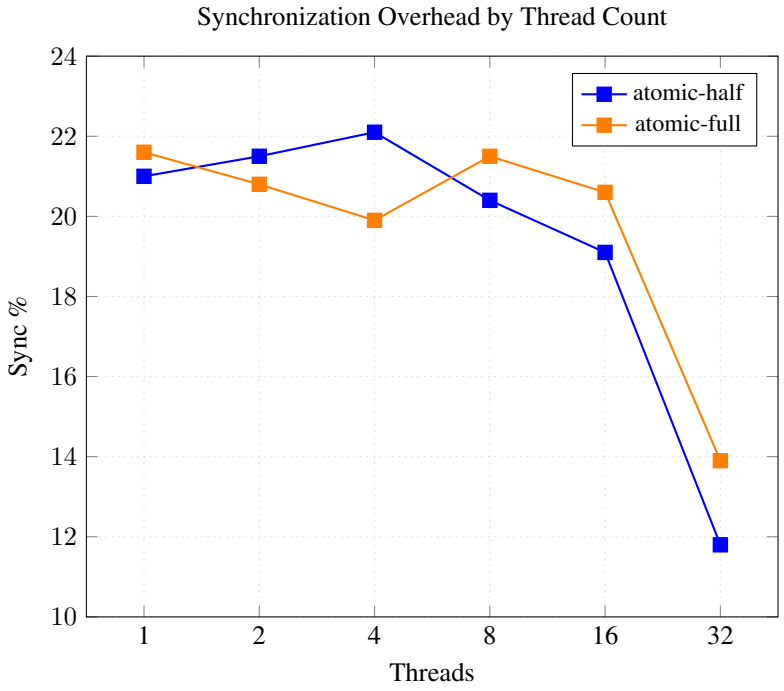


Figure 4: Synchronization Overhead over Thread Count (Dense $n = 16384$)

The results show that while full path-compression yields shorter path and better amortized lookup overhead from an algorithmic standpoint, it can be suboptimal when used in a parallel runtime due to more frequent CAS operations. In a sequential union-find, `atomic-full`'s aggressive compression is usually good because it flattens trees more aggressively and improves future lookups. But in a parallel implementation, the frequent CAS operations (`atomic-full` has $2\times$ more component relabeling than `atomic-half`) causes more contention and retries, as well as false sharing cache lines of the union-find's memory.

Notably, the sync time percentage drops off at higher thread counts. This is likely due to memory access time dominating the runtime at higher thread count. While more threads lead to more atomic operator contention, the impact on memory access stalls due to false sharing increases faster and dominates the runtime. Another overhead at high thread count is OpenMP's fixed parallelization overhead. The increase of these two overheads dilute the sync stall time at high thread count.

This also reveals that the major obstacle to further speedup in MP is sync time. Almost 20% of runtime is spent on CAS contention. This is not trivially avoidable due to its dependence on graph structure and contraction pattern.

Cache performance

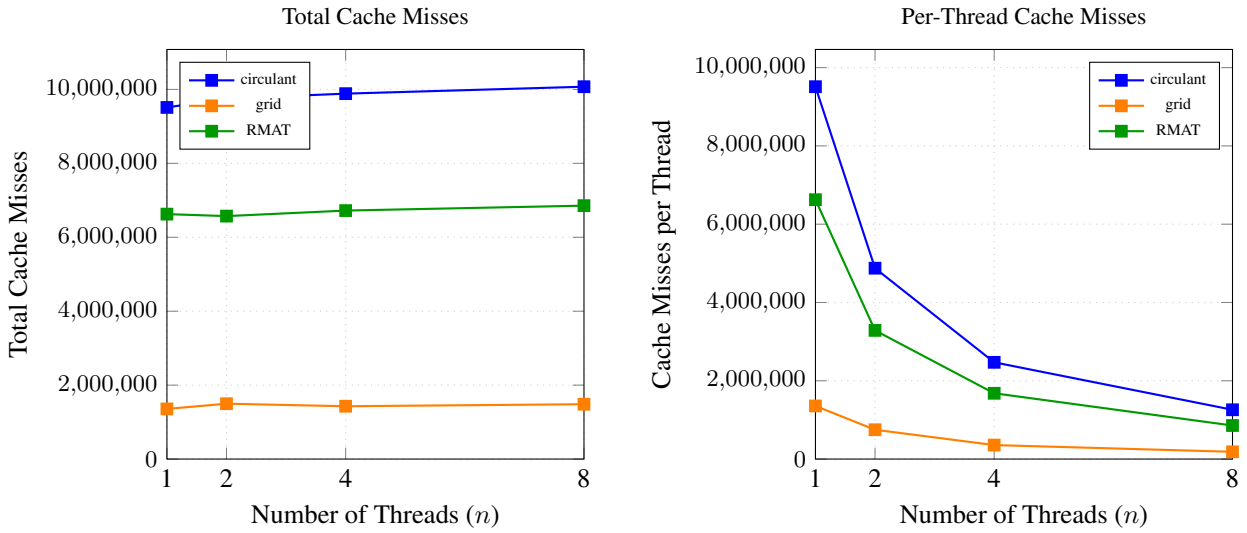


Figure 5: Total cache misses (left) and per-thread cache misses (right).

The cache miss data reveals that while the total number of cache misses remains relatively flat as thread count increases, the per-thread cache misses drop proportionally, indicating that the memory penalty is being divided among the available cores. The high volume of cache misses across all graphs is primarily driven by the Union-Find (`uf.find`) operations. During the edge-evaluation phase, threads must chase pointers up the `parent_array` to locate component roots. Because edges connect arbitrary vertices, these memory accesses jump unpredictably across the array, destroying spatial locality and resulting in frequent cache thrashing. The total miss count reflects the graph's structural density and degree of pointer-chasing: Circulant has the highest total misses due to its high uniform degree and deep, localized tree structures. RMAT, despite having many edges, exhibits fewer total misses than Circulant because its hub vertices act as universal roots. Once the hubs are cached, subsequent `uf.find` traversals hit the same hot cache lines. Grid graphs have the lowest absolute miss count simply because they have much fewer edges to evaluate per round.

Scheduling

Graph	Static	Dynamic	Guided
circulant (k=8)	3.60×	0.12×	**4.43**×
grid	2.85×	0.15×	**3.04**×
RMAT (8x edges)	4.23×	0.24×	**5.12**×

Table 3: 8-thread speedup over sequential Kruskal ($n = 262144$)

The scheduling comparison table highlights the performance difference between static, dynamic, and guided OpenMP scheduling strategies. The dynamic schedule exhibits poor performance degradation (e.g., 0.12× for Circulant). This occurs because checking an edge and traversing a shallow Union-Find tree is computationally cheaper than the overhead of dynamic scheduling, where threads constantly lock the work queue to fetch their next task. Static scheduling eliminates this queueing overhead by pre-assigning fixed chunks, achieving better speedups (3.60× to 4.23×), but it suffers from load imbalance when one thread, assigned a chunk of edges belonging to the same connected component, finishes instantly, while other threads process complex Union-Find traversals to merge different components. Guided scheduling achieves the highest speedups across all graphs because guided scheduling starts with large chunks to minimize scheduling overhead and gradually shrinks the chunk size towards the end of the loop to mitigate load imbalance.

Sensitivity to Graph Size

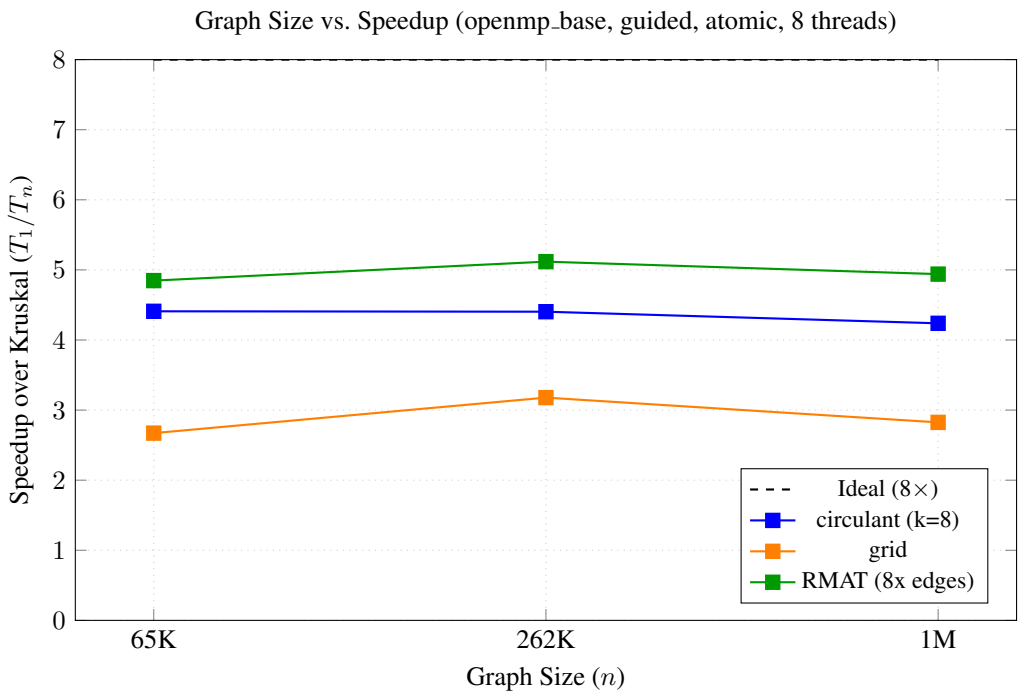


Figure 6: Speedup of OpenMP Borůvka (edge-parallel, guided scheduling, atomic union-find, 8 threads) over sequential Kruskal as graph size n varies from 65K to 1M.

The MP implementation is insensitive to graph size when tested across a 16x increase in graph size (from 65K to 1M vertices), indicating that parallel efficiency is dictated by graph topology rather than problem scale. Across all sizes, RMAT consistently yields the highest speedup (4.85x to 5.12x), followed by Circulant (4.24x to 4.41x) and Grid (2.67x to 3.18x). This constant scaling occurs because the ratio of useful parallel work to sequential synchronization costs remains relatively stable as the number of vertices and edges grows. While one might expect larger graphs to better amortize thread startup and barrier overheads, the guided scheduling strategy already effectively mitigates these penalties even at smaller sizes like n=65536. Instead, the fundamental bottleneck is Borůvka's sequential round structure: each algorithmic round must completely finish and synchronize before the next begins. Consequently, the performance hierarchy remains tightly bound to structural properties.

This indicates that the effectiveness of our MP program is dependent on graph structure. This highlights the necessity of graph structure analysis in our hybrid algorithm.

Conclusion

Overall, MP demonstrated good scaling on skewed, irregular graphs due to less contraction rounds, making it good for execution on sparse graphs without clear structure and vertex degree consistency. MP's speedup is relatively invariant across graph size, showing that it works well on small graphs too. Benchmarks on sync stalls and scheduling strategy reveal that the optimal parameter for MP is `atomic-half` union-find implementation with `guided` scheduling when parallelizing over the edges. These will be our parameters used for MP in the Hybrid strategy.

4.2 CUDA

Runtime Analysis

To test the sensitivity of CU on different graph sizes, we benchmarked its result on complete (fully-connected) graph of varying sizes, against MP (with 8 cores) and baseline Kruskal for comparison.

n	Kruskal (s)	OpenMP $t = 8$ (s)	CUDA (s)	Tree cost
512	0.02	0.02	0.28	1423
1024	0.09	0.04	0.80	1758
4096	1.83	0.23	0.70	4170
16384	35.22	2.98	2.07	16383

Table 4: Runtime comparison of Kruskal, MP, and CU across selected graph sizes.

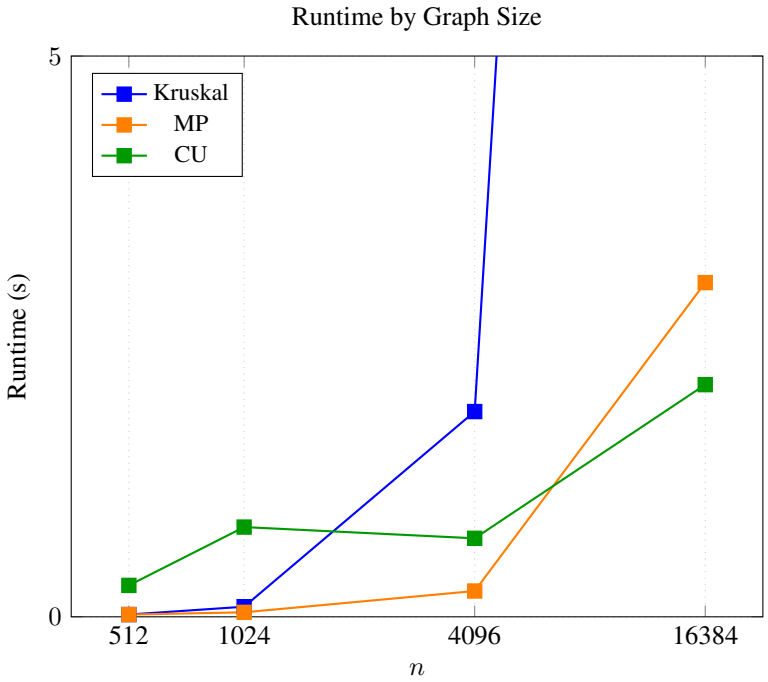


Figure 7: Runtime comparison of Kruskal, MP, and CU across selected graph sizes.

The main trend is that **CU has high overhead on small graphs, but scales better once the complete graph is large enough**. Since the graph is fully connected, the number of edges grows as $O(n^2)$.

At $n = 512$ and $n = 1024$, CU is slower than both Kruskal and MP. The GPU does not have enough work to hide kernel-launch overhead, memory transfers or device setup cost. MP performs better in this range because it has lower launch overhead and avoids host-device transfer costs. It becomes better at larger graph sizes. At $n = 16384$, CU is faster than MP, showing that the GPU has enough parallel work to amortize its overhead. From $n = 4096$ to $n = 16384$, MP runtime increases sharply, while CU grows smoothly.

This behavior matches our CU implementation. CU scans a dense adjacency matrix, which is inefficient for sparse graphs but efficient for dense to complete graphs. In a complete graph, every matrix entry is a valid edge, so the scan does little wasted work. The warp-per-vertex design also keeps control flow more uniform because most lanes perform similar edge checks, reducing warp divergence.

It's worth noting that our baseline Kruskal is fast for small n, but its runtime grows significantly by $n = 4096$ (even exceeding the plotted range afterward). This is expected because Kruskal must sort $O(n^2)$ edges on a complete graph, making edge sorting the dominant bottleneck as n increases.

Break-Even Analysis

When assigning strategies to a subgraph in the Hybrid implementation, when should we use MP vs. CU? We generated 10 graphs of varying edge density to find the threshold at which CU starts to out-perform MP. Each graph has $n = 16384$ vertices, and each vertex pair has probability p of containing an edge, with a total of pn^2 edges. The graph is a complete graph with $m = n^2$ edges at $p = 1.0$, and a discrete graph with no edges at $p = 0.0$.

The following result is obtained with MP on 8 cores, which is the maximum core count for GHC.

p	Edges	MP (s)	CU (s)	Speedup (MP / CU)
0.10	1.34×10^7	0.45	2.06	0.22
0.20	2.68×10^7	0.77	1.92	0.40
0.30	4.02×10^7	1.08	1.92	0.56
0.40	5.36×10^7	1.49	2.20	0.67
0.50	6.71×10^7	1.99	1.91	1.04
0.60	8.05×10^7	1.87	1.92	0.97
0.70	9.39×10^7	2.84	2.07	1.36
0.80	1.07×10^8	2.68	2.08	1.28
0.90	1.20×10^8	2.92	2.12	1.37
1.00	1.34×10^8	3.21	1.92	1.66

Table 5: Runtime comparison between MP and CU across different graph densities.

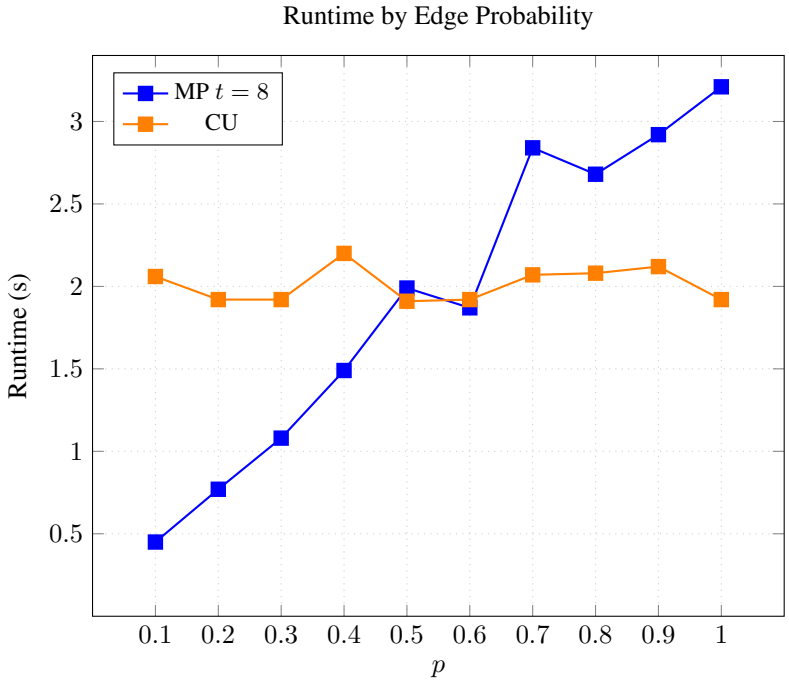


Figure 8: Runtime comparison between MP and CU across different edge probabilities ($n = 16384$).

The above result demonstrated that CU's runtime stays stable across all graph densities, exhibiting identical performance at $p = 0.1$ and $p = 1.0$, even when the edge count increases to $10\times$. MP's runtime starts off small on sparse graphs, and increase almost linearly as the graph gets denser with more edges. At around $p = 0.5$, MP's runtime start to dominate CU and perform worse consistently on $p > 0.5$ graphs. Therefore, fixing a vertex count n, CU outperforms MP when $m > n^2/2$. This is used as our heuristics for the Hybrid implementation.

At $p = 0.1$, MP takes 0.46s, while CU takes 2.07s, giving only $0.221\times$ speedup. This means CU is about $4.5\times$ slower than OpenMP here. The main reason is that CU scans the graph as a dense n^2 adjacency matrix, so its work is largely tied to the number of possible edges, as opposed to just the number of real edges. Even when p is small, each warp still scans across an adjacency row and checks many missing edges.

The CU runtime stays around 1.9-2.2s for almost all values of p. The `find_min` kernel assigns one warp per vertex and scans possible neighbors from the dense `adj` matrix. Because the scan length is approximately n per vertex regardless of how many edges actually exist, increasing p does not increase CU's work very much. The code has roughly the same scanning cost for sparse and dense graphs.

These results reveal that CU's speedup is limited by graph sparsity. It performed vertex-based contraction and waste warps even when a vertex has almost no neighbors. While the edge-scanning ensured good scaling as number of edges increase, its speedup as number of vertices increase is severely limited.

Conclusion

In short, the warp-per-vertex dispatch and parallel edge scanning make CU very efficient on dense graphs ($m > n^2/2$) and with size larger than 4096 to amortized the fixed cost on launch and host-device memory transfer. Moreover, CU is less sensitive to graph size, meaning we can expect its runtime to grow slowly with the size and density of the graph on large graphs, which is shown by its consistent runtime on the graph size and density results. This makes CU suitable for large, clique-like subgraphs in the Hybrid strategy. However, it suffers from significant fixed cost due to host-device memory transfer and CUDA launch time.

4.3 Hybrid

In the Hybrid implementation, we used K-Core decomposition to partition the graph, and then continuously altered the weights of the graph and recomputed its MST every round.

We used the R-MAT graph, which is skewed to simulate a graph similar to social network graphs, for this benchmark. It has highly skewed graph connections and densely connected cliques, which is suitable for testing the adaptability of our Hybrid algorithm. The graph size used has vertices $n = 49152$, with ≈ 16384 of them belonging to densely packed clusters that has density $m > n^2/2$.

Runtime on Varying Rounds

The following benchmark is from running 30 rounds of continuous MST generations. The one-shot strategies, CU and MP, calculates the MST of each round from the input graph directly, while Hybrid uses the precomputed graph decomposition to effectively assign one-shot strategies to subgraphs. We chose to use one-shot strategies instead of sequential Kruskal as benchmark baseline for Hybrid because we are testing for orchestration overhead and latency hiding, so runtime comparison should be based on

The result is obtained by running on 8 cores. One of them (thread 0) is dedicated for CUDA orchestration for launching CU on dense subgraphs, while the other 7 threads participate in MP's edge-parallel strategy.

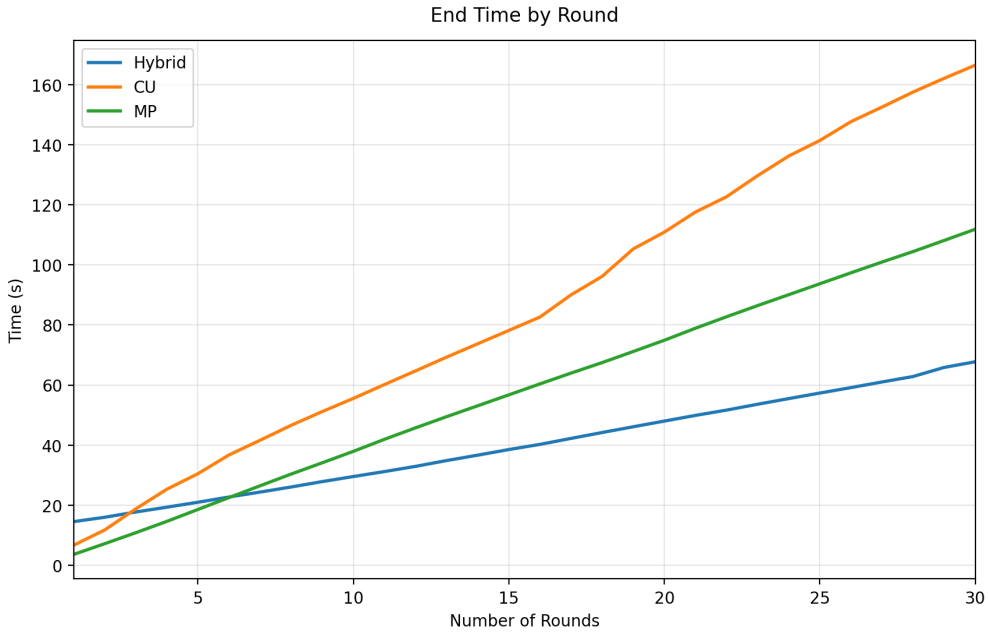


Figure 9: Accumulated Execution Time over Number of Rounds

The execution time graph shows excellent results. We were initially concerned about the 18.42 seconds of initialization time on Hybrid, which is caused by the K-Core decomposition initialization. However this fixed cost was quickly amortized away after 6 rounds and out-performs both the CU and MP one-shot strategies.

The per-round execution time (the slope of th graph) shows that Hybrid only takes $\approx 0.5\times$ the time it take other strategies to complete a round. Based on this speedup, we speculated three potential advantage of the Hybrid strategy:

- **Smaller CPU Working Set**: By delegating dense components of the graph to GPU, the MP implementation ran on the rest of the graph has significantly less edges to process, effectively reducing the CPU working set, allowing more data to fit inside each core's cache. This resulted in less cache misses, less memory stalls and higher arithmetic intensity.
- **Host-Device Latency Hiding**: The heterogeneous orchestration of the Hybrid strategy enables latency hiding when launching CUDA. Normally CUDA has a high launch stall, which is an unavoidable fixed cost. By using one core for CUDA and other cores for CPU parallelism, only one core halts on memory traffic to GPU, while the other cores continue to work. This hides the fixed cost of CU, which the one-shot strategy suffered from.
- **Uniform Graph Structure**: The R-MAT graph described in the above benchmark is highly irregular, with both dense and sparse regions. The sparse region stalls CU, which wastes warps on processing nonexistent edges; the dense region stalls MP with its high edges count. Hybrid avoid both by separating the dense and sparse regions and assigning them to CU and MP respectively.

Speedup Limitation

As $T - 1$ threads are still running MP, Hybrid suffers from the same bottleneck of high contention and sync stalls on high thread counts. Depending on how much concurrent contraction MP is trying to perform, Hybrid's speedup might collapse on similar graph patterns as MP. Moreover, if there isn't enough dense cores to schedule CU on, Hybrid regresses to MP's performance.

Initialization Time

One drawback of Hybrid is its high initialization time. K-Core decomposition through vertex peeling performs multiple iterations over the graph vertices, which scales linearly with the size of the graph. To test the initialization time's sensitivity to graph size, we timed the initialization time across complete (fully-connected) graphs of different sizes:

n	Edges	Decompose Time (s)
16	1.20×10^2	0.000037
64	2.02×10^3	0.000151
128	8.13×10^3	0.000509
512	1.31×10^5	0.009187
1024	5.24×10^5	0.037616
4096	8.39×10^6	0.625885
16384	1.34×10^8	10.952923

Table 6: K-Core Initialization Time over Graph Size

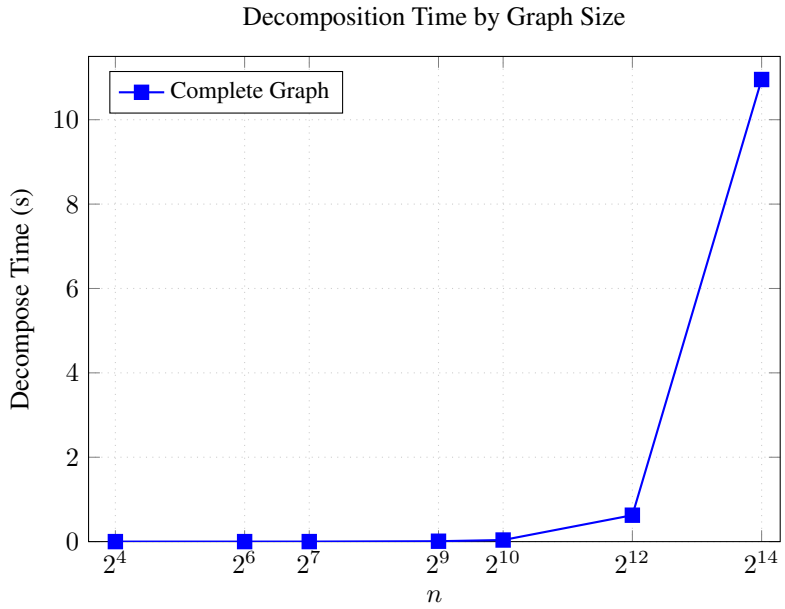


Figure 10: Decomposition time for complete graphs across different graph sizes.

On complete graphs, the number of edges is $O(n^2)$, so the observed initialization time appears quadratic in n. More generally, the peeling work scales with the number of vertices and edges processed by the decomposition. This is expected as the peeling algorithm repeatedly iteration through all vertices and edges. While this is unavoidable, it can be mitigated by either parallelizing the K-Core decomposition algorithm, or run the Hybrid at a large enough round number $r > 5$ to amortize the fixed cost.

5 Work Distribution

The work in this project is distributed evenly across Linxuan Ma and Joseph Wan. The total credit for the project should be divided evenly.

- **Linxuan**: One-shot strategies, CUDA optimizations, code base + unit tests, report writing.
- **Joseph**: Benchmark + evaluation, K-Core decomposition, dataset creation, poster writing.

References

[1] S. Hong, S. K. Kim, T. Oguntebi, and K. Olukotun, "Accelerating CUDA Graph Algorithms at Maximum Warp," in *Proceedings of the 16th ACM Symposium on Principles and Practice of Parallel Programming (PPoPP '11)*, 2011, pp. 267–276.

[2] V. Batagelj and M. Zaversnik, "An O(m) Algorithm for Cores Decomposition of Networks," *arXiv preprint cs/0310049*, 2003.