cuMIS: A Unified Scalable Framework for Computing Maximal Independent Sets on Trillion-Edge Graphs

Joseph Nke
Texas State University
San Marcos, TX, USA
pmh72@txstate.edu

Seungwha Kang
NVIDIA
Santa Clara, CA, USA
seunghwak@nvidia.com

Bradley Rees
NVIDIA
Santa Clara, CA, USA
brees@nvidia.com

Chul-Ho Lee
Texas State University
San Marcos, TX, USA
chulho.lee@txstate.edu

Abstract

This paper addresses the problem of computing a maximal independent set (MIS), defined as a set of vertices where no two vertices are connected by an edge and no additional vertex can be added without violating the independence property. While several GPU-accelerated algorithms exist to find the MIS efficiently, the problem remains challenging for graphs exceeding the memory of a single GPU. In this paper, we present cuMIS, a unified scalable framework for computing MIS on single-GPU, multi-GPU, and distributed multi-node configurations. cuMIS employs a data-driven approach that processes only an active set of undecided vertices for reduced memory access and a degree-aware workload distribution that mitigates imbalance and thread divergence. Our results show that cuMIS outperforms ECL-MIS and MG-MIS—the state-of-the-art single-GPU and multi-GPU baselines—achieving speedups of up to 6.5× and 156×, respectively, while maintaining comparable or superior solution quality. Finally, we demonstrate that cuMIS scales effectively to process trillion-edge graphs in distributed multi-node environments where existing approaches fail to operate.

CCS Concepts

• **Computing methodologies → Massively parallel algorithms**.

Keywords

Maximal independent sets, Trillion-edge graphs, Unified framework

ACM Reference Format:
Joseph Nke, Seunghwa Kang, Bradley Rees, and Chul-Ho Lee. 2026. cuMIS: A Unified Scalable Framework for Computing Maximal Independent Sets on Trillion-Edge Graphs. In *2026 International Conference on Supercomputing (ICS '26), July 06–09, 2026, Belfast, United Kingdom.* ACM, New York, NY, USA, 15 pages. https://doi.org/10.1145/3797905.3800529

1 Introduction

The maximum independent set problem is a fundamental challenge in graph theory. An independent set is formally defined as a subset of vertices in a graph where no two vertices are adjacent. This set becomes maximal if it cannot be augmented by any other vertex in the graph without violating this independence property. Note that the order in which vertices are processed can result in multiple distinct maximal independent sets for a single graph. The set with the largest cardinality among all possibilities is the maximum independent set, and finding this set is NP-hard and computationally expensive [20]. Despite being a relaxed version of the maximum independent set, a maximal independent set (MIS) serves multiple real-world applications. It is directly applied in resource allocation, scheduling, and circuit layout to identify non-conflicting tasks or components that can be processed concurrently [12, 34]. MIS also serves as a fundamental building block for graph algorithms such as graph coloring [11, 30] and minimum vertex cover [26], which enable applications in network analysis and computer vision [3, 10, 16].

Prior work on GPU-accelerated algorithms for computing the MIS identifies ECL-MIS [7] as a state-of-the-art parallel algorithm, achieving superior speed and larger independent sets compared to other parallel approaches such as CUSP [9], Pannotia [8], IrGL [32], Ligra [35], and PBBS [2]. Despite its high performance, ECL-MIS is fundamentally constrained by the memory capacity of a single GPU, which limits the size of the graphs that can be processed. One viable approach to overcome this limitation is to use unified virtual memory (UVM) [1, 17]. UVM enables the processing of significantly larger graphs by allowing the system to implicitly manage data movement between the host (CPU) and device (GPU). However, while UVM facilitates memory oversubscription (where data pages are swapped between CPU and GPU memory as required), this on-demand migration introduces considerable latency. This performance penalty is particularly severe when algorithms exhibit non-optimal memory access patterns and poor data locality.

Recently, Mongandampulath Akathoott *et al.* [29] propose MG-MIS as the first multi-GPU algorithm to improve the scalability of computing MIS on graphs by leveraging the aggregated memory of all GPUs. Their approach achieves superior performance compared to the UVM-based implementation of ECL-MIS for graph sizes exceeding a single device's capacity. This performance is experimentally demonstrated across multiple 'single-node' GPU configurations and different architectures. However, the approach remains fundamentally constrained by the total memory capacity available on a single node. Consequently, continued scaling for even larger

This work is licensed under a Creative Commons Attribution 4.0 International License.
ICS '26, Belfast, United Kingdom
© 2026 Copyright held by the owner/author(s).
ACM ISBN 979-8-4007-2522-7/26/07
https://doi.org/10.1145/3797905.3800529

graphs necessitates the adoption of UVM, which reintroduces the same performance and complexity challenges initially observed in single-GPU implementations, or the use of multi-node GPU environments, for which no MIS implementation currently exists.

Designing scalable multi-GPU graph algorithms, including MIS computation, presents two fundamental challenges. First, graph algorithms exhibit irregular, data-dependent memory access patterns that lead to poor cache locality and load imbalance. Second, multi-GPU execution introduces significant communication overhead. Although interconnects like NVLink offer improvements over PCIe, they remain slower than local memory access. This overhead is amplified in multi-node environments, where inter-node communication (e.g., via InfiniBand) introduces further latency. Therefore, efficient algorithms must carefully address data partitioning, load balancing, and communication pattern to minimize these costs.

To address these challenges, we introduce cuMIS, a high-performance algorithmic framework that integrates several novel techniques to maximize throughput for MIS computation on modern GPU architectures. First, we design cuMIS based on a synchronous, data-driven execution model that processes only a dynamic active set of undecided vertices, reducing memory access and improving resource utilization. It employs a read-only neighbor traversal, allowing a vertex to decide its own status (inclusion or exclusion) without waiting for updates from neighbors. This minimizes the memory contention and redundant scanning inherent in state-of-the-art algorithms, where a vertex may need to wait for its status to be updated by neighbors, often leading to serialization. This approach is particularly effective under UVM, as the read-only traversal keeps pages clean, eliminating device-to-host write-backs. Second, to handle skewed degree distributions (e.g., power-law), we implement a hybrid-granularity workload distribution. By dynamically mapping vertices to compute units (i.e., threads, warps, and blocks) based on their degree—using thread-level processing for low degrees and warp/block-centric execution for higher degrees—we effectively eliminate resource underutilization and load imbalance, maximizing hardware utilization across the entire degree spectrum. Third, we introduce an early-exit strategy that, during the first iteration, terminates neighbor traversal immediately upon encountering a higher-priority neighbor, drastically reducing the computational workload when the active set is largest. Finally, the framework extends seamlessly to multi-node multi-GPU environments through a scalable 2D partitioning strategy. By effectively balancing workload distribution and mitigating communication bottlenecks, we demonstrate that our framework enables scalability to trillion-edge graphs on hundreds of GPUs.

We first conduct comprehensive benchmarks on 15 large-scale Graph500 synthetic graphs across NVIDIA Volta and Hopper architectures to demonstrate the superiority of cuMIS over state-of-the-art baselines. cuMIS outperforms ECL-MIS by up to 5× with comparable solution quality and achieves speedups of up to 66× over MG-MIS while consistently producing larger independent sets. Furthermore, cuMIS on a single GPU with UVM often rivals or even outperforms MG-MIS on multiple GPUs, effectively mitigating the overhead associated with memory oversubscription. We also evaluate cuMIS on four large real-world graphs spanning social, road, and web topologies. In these cases, cuMIS achieves speedups of up to 6.5× over ECL-MIS on a single GPU and up to 156× over MG-MIS in multi-GPU configurations, while maintaining comparable or superior solution quality. Beyond single-node configurations, cuMIS scales to 256 GPUs across 32 nodes, successfully processing trillion-edge graphs that exceed the memory capacity of any single-node system.

Our contributions can be summarized as follows:

- We present cuMIS, a highly efficient and scalable unified algorithmic framework for computing MIS on single-GPU (with and without UVM), multi-GPU, and distributed multi-node configurations.
- We propose several novel techniques in cuMIS, including a data-driven execution model with read-only neighbor traversal, a degree-aware workload distribution, and an early-exit strategy for efficient workload reduction.
- We demonstrate that cuMIS significantly outperforms single-GPU and multi-GPU state-of-the-art baselines while often achieving better solution quality.
- We further demonstrate the scalability of cuMIS to trillion-edge graphs (Scale 35, 34.4 billion vertices) across 256 GPUs, a scale unattainable by existing algorithms.

2 Background

2.1 Preliminaries

For an undirected graph $G = (V, E)$, where V is the vertex set and E is the edge set, an MIS is a subset $I \subseteq V$ that satisfies two key properties. First, any two vertices u and v in I are not adjacent to each other, i.e., $(u, v) \notin E$. Second, I cannot be augmented further by any vertex $w \in V \setminus I$ without violating the "independent set" property; that is, every vertex outside the set is adjacent to at least one vertex in I. Let $N(v)$ denote the set of neighbors of vertex v, and $\deg(v)$ be the degree of v (i.e., $\deg(v) = |N(v)|$). Furthermore, let $\pi(v)$ be the priority value assigned to v. We assume that priority values are distinct for all vertices (i.e., $\pi(u) \neq \pi(v)$ for $u \neq v$) to ensure a strict total ordering and that the graph contains neither self-loops nor multi-edges.

2.2 Related Work

Since the seminal work of Karp and Wigderson [24] and Luby [27], parallel algorithms for computing the MIS have evolved significantly. In particular, Luby's priority-based approach, which assigns priority values to vertices and determines their inclusion in the independent set through local neighborhood comparisons, has been widely adopted in modern parallel implementations. CUSP [9] computes the MIS using sparse matrix operations via the Thrust library. Pannotia [8] provides an OpenCL implementation that directly processes graphs in compressed sparse row (CSR) format. IrGL [32] generates optimized CUDA kernels from a graph-specific language through compiler transformations. On shared-memory CPUs, Ligra [35] offers a lightweight framework supporting compressed representations, while PBBS [2] introduces an early-termination strategy that exits neighbor traversal upon finding a higher-priority vertex. Burtscher *et al.* [7] later proposed ECL-MIS, demonstrating superior performance over CUSP, IrGL, Pannotia,

Algorithm 1: ECL-MIS

```
Input  : G(V, E), {π(v), v ∈ V}
Output: I
1  I ← ∅;
2  for each v ∈ V in parallel do
3      while true do
4          if π(v) ≠ −∞ and π(v) ≠ ∞ then
5              include ← true;
6              for each u ∈ N(v) do
7                  if π(v) < π(u) then
8                      include ← false;
9                      break;
10             if include then
11                 π(v) ← ∞;
12                 I ← I ∪ {v};
13                 for each u ∈ N(v) do
14                     π(u) ← −∞;
15         else
16             break;
```

Algorithm 2: Basic algorithm of cuMIS

```
Input  : G(V, E), {π(v), v ∈ V}
Output: I
1  I ← ∅;  V_r ← V;
2  while true do
3      for each v ∈ V_r in parallel do
4          π_max ← max_{u∈N(v)} π(u);
5          if π_max = ∞ then
6              π(v) ← −∞;
7          else if π(v) > π_max then
8              π(v) ← ∞;
9              I ← I ∪ {v};
10     for each v ∈ V_r in parallel do
11         if π(v) = −∞ or π(v) = ∞ then
12             V_r ← V_r \ {v};
13     if |V_r| = 0 then
14         break;
```

and PBBS in both runtime and solution quality. Recently, Mongandampulath Akathoott *et al.* [29] introduced MG-MIS as the first multi-GPU algorithm for computing the MIS.

Beyond fundamental MIS computation, MIS algorithms are naturally leveraged to approximate the maximum independent set problem. Imanaga *et al.* [20, 21] repeatedly compute the MIS on a graph with randomized priority values, retaining the set with the largest cardinality. While they present a multi-GPU implementation, it processes an independent graph instance on each device rather than distributing a single graph across multiple GPUs for collaborative computation. In addition, variants of MIS have been defined and studied. For example, Distance-2 MIS [25] imposes a stricter constraint requiring that selected vertices share no common neighbors, enabling applications in algebraic multigrid and graph coarsening. Similarly, MIS computation is applied to k-mer graphs with edit-distance constraints for sequence analysis [28]. The computation of MIS is also extended to evolving graphs [37]. These domain-specific problems lie outside the scope of this paper; we focus on the efficiency and scalability of general-purpose MIS computation on massive graphs, scaling up to trillion edges.

2.3 ECL-MIS and MG-MIS

To proceed, we review the algorithmic characteristics of ECL-MIS [7] and MG-MIS [29], as they serve as the single-GPU and multi-GPU baselines in this work, respectively. ECL-MIS is an asynchronous, topology-driven algorithm based on Luby's priority-based approach. Each vertex v is assigned a priority value $\pi(v)$ based on degree with random perturbation, and its inclusion in the MIS is determined by comparing its priority against those of its neighbors. One thread is assigned to each vertex to perform such a comparison.

As shown in Algorithm 1, ECL-MIS proceeds in rounds. The inclusion or exclusion of a vertex v is indicated by setting its priority value to $\pi(v)=\infty$ (included) or $\pi(v)=-\infty$ (excluded). In each round, every vertex v checks if it is decided (included or excluded) or not. If undecided, it checks if any neighbor has a higher priority. If so, its status remains undecided for the current round. However, if $\pi(v)$ is strictly higher than the priorities of all its neighbors, v is added to the MIS (i.e., $\pi(v)$ is set to ∞). Consequently, all neighbors u in the neighbor set $N(v)$ are immediately excluded (i.e., $\pi(u)$ is set to $-\infty$), as they are adjacent to a selected vertex. While this approach avoids global synchronization barriers, it can introduce significant memory contention. This is because there may be multiple threads that attempt to update the status of the same vertex u to $-\infty$. Furthermore, since threads continue to be launched even after vertices are decided, this approach leads to substantial thread divergence and wasted computational cycles.

MG-MIS, the first distributed multi-GPU algorithm, operates similarly to ECL-MIS but adopts a coarse-grained, bulk-synchronous approach. Unlike ECL-MIS, MG-MIS assigns priorities randomly to avoid the communication overhead of computing vertex degrees across multiple GPUs. The graph is partitioned across GPUs, with each device responsible for a subset of vertices. While ECL-MIS immediately updates the status of each neighbor if it needs to be excluded, MG-MIS buffers status updates for remote neighbors. The algorithm proceeds in two distinct phases. In the local computation phase, each device identifies independent set candidates within its partition. This is followed by a global synchronization phase where buffers containing the IDs of decided vertices are exchanged between devices. Despite the scalability enabled by using multiple GPUs, a key limitation of this design is the fixed-size communication buffers. If a buffer fills up during a local round, threads stall, preventing them from sending status updates even after adding a vertex v to the MIS. This approach prevents buffer overflows but can delay convergence by forcing additional iterations solely to flush communication buffers.

3 cuMIS

In this section, we present cuMIS, a unified framework for computing MIS in single-GPU, multi-GPU, and multi-node environments. We first explain its basic algorithm. We then detail our degree-aware workload distribution strategy and efficient neighbor traversal technique, designed to maximize system performance.

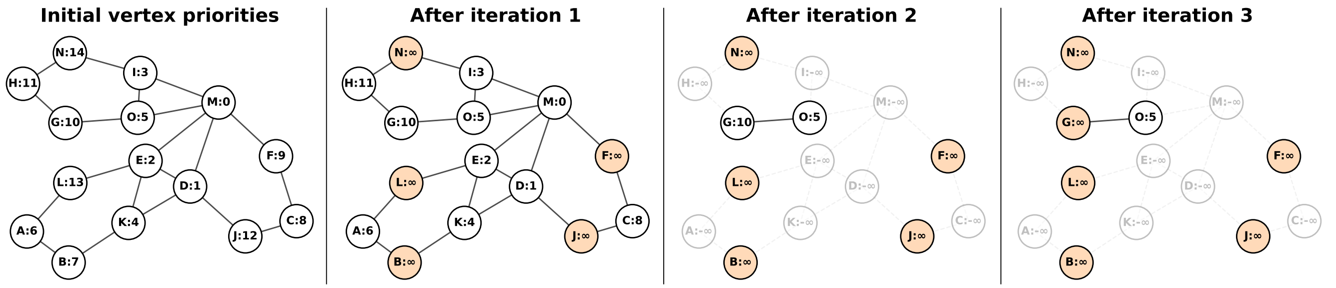


Figure 1: The execution of cuMIS on an example graph.

3.1 Basic Algorithm

Algorithm 2 outlines the basic algorithm of cuMIS. Unlike ECL-MIS, cuMIS is a synchronous, data-driven algorithm. This means that the algorithm runs in synchronous rounds and maintains an active vertex set, denoted as V_r, containing all undecided vertices in each round. Furthermore, unlike MG-MIS, cuMIS assigns priorities based on vertex degrees. Despite the communication overhead in multi-GPU environments, we compute vertex degrees to enable both the priority assignment and the degree-aware workload distribution.

Initially, each vertex v is assigned a unique priority value $\pi(v)$ based on its degree and ID. Specifically, we assign lower priority values to higher-degree vertices; when vertices have the same degree, they are further ordered according to their IDs. This assignment effectively gives lower priorities to high-degree hubs, reducing the likelihood of their inclusion in the MIS while preserving their lower-degree neighbors, thereby promoting a larger MIS size. In the main iterative loop (Lines 2–14), a vertex v joins the MIS I and is removed from V_r if its priority is higher than those of all its neighbors (Lines 7–9 and 11–12). Consequently, in the subsequent iteration, neighbors of the newly added vertex detect its presence (because the observed maximum neighbor priority is infinity, i.e., $\pi_{\max} = \infty$) and exclude themselves from V_r by setting their priorities to $-\infty$ (Lines 5–6 and 11–12). The algorithm terminates when V_r becomes empty (Lines 13–14).

It is worth noting that cuMIS exhibits the characteristics of an unordered, data-driven, local computation algorithm in the Tao analysis [33]. The inclusion or exclusion of a vertex v is not predetermined by the topology but is driven entirely by the dynamic values of its neighborhood, and the operation on vertex v is limited to the vertex and its immediate graph neighbors. Furthermore, in cuMIS, a change in the status of v does not immediately trigger a cascade of potential updates over its neighborhood and beyond; instead, the updates are postponed to the subsequent synchronous round. This approach serves two key purposes. First, it ensures global consistency. Deferring updates guarantees that all decisions in a round rely on a consistent snapshot of priority values, preventing race conditions and ensuring that the strict dominance rule is never violated by transient states. Vertices with locally maximal priorities (local maxima) form a non-conflicting set and can safely join the MIS in parallel without synchronization, since no two neighbors can both be local maxima simultaneously due to the uniqueness of priority values. Second, it facilitates efficient frontier management by focusing solely on the remaining active set V_r of undecided vertices, thereby eliminating redundant checks on decided vertices and leading to faster convergence of the algorithm.

Consider the leftmost graph in Figure 1 as an example, where each vertex is labeled with a $\langle \text{ID} : \text{priority} \rangle$ pair. The subsequent graphs in Figure 1 illustrate the execution of cuMIS on this example. cuMIS proceeds in synchronous rounds. The priority of each vertex v eventually becomes $\pi(v) = \infty$ (included) or $\pi(v) = -\infty$ (excluded). In the first iteration, vertices `B`, `F`, `J`, `L`, and `N` identify themselves as local maxima and join the MIS. In the second iteration, their neighbors (i.e., `A`, `C`, `D`, `E`, `H`, `I`, `K` and `M`) observe that they are now adjacent to a vertex in the MIS and are consequently excluded. In the third iteration, vertex `G` becomes the new local maxima within the active set V_r and joins the MIS. In the fourth iteration, the last remaining vertex (i.e., `O`) is excluded, completing the process.

3.2 Degree-Aware Workload Distribution

The parallel implementation of cuMIS on GPUs faces technical challenges due to the inherent irregularity of real-world graphs, which often leads to load imbalance and thread divergence. Since cuMIS is a data-driven algorithm, a critical challenge is how to efficiently process the dynamic active vertex set V_r in each round on the GPU's single instruction, multiple threads (SIMT) architecture. The challenge arises because the active set shrinks rapidly and becomes increasingly sparse as the number of rounds increases. While this approach significantly reduces the total workload compared to topology-driven approaches like ECL-MIS that iterate over the entire vertex set in every round, the workload becomes highly irregular and dynamic, evolving from a dense graph traversal in the first iteration to sparse updates in later rounds.

The core problem lies in efficiently mapping the active (or undecided) vertices in each round to compute units (i.e., threads, warps, and blocks) to avoid severe thread divergence and uncoalesced memory accesses. A naive thread-centric approach, where a single thread processes a vertex and its adjacency list (for its immediate neighbors), suffers from significant drawbacks. For high-degree vertices, a single thread serializes the neighbor traversal, leading to extreme load imbalance. Furthermore, since the neighbor lists of consecutive vertices usually reside in non-contiguous memory regions, threads in a warp accessing neighbors of different vertices generate scattered memory requests, drastically reducing effective memory bandwidth [19].

The warp-centric approach is a common solution for irregular graph processing. In this approach, a warp (typically 32 threads) is assigned to process a single vertex. This allows the threads within

the warp to traverse the vertex's neighbor list in parallel, ensuring that memory accesses are coalesced into fewer transactions. While the warp-centric approach significantly outperforms the thread-centric one on power-law graphs by mitigating serialization and improving memory efficiency, it is not without flaws. It imposes a rigid granularity that is suboptimal for the extremes of the degree distribution. For the vast majority of vertices with low degrees (e.g., degree < 32), allocating a full warp results in substantial underutilization, as most threads remain idle. Conversely, for massive "hub" vertices with millions of edges, a single warp lacks the parallelism required to process them all. Although a block-centric approach—assigning an entire thread block to a vertex—can handle these hubs, applying it uniformly is prohibitive, as it would severely underutilize resources for the millions of low-degree vertices.

To achieve efficient resource utilization across the entire degree spectrum, we propose a hybrid-granularity processing strategy [22]. Instead of using a uniform execution model, we dynamically map the GPU's hierarchical compute units to vertices based on their degree. Our approach begins with a degree-aware vertex renumbering phase. We sort vertices in descending order of degree, grouping them into contiguous segments. This reordering provides the segment metadata (i.e., start and end indices of each degree range), allowing the system to dispatch the most appropriate kernel without runtime branching overhead. Specifically, we divide the degrees into three ranges: high, intermediate, and low. Let τ_h denote an upper threshold to separate high and intermediate degrees, and τ_l denote a lower threshold to separate low and intermediate degrees.

High-degree vertices with degrees greater than τ_h are processed by entire thread blocks, maximizing parallelism and register usage for the most computationally intensive parts of the graph. For the intermediate range of degrees between τ_h and τ_l, we employ the warp-centric kernel to preserve the benefits of coalesced access and load balancing. Finally, the majority of low-degree vertices with degrees less than τ_l are processed by individual threads. Since these vertices have few neighbors, the serialization penalty is negligible, allowing us to efficiently recover the massive parallelism needed for millions of such vertices. This three-tiered approach ensures that no compute resources are wasted on small tasks while providing sufficient horsepower for heavy workloads, effectively minimizing thread divergence and maximizing throughput. While τ_h and τ_l are configurable parameters, we set $\tau_h = 1024$ and $\tau_l = 32$ in this work.

3.3 Efficient Neighbor Traversal

The degree-aware workload distribution ensures that the expensive operation of traversing neighbors is handled by the most appropriate compute unit. To further reduce the computational cost, we introduce an early-exit optimization for the first iteration.

In the standard execution of cuMIS, a vertex v checks all its neighbors to determine its status. As shown in Algorithm 2, deciding inclusion or exclusion requires comparing $\pi(v)$ against the maximum neighbor priority $\pi_{\max}$. However, during the first iteration, if a vertex encounters any neighbor with a higher priority than itself, it immediately knows it cannot join the MIS in the current round. We leverage this by terminating the traversal early, which drastically reduces memory traffic during the algorithm's most expensive phase (when the active set is largest). This optimization is conceptually analogous to direction-optimizing BFS [5], which switches to a bottom-up traversal to skip edge checks when the frontier is large. Note that this early-exit strategy is a standard feature in topology-driven algorithms like ECL-MIS and MG-MIS.

However, unlike ECL-MIS and MG-MIS, we restrict this optimization to the first iteration. In subsequent rounds, a vertex must identify if any neighbor has joined the MIS (indicated by $\pi_{\max} = \infty$) to correctly exclude itself. An early exit triggered by a neighbor with merely higher finite priority could prevent the detection of a different neighbor already in the MIS. Missing this information would leave the vertex undecided, forcing unnecessary additional iterations and delaying convergence.

4 Implementation Details

In this section, we detail the implementation of cuMIS, which seamlessly supports single-GPU, multi-GPU, and distributed multi-node configurations, including support for UVM. Algorithm 3 outlines the unified algorithm that governs all these configurations. This single-codebase approach eliminates the need for separate implementations, thereby streamlining development and enhancing portability. This stands in contrast to ECL-MIS and MG-MIS, which are separately developed for single-GPU and multi-GPU environments, respectively.

4.1 Single GPU Implementation

The algorithm takes two inputs: an undirected graph in CSR format (where each undirected edge is stored as two directed edges) and a boolean flag, `mgpu`, indicating whether multiple GPUs are used. It begins by computing the degree of each vertex, a prerequisite for both priority assignment and degree-aware workload distribution. A thread is assigned to each vertex to compute its degree in parallel, achieved by simply reading the difference between two consecutive offset values in the CSR representation. Next, we leverage parallel radix sort to order vertices based on their degrees and IDs—first by degree in descending order, and then by ID in ascending order. We reassign the ID of each vertex v, denoted as $id(v)$, to correspond to its rank in this sorted list. The CSR representation is also permuted to align with the new vertex IDs. Furthermore, we classify vertices into three groups (high, intermediate, and low degree) according to the upper and lower thresholds, τ_h and τ_l. This grouping is essential for the degree-aware workload distribution.

Subsequently, the algorithm executes the initialization phase. The priority value of each vertex v is set to its new ID, i.e., $\pi(v) = id(v)$ (Line 3). Any zero-degree vertices are immediately identified as isolated and added to the MIS (Lines 4–5). The algorithm then proceeds to the main loop. For a single GPU, each iteration consists of two primary steps. First, neighbor traversal is performed to identify the highest-priority neighbor using ComputeMaxNeighbor, as detailed in Algorithm 4. This kernel is launched according to our degree-aware workload distribution; that is, each vertex is processed by an appropriate compute unit (thread, warp, or block) corresponding to its degree group. During the first iteration, the early-exit optimization is employed to terminate traversal immediately upon discovering a higher-priority neighbor (Lines 10–11 of Algorithm 4). Second, based on the traversal result, the inclusion or exclusion of each vertex $v \in V_r$ is determined (Lines 15–20), as

Algorithm 3: cuMIS

```
Input  : G(V, E) , mgpu (boolean flag)
Output : I
1  I ← ∅;  V_r ← V;
   // Initialization phase
2  for each v ∈ V in parallel do
3      π(v) ← id(v);
       // Find isolated vertices
4      if deg(v) = 0 then
5          π(v) ← ∞;  I ← I ∪ {v};  V_r ← V_r \ {v};
6  itr ← 0;
7  V_pre ← ∅;  // Vertices processed in previous iteration
8  while true do
       // Priority synchronization
9      if mgpu then
10         if itr = 0 then
               // Sync all vertices in V_r
11             SynchronizePriorities(V_r, π);
12         else
               // Sync processed vertices
13             SynchronizePriorities(V_pre, π);
       // Compute the maximum neighbor priority
14     π_max ← ComputeMaxNeighbor(V_r, π, mgpu, itr);
       // Decision making: Add v to MIS or discard
15     for each v ∈ V_r in parallel do
16         if π_max(v) ≥ ∞ then
17             π(v) ← −∞;  // Discard v
18         else if π(v) > π_max(v) then
19             π(v) ← ∞;
20             I ← I ∪ {v};  // Add v to MIS
       // Update remaining and processed vertices
21     V_pre ← ∅;
22     for each v ∈ V_r in parallel do
23         if π(v) = −∞ or π(v) = ∞ then
24             V_r ← V_r \ {v};
25             V_pre ← V_pre ∪ {v};
       // Termination check
26     if mgpu then
27         n ← GlobalSum(|V_r|);  // Sum across all GPUs
28     else
29         n ← |V_r|;
30     if n = 0 then
31         break;
32     itr ← itr + 1;
33 return I;
```

explained in Section 3.1. Finally, the active vertex set V_r is updated by removing vertices that have just been included in or excluded from the MIS (Lines 22–25). The algorithm terminates when V_r becomes empty.

4.2 Unified Virtual Memory (UVM) Support

We utilize NVIDIA's RAPIDS Memory Manager (RMM) [18] to seamlessly support UVM, enabling the exact same single-GPU code

Algorithm 4: ComputeMaxNeighbor

```
Input  : V_r, {π(v), v ∈ V_r}, mgpu, itr
Output : {π_max(v), v ∈ V_r}
1  for each v ∈ V_r in parallel do
2      π_max(v) ← −∞;  // Initialize to lowest value
3      for each u ∈ N(v) in parallel do
4          if mgpu and u is remote then
5              priority ← π_c(u);  // if u is remote neighbor
6          else
7              priority ← π(u);  // if u is local neighbor
8          if priority > π_max(v) then
9              π_max(v) ← priority;
10             if itr = 0 and priority > π(v) then
                   // Early stop: found higher priority
                      neighbor
11                 break;
12 return {π_max(v), v ∈ V_r};
```

to scale to out-of-core graphs without modification. By abstracting low-level allocation, RMM allows transparent paging between host and device memory. Our data-driven approach is highly effective with UVM because it maintains and processes only the active set V_r, which shrinks over time, thereby minimizing page thrashing. Unlike topology-driven scans that process the entire vertex set in every round, our approach ensures that once a vertex is decided (included or excluded), its associated memory pages are rarely accessed again. Thus, this drastically reduces data transfer volume by over 150× compared to ECL-MIS for large graphs, as we demonstrate in Section 5.

In addition, the degree-based vertex reordering for the degree-aware workload distribution significantly improves spatial locality for the graph. Since CSR offsets are indexed by vertex IDs, grouping vertices with similar degrees into consecutive ID ranges ensures that their metadata resides in contiguous memory pages. Such contiguity means that fetching a page for one vertex typically retrieves the offsets for the neighbor lists of subsequent vertices in the batch. This effective prefetching reduces data transfer volume and mitigates the memory bandwidth bottleneck, allowing the system to efficiently process graphs that exceed physical memory capacity.

4.3 Multi-GPU Implementation

Our multi-GPU implementation for both single-node and multi-node environments preserves the core single-GPU logic of the algorithm while introducing mechanisms for data distribution and status synchronization. Unlike MG-MIS, which employs traditional block-based 1D vertex partitioning where contiguous vertex ranges and all their incident edges are assigned to each GPU, we adopt a hash-based partitioning strategy. Vertices are mapped to GPUs via a hash function, and edges are distributed according to the hash of their destination vertex. This randomized assignment prevents high-degree hubs from being concentrated on a single device, leading to superior load balance compared to contiguous partitioning for scale-free graphs [6, 13]. Let V_i denote the set of vertices assigned to GPU i, and E_i be the set of edges stored on GPU i.

Algorithm 5: ComputeGlobalDegrees

Input : $G(V, E)$, mgpu
Output: $\{\deg(v), v \in V\}$
1 **if not** mgpu **then**
2 **for each** $v \in V$ **in parallel do**
3 $\deg(v) \leftarrow |N(v)|$;
4 **else**
// Each GPU i executes the following independently
// Compute the local degrees of vertices in GPU i
5 **for each** $v \in V_i$ **in parallel do**
6 $\deg(v) \leftarrow |N_i(v)|$;
// Send V_i to GPU j $(j \neq i)$
7 **for each** $j \neq i$ **do**
8 Send V_i to GPU$_j$;
// Receive V_j from GPU j and compute partial degrees
9 **for each** $j \neq i$ **do**
10 Receive V_j from GPU$_j$;
11 **for each** $v \in V_j$ **in parallel do**
12 $\deg_{j,i}(v) \leftarrow |N_i(v)|$;
// Send partial degrees back to owner
13 Send $\{\deg_{j,i}(v), v \in V_j\}$ to GPU$_j$;
// Receive partial degrees from GPU j
14 Receive $\{\deg_{i,j}(v), v \in V_i\}$ from GPU$_j$;
// Aggregate partial degrees from all GPUs
15 **for each** $v \in V_i$ **in parallel do**
16 **for each** $j \neq i$ **do**
17 $\deg(v) \leftarrow \deg(v) + \deg_{i,j}(v)$;
18 **return** $\{\deg(v), v \in V\}$;

Since the hash-based distribution spreads the edges of each vertex across multiple GPUs, no single GPU can determine the degree of a vertex locally. A coordinated global degree computation is therefore required. Algorithm 5 outlines the procedure for this computation on multiple GPUs. Each GPU first computes local degree contributions from the edges it owns (Lines 5–6) and broadcasts its assigned vertex set to peers (Lines 7–8), where $N_i(v)$ indicates the set of neighbors of vertex v residing on GPU i. Each peer computes partial degrees for the received vertices using its local edge partition and returns these contributions to the owner GPUs (Lines 9–14), where $\deg_{j,i}(v)$ denotes the partial degree of $v \in V_j$ computed by GPU i. The owner GPU of each vertex then aggregates these partial degrees to obtain the global degree (Lines 15–17). With global degrees available, each GPU assigns global IDs to its local vertices. As in the single-GPU case, vertices are ordered first by degree in descending order and then by original vertex ID in ascending order. Here, each GPU owns a contiguous range of the global ID space, with ranges partitioned consecutively across GPUs. As a result, the ID of each vertex v, $id(v)$, becomes its global ID, and its priority is set to this new ID such that $\pi(v) = id(v)$.

Subsequently, the algorithmic steps remain consistent with the single-GPU case, operating on the local subgraph defined by V_i and E_i. Once priorities are assigned, the next key algorithmic operation is neighbor traversal, where each vertex must query the priorities of its neighbors to determine local maxima. In a multi-GPU setting, a naive approach would require fetching these priority values from remote GPUs on demand, incurring prohibitive latency due to excessive remote data transfer. To eliminate this overhead, cuMIS maintains "cached replicas" for boundary vertices. A vertex v owned by GPU i is replicated on GPU j if it is adjacent to any vertex owned by GPU j. Let $\pi_c(v)$ denote the cached priority of v on GPU j.

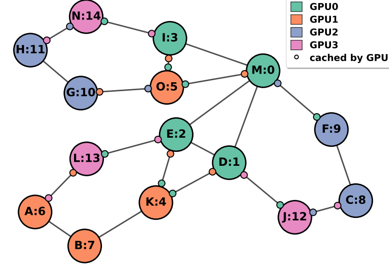


Figure 2: Vertex distribution in four GPUs with cached boundary priorities.

Algorithm 6: SynchronizePriorities (Multi-GPU only)

Input : V_i, $\{\pi(v), v \in V_i\}$
// Each GPU i executes the following independently
// Assume known remote partitions $\{V_j, j \neq i\}$
// Send updated priorities to remote GPUs
1 **for each** $v \in V_i$ **in parallel do**
// Send $\pi(v)$ to v's neighbors on GPU j $(j \neq i)$
2 **for each** $j \neq i$ **do**
3 **if** $N(v) \cap V_j \neq \emptyset$ **then**
4 Send $\pi(v)$ to GPU$_j$;
// Receive updated priorities from remote GPUs
5 **for each** $v \in V_i$ **in parallel do**
// Receive $\pi(u)$ from v's neighbor u on GPU j $(j \neq i)$
6 **for each** $j \neq i$ **do**
7 **for each** $u \in N(v) \cap V_j$ **in parallel do**
8 Receive $\pi(u)$ from GPU$_j$;
9 $\pi_c(u) \leftarrow \pi(u)$;

Consider an example graph as depicted in Figure 2. For example, vertex `M` (owned by GPU 0) is adjacent to `O` (GPU 1) and `F` (GPU 2). Copies of `M` are then maintained in the memory of GPU 1 and GPU 2 (indicated by the coral and blue cache dots). Similarly, `C` (GPU 2) is cached on GPU 3 due to its edge to `J`. Therefore, these cached replicas allow the ComputeMaxNeighbor kernel to read neighbor priorities from local memory, minimizing remote data transfer. As detailed in Algorithm 4 (Lines 4–7), the kernel transparently checks whether a neighbor u is local or remote and retrieves the priority value via $\pi(u)$ or $\pi_c(u)$, respectively, without stalling for communication.

Following this neighbor traversal, vertices determine their status based on the local maximum priority (Lines 15–20 of Algorithm 3). When the status of a vertex is updated, it must be synchronized across devices. Efficient synchronization of this information is critical. In the first iteration, all vertex priorities are broadcast globally to ensure a consistent initial state (Lines 10–11 of Algorithm 3).

Table 1: Information about the input graphs

Input	$\lvert V\rvert$	$\lvert E\rvert$	d_{avg}	d_{max}	Size (GB)
g-25s-2ef (25-2)	33,554,432	133,516,554	3.98	115,060	1.28
g-28s-2ef (28-2)	268,435,456	1,070,786,720	3.99	419,426	10.23
g-26s-10ef (26-10)	67,108,864	1,322,545,710	19.71	694,301	10.42
g-26s-11ef (26-11)	67,108,864	1,453,274,642	21.66	748,870	11.39
g-26s-12ef (26-12)	67,108,864	1,583,792,718	23.60	802,382	12.36
g-26s-14ef (26-14)	67,108,864	1,844,210,488	27.48	905,020	14.30
g-26s-16ef (26-16)	67,108,864	2,103,844,848	31.35	1,003,347	16.24
g-26s-18ef (26-18)	67,108,864	2,362,730,378	35.21	1,097,929	18.17
g-26s-20ef (26-20)	67,108,864	2,620,933,926	39.05	1,188,533	20.09
g-27s-10ef (27-10)	134,217,728	2,652,246,640	19.76	1,081,828	20.89
g-29s-2ef (29-2)	536,870,912	2,142,721,430	3.99	643,804	20.46
g-27s-16ef (27-16)	134,217,728	4,223,282,274	31.47	1,572,126	32.59
g-28s-8ef (28-8)	268,435,456	4,259,434,956	15.87	1,399,843	33.99
g-29s-4ef (29-4)	536,870,912	4,278,119,462	7.97	1,196,042	36.37
g-30s-2ef (30-2)	1,073,741,824	4,287,290,382	3.99	987,912	40.94

However, for subsequent iterations, synchronization is performed exclusively for vertices that have changed their status in the previous iteration (Lines 12–13 of Algorithm 3). Specifically, each GPU identifies the vertices updated in the previous iteration, denoted as V_{pre} (Lines 23–25 of Algorithm 3), and sends updated priorities only to the peer GPUs that require them. Conversely, each GPU integrates incoming priority updates into its local cached replicas. This data-driven approach significantly reduces inter-GPU bandwidth consumption by strictly limiting communication to the relevant subset of vertices. The SynchronizePriorities routine details this synchronization process and is summarized in Algorithm 6.

5 Performance Evaluation

In this section, we evaluate the runtime performance and solution quality of cuMIS[1] against ECL-MIS [7] and MG-MIS [29], which represent the state-of-the-art in single-GPU and multi-GPU algorithms, respectively. Our evaluation covers single-GPU, single-GPU with UVM support, and multi-GPU configurations. The performance of cuMIS in distributed multi-node environments is discussed separately in Section 7. Our experiments demonstrate cuMIS's capabilities in four primary areas. First, we quantify the execution time speedups of cuMIS over ECL-MIS and MG-MIS on a wide range of graphs, extending to billion-vertex and billion-edge scales. Second, we validate the solution quality improvement achieved by cuMIS over the baselines. Third, we assess the memory traffic reduction achieved by cuMIS when UVM is enabled in single-GPU environments. Finally, we analyze the computational overheads of key operations in cuMIS.

5.1 Experimental Setup

For experiments, we utilize a cluster of 32 compute nodes, which is a partition of a large-scale supercomputer. Each compute node is equipped with dual Intel Xeon processors and 8 NVIDIA H100 GPUs, each of which has 14,592 CUDA cores and 80 GB HBM3 memory. This cluster provides exceptional data throughput, featuring 900 GB/s of intra-node NVLink bandwidth per GPU and high inter-node bandwidth supported by 400 Gb/s InfiniBand links. To evaluate cuMIS on widely deployed legacy architectures, we also employ a system equipped with 8 NVIDIA Tesla V100 GPUs, featuring 5,120 CUDA cores and 32 GB HBM2 memory each.

We benchmark cuMIS using large-scale Kronecker graphs generated by Graph500 [4, 14], which exhibit highly irregular structures representative of real-world workloads. As detailed in Table 1, our single-node performance evaluation is performed on 15 graphs ranging from scale 25 (33.5 million vertices) to scale 30 (1.07 billion vertices), with edge counts exceeding 4 billion. We further demonstrate cuMIS's extreme scalability by processing a massive 1 trillion edge graph (scale 35, 34.4 billion vertices) on 32 compute nodes in Section 7. The performance evaluation is conducted against two state-of-the-art baselines: ECL-MIS [7] and MG-MIS [29]. The former is used for single-GPU configurations with and without UVM support, while the latter is used for multi-GPU configurations. Note that the experimental results reported in this section are limited to single-node experiments, as ECL-MIS and MG-MIS do not support multi-node environments. Since the four largest graphs (i.e., g-27s-16ef, g-28s-8ef, g-29s-4ef, and g-30s-2ef) exceed the physical memory capacity of the V100 (32 GB), we run experiments for cuMIS and ECL-MIS on the V100 system with UVM support enabled for these cases. Finally, we ensure the correctness of our implementation by rigorously verifying the independence and maximality properties of the resulting MIS I after every experiment.

5.2 Runtime Performance

We evaluate the computational efficiency of cuMIS by quantifying execution time speedups relative to state-of-the-art baselines. We begin with single-GPU configurations. To assess the efficacy of our degree-aware workload distribution, we include a variant called cuMIS-warp. This variant operates identically to cuMIS but employs a uniform warp-centric strategy for all vertices, thereby bypassing the degree-aware vertex reordering and grouping, reassigning new vertex IDs, and updating the CSR representation. Note that the reported execution times for both cuMIS and cuMIS-warp include the time required for vertex degree computation and all subsequent steps, ensuring a fair comparison of end-to-end performance.

Figure 3 reports the execution time speedups of cuMIS-warp and cuMIS over ECL-MIS on single-GPU configurations. cuMIS-warp consistently outperforms ECL-MIS on the modern H100 system, achieving speedups up to 4.2×. This advantage stems partially from the warp-centric execution, which handles mid-to-high degree vertices more effectively than ECL-MIS's thread-centric design by improving memory coalescing. However, on the V100 system, we observe minor regressions on a few graphs (i.e., g-26s series, 10ef–14ef). Because cuMIS-warp applies the same warp-based strategy to all vertices, it suffers from significant thread underutilization when processing the vast majority of low-degree vertices. Nonetheless, cuMIS effectively eliminates these bottlenecks. With the degree-aware workload distribution, cuMIS achieves even higher speedups over ECL-MIS on both architectures. It reaches up to 5.8× on the H100 and 2.57× on the V100. This consistent performance gain across hardware specifications confirms the effectiveness of cuMIS in mitigating the underlying irregularities of power-law graphs.

We next present the speedups of cuMIS over MG-MIS on multi-GPU configurations—using both H100 and V100 clusters with up to 8 GPUs each. As shown in Figure 4, cuMIS consistently outperforms MG-MIS across all test graphs on both clusters. Speedups range from 15× to 65× on the H100 cluster, and from 2× to 32×

[1] https://github.com/jnke2016/cuMIS

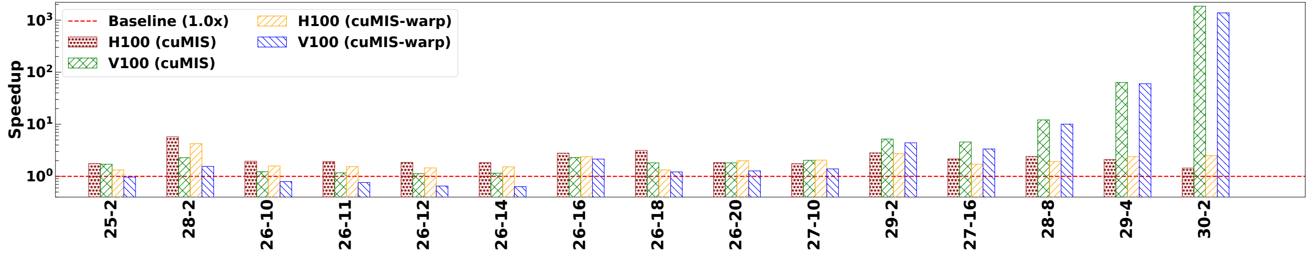
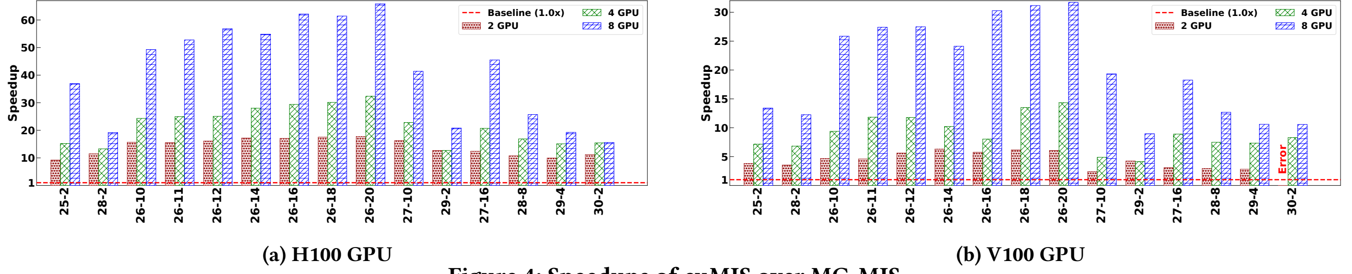


Figure 3: Speedups of cuMIS over ECL-MIS.



(a) H100 GPU

(b) V100 GPU

Figure 4: Speedups of cuMIS over MG-MIS.

Table 2: Iteration counts by GPU configuration (mean ± std)

	ECL-MIS/MG-MIS		cuMIS	
# GPU	H100	V100	H100	V100
1 GPU	1891.6 ± 1436.6	206.7 ± 202.8	8.8 ± 0.4	9.0 ± 0.5
2 GPUs	93.5 ± 31.0	94.4 ± 32.0	9.9 ± 0.6	10.1 ± 0.5
4 GPUs	104.6 ± 34.8	104.6 ± 34.8	10.9 ± 0.6	11.1 ± 0.6
8 GPUs	111.9 ± 37.3	111.9 ± 37.3	11.5 ± 0.8	11.7 ± 0.7

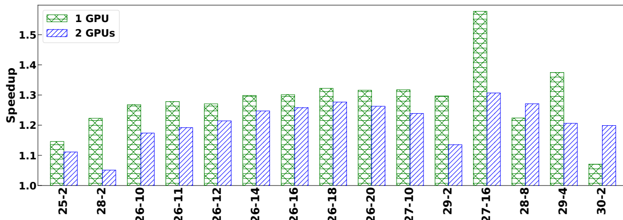


Figure 5: Speedup from the early-exit strategy.

on the V100 cluster. The performance disparity stems fundamentally from differences in graph partitioning, algorithmic execution, and communication strategies. MG-MIS relies on block-based 1D partitioning, which can lead to severe load imbalance on scale-free graphs by concentrating high-degree vertices on a single device. It also relies on fixed-size buffers for propagating vertex status updates across devices. Since these buffers frequently overflow, the algorithm needs to run extra iterations solely to clear the backlog. In contrast, cuMIS employs hash-based partitioning, ensuring a more balanced workload distribution across devices. In addition, our degree-aware workload distribution ensures efficient processing of the local subgraph on each device. Furthermore, cuMIS minimizes remote memory access and data transfer by utilizing cached priority replicas for boundary vertices and limiting synchronization to vertices that have changed status in the previous iteration. Thus, cuMIS maintains high GPU utilization and minimizes communication overhead, achieving superior performance to MG-MIS.

To support the speedups of cuMIS over ECL-MIS and MG-MIS, we report the number of iterations required by each algorithm until convergence in Table 2, where ECL-MIS and MG-MIS are used for single-GPU and multi-GPU configurations, respectively. For example, on the H100 system, ECL-MIS requires an average of 1891.6 iterations, while cuMIS converges in just 8.8 iterations on average. ECL-MIS's high iteration count stems from its asynchronous execution model. Vertices with undecided higher-priority neighbors trigger a retry loop, effectively causing the algorithm to spin through a large number of iterations until their status is decided. This inefficiency is exacerbated by the thread-centric design. When a high-degree vertex joins the MIS, its assigned single thread must sequentially update the status of all its neighbors. During this serialized update, all its neighbors repeatedly iterate until their status is finally resolved by the thread owning the high-degree vertex.

For multiple GPUs, MG-MIS runs for fewer iterations than ECL-MIS but still requires about 100 iterations on average, which is ten times more than cuMIS. In MG-MIS, the status of remote neighbors is communicated via fixed-size buffers. When these buffers fill, the algorithm triggers auxiliary iterations solely to flush the backlog, delaying convergence. In contrast, cuMIS defers status updates to the next synchronous round, ensuring that the inclusion and exclusion of a vertex are determined based on a consistent global snapshot. This allows all local maxima to join the MIS simultaneously in parallel, maximizing the number of finalized vertices per iteration without the need for retries or buffer-induced stalls.

We also assess the benefit of the early-exit strategy in cuMIS by comparing its performance against a variant where this optimization is disabled. As explained in Section 3.3, although early-exit is standard in ECL-MIS and MG-MIS, we restrict it exclusively to the first iteration in cuMIS. While an early exit triggered by a neighbor with merely higher priority could normally mask a neighbor already in the MIS, such conflicts are impossible during the first iteration. As shown in Figure 5, this optimization delivers additional speedups of up to 1.6× on a single H100 GPU and 1.3× on

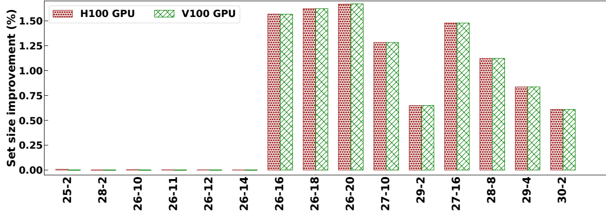


Figure 6: Set size improvement of cuMIS over ECL-MIS.

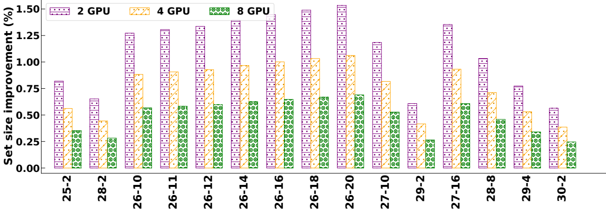


Figure 7: Set size improvement of cuMIS over MG-MIS.

two H100 GPUs. We observe similar improvements across different GPU counts and on the V100 system.

5.3 MIS Solution Quality

We next evaluate the quality of the MIS generated by cuMIS, quantified by its cardinality. Larger MIS sizes indicate superior solution quality. Figures 6 and 7 present the relative improvements in set size achieved by cuMIS over ECL-MIS and MG-MIS, respectively. Here, the relative improvement is computed as $((|I_{\text{cuMIS}}| - |I_{\text{baseline}}|)/|I_{\text{baseline}}|) \times 100$, where I_{cuMIS} and I_{baseline} denote the independent sets found by cuMIS and the baseline, respectively. For instance, on the H100 system with 2 GPUs, a 0.56% improvement on g-30s-2ef corresponds to an absolute increase of approximately 5.7 million vertices in the MIS. Note that because solution quality is determined by the algorithm and graph topology—independent of the hardware—the results are identical for both the H100 and V100 clusters. Thus, in Figure 7, we only report results on the H100 cluster for brevity.

We observe that cuMIS matches the solution quality of ECL-MIS on six small graphs and surpasses it on the remaining nine graphs. Furthermore, cuMIS consistently yields larger MIS sizes than MG-MIS across all test graphs. It is known that assigning higher priority to low-degree vertices generally yields larger independent sets [7]. While both cuMIS and ECL-MIS adopt this degree-based strategy, MG-MIS employs a simple random priority assignment to avoid the inter-GPU communication overhead required for global degree computation. Although this avoidance simplifies the implementation, it compromises solution quality. In fact, cuMIS demonstrates that this trade-off is unnecessary. Even with the communication cost of degree computation, cuMIS consistently outperforms MG-MIS in both execution speed and solution quality.

5.4 Memory Efficiency and Oversubscription

We provide an in-depth analysis of the runtime performance of cuMIS and ECL-MIS on the four largest graphs (i.e., g-27s-16ef, g-28s-8ef, g-29s-4ef, and g-30s-2ef) when a single V100 GPU is used with UVM support. For these graphs, the performance improvement of cuMIS over ECL-MIS is significant, with speedups reaching over 1000×. We observe that ECL-MIS experiences severe slowdowns due to page thrashing. For example, on g-30s-2ef, it requires over 2 hours to complete. However, cuMIS computes the MIS on the same graph in just 3.98 seconds. To better understand this performance disparity, we analyze their UVM data transfers using NVIDIA Visual Profiler [31] and report the details in Table 3.

Table 3: Data transfers under UVM execution

(a) ECL-MIS

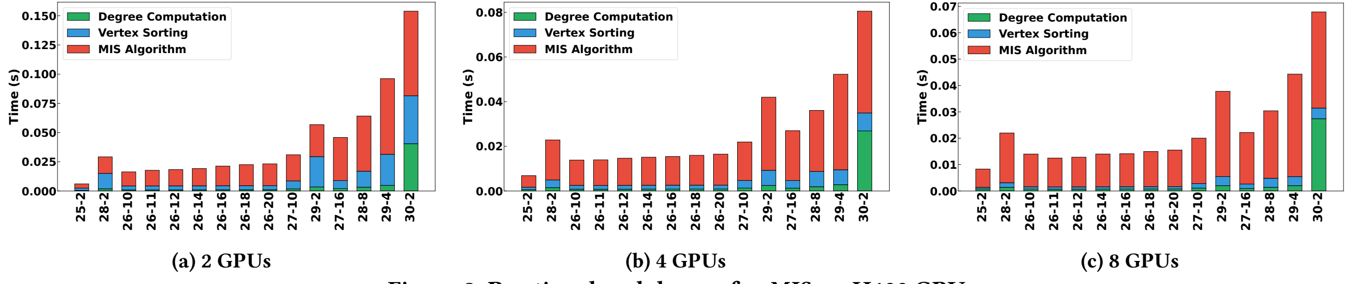
Input		Count	Total Size	Total Time
g-27s-16ef	Host To Device	110,562	4.598 GB	0.820 s
	Device To Host	3,434	6.707 GB	0.619 s
	Page faults	1,622	9 MB	3.855 s
g-28s-8ef	Host To Device	712,315	35.509 GB	6.909 s
	Device To Host	22,660	40.402 GB	4.218 s
	Page faults	12,417	78 MB	25.469 s
g-29s-4ef	Host To Device	12,989,659	593.299 GB	96.311 s
	Device To Host	415,015	595.771 GB	52.764 s
	Page faults	194,507	1.328 GB	312.737 s

(b) cuMIS (1 GPU)

Input		Count	Total Size	Total Time
g-27s-16ef	Host To Device	100,410	4.188 GB	0.724 s
	Device To Host	18	252 B	69 μs
	Page faults	4,700	0.244 GB	1.248 s
g-28s-8ef	Host To Device	98,624	3.843 GB	0.684 s
	Device To Host	18	252 B	62 μs
	Page faults	7,261	0.339 GB	1.621 s
g-29s-4ef	Host To Device	129,674	4.371 GB	0.836 s
	Device To Host	20	280 B	86 μs
	Page faults	11,094	0.490 GB	2.195 s

For example, on g-28s-8ef, ECL-MIS triggers 712,315 host-to-device transfers totaling 35.5 GB, while cuMIS requires only 98,624 transfers amounting to 3.843 GB—a 7× reduction in transfer count and a 9× reduction in transfer data volume. The performance disparity arises because ECL-MIS processes all vertices in every iteration, and when a vertex is included in the MIS, it must update the status of all its neighbors. These algorithmic characteristics lead to scattered dirty pages across memory, forcing costly write-back evictions when device memory is saturated. This creates a destructive thrashing cycle, where pages are repeatedly evicted and re-fetched, inducing severe memory contention. For g-28s-8ef, the near-equal host-to-device (35.5 GB) and device-to-host (40.4 GB) transfers confirm that the same data is continuously cycling between host and device memory. On g-27s-16ef, the memory contention causes ECL-MIS to experience 3× longer page fault time than cuMIS despite having fewer faults. However, cuMIS minimizes these bottlenecks by limiting processing to the active set of undecided vertices in each iteration and employing a read-only neighbor inspection strategy. In addition, the degree-based vertex grouping increases spatial locality, ensuring that neighbor access patterns are contiguous. These optimizations align well with UVM to retain frequently accessed pages on the device, minimizing both data transfer and contention with virtually no device-to-host evictions for all datasets.

It is also worth noting that cuMIS on a single V100 GPU often rivals or even exceeds the performance of MG-MIS running on multiple V100 GPUs. For instance, on g-27s-16ef, single-GPU cuMIS (0.89 seconds) outperforms eight-GPU MG-MIS (1.21 seconds). This



(a) 2 GPUs (b) 4 GPUs (c) 8 GPUs

Figure 8: Runtime breakdown of cuMIS on H100 GPU.

Table 4: Information about the real-world input graphs

Input	$\|V\|$	$\|E\|$	d_{avg}	d_{max}	Size (GB)
europe-osm	50,912,018	108,109,320	2.12	13	0.59
soc-twitter-2010	21,297,772	530,051,090	24.89	698,112	2.05
it-2004	41,291,318	2,054,949,894	49.77	1,326,745	8.77
uk-2007-05	105,896,435	6,603,753,128	62.36	975,418	25.39

efficiency extends to larger graphs. On g-29s-4ef, single-GPU cuMIS (2.31 seconds) matches the performance of two-GPU MG-MIS (2.34 seconds). Even for the largest graph, g-30s-2ef, where memory pressure is maximal, single-GPU cuMIS (3.98 seconds) remains highly competitive with four-GPU MG-MIS (3.09 seconds), while MG-MIS fails with an out-of-memory error on two GPUs. These results highlight the resource efficiency of cuMIS, demonstrating its ability to process massive graphs on a single device that would traditionally require a multi-GPU cluster.

5.5 Computational Overheads

We finally analyze the computational overheads of cuMIS. Figure 8 presents the runtime breakdown of cuMIS on H100 GPUs, with all preprocessing costs included. Global degree computation constitutes 7–23% of total runtime, averaging about 12% across all multi-GPU configurations. We observe that inter-GPU communication accounts for 89–90% of the time required for the global degree computation. As the GPU count increases from two to eight, the absolute cost of degree computation grows by 1.7×, but its share of the total runtime remains stable at 12%, indicating efficient scaling.

In addition, vertex sorting contributes another 5–14% of the total runtime. This phase includes parallel radix sort, degree-based vertex grouping, and vertex ID reassignment. It is important to note that this phase is performed entirely on local data with no inter-GPU communication. The combined preprocessing overhead of degree computation and vertex sorting amounts to approximately 12–37% of the total runtime. Despite this overhead, the resulting degree-based priority assignment and degree-aware workload distribution lead to significant improvements in both execution speed and solution quality. That is, the benefits outweigh the costs.

6 Performance Evaluation with Real Graphs

To complement the performance evaluation in Section 5, which focuses on Graph500 synthetic datasets to stress-test implementations at scale, we next benchmark cuMIS against ECL-MIS and MG-MIS on four large-scale, real-world graphs. As summarized in Table 4, these datasets span distinct structural classes. The soc-twitter-2010 graph is a social network exhibiting the power-law

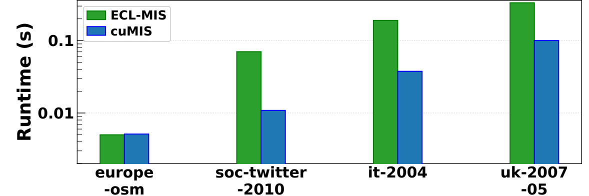


Figure 9: Single-GPU runtimes of cuMIS and ECL-MIS.

Table 5: Iteration counts and MIS coverage (%) of cuMIS and ECL-MIS

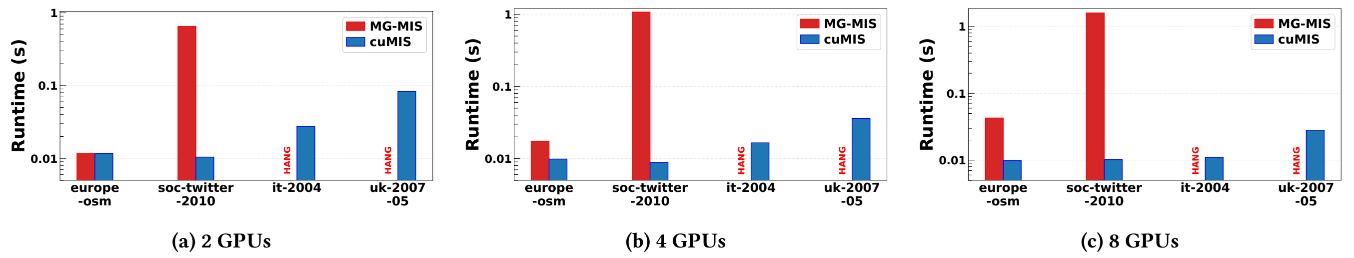
Input	ECL-MIS		cuMIS	
	Iters	Cov. (%)	Iters	Cov. (%)
europe-osm	22	45.00	2441	49.09
soc-twitter-2010	1979	63.50	11	63.48
it-2004	3104	60.30	194	60.37
uk-2007-05	1658	59.10	221	61.38

degree distribution typical of such graphs. The europe-osm graph is a road network characterized by an extremely low average degree ($d_{\text{avg}} = 2.12$) and a near-uniform degree distribution ($d_{\text{max}} = 13$). The uk-2007-05 and it-2004 graphs are web crawls featuring highly skewed degree distributions with maximum degrees exceeding 1.3 million. Since uk-2007-05 and it-2004 are originally directed graphs, they were converted to undirected graphs with all self-loops and multi-edges removed. These four topologies provide a comprehensive evaluation across fundamentally different graph structures. All experimental results reported in this section are obtained using the H100 system, as its physical memory capacity is sufficient to accommodate these four datasets.

6.1 Single-GPU Performance

We first present experimental results for single-GPU configurations. Figure 9 shows the single-GPU runtimes of cuMIS and ECL-MIS, both of which successfully process all four datasets. cuMIS achieves a 6.5× speedup over ECL-MIS on soc-twitter-2010, a 5× speedup on it-2004, and a 3.32× speedup on uk-2007-05. The europe-osm graph, with its low average degree of 2.12 edges per vertex, yields comparable runtimes for both algorithms. Overall, the results confirm the superior performance of cuMIS over ECL-MIS.

In Table 5, we report the iteration counts and MIS solution qualities of cuMIS and ECL-MIS, where the MIS quality is measured as the coverage (%) of the MIS relative to the original vertex count. cuMIS converges in significantly fewer iterations than ECL-MIS on the soc-twitter-2010, it-2004, and uk-2007-05 graphs. Specifically,



(a) 2 GPUs **(b) 4 GPUs** **(c) 8 GPUs**

Figure 10: Multi-GPU runtimes of cuMIS and MG-MIS on real-world graphs.

cuMIS requires just 11 iterations on soc-twitter-2010 (vs. 1,979 for ECL-MIS), 194 on it-2004 (vs. 3,104), and 221 on uk-2007-05 (vs. 1,658).

An exception to this trend is the europe-osm graph, where cuMIS requires 2,441 iterations, the highest across all datasets, while ECL-MIS converges in just 22 iterations. This is a direct consequence of the road-network structure, where the vast majority of vertices have a degree of 2, forming long paths with few branching points. When nearly all vertices share the same degree, cuMIS's priority assignment reduces to tie-breaking by vertex ID, creating long monotonic priority sequences along these paths. In contrast, the random perturbation in ECL-MIS's priority computation breaks these chains. It is worth noting that cuMIS's deterministic priority assignment may sacrifice the $O(\log |V|)$ convergence, which is the desirable convergence property of Luby's randomized approach [27]. The europe-osm graph represents a pathological monotonic-path scenario where cuMIS requires more iterations to converge. Nonetheless, its overall runtime remains comparable to that of ECL-MIS.

Furthermore, the solution quality of cuMIS is consistently on par with or superior to that of ECL-MIS. Specifically, both algorithms achieve an MIS coverage of approximately 63% for soc-twitter-2010 and 60% for it-2004. However, cuMIS achieves a higher MIS coverage than ECL-MIS on the europe-osm (49.09% vs. 45.00%) and uk-2007-05 (61.38% vs. 59.10%) graphs.

6.2 Multi-GPU Performance

We present the experimental results for multi-GPU configurations. Figure 10 shows the runtimes of cuMIS and MG-MIS on 2, 4, and 8 GPUs. cuMIS successfully processes all four datasets across all GPU configurations, while MG-MIS completes only on soc-twitter-2010 and europe-osm, failing to terminate within two hours on it-2004 and uk-2007-05. On soc-twitter-2010, cuMIS achieves speedups ranging from 62× to 156× over MG-MIS, with the performance gap widening at higher GPU counts. These results are consistent with the trends observed on the Graph500 synthetic datasets in Section 5. While a similar improvement is observed on europe-osm, the performance gap is more modest, ranging from 0.99× to 4.35×. This may be due to MG-MIS's random priority assignment, which breaks the long monotonic tie-breaking chains of such a low-degree road graph.

In Table 6, we present the iteration counts and MIS coverage of cuMIS and MG-MIS for multi-GPU configurations. cuMIS's consistently low iteration counts confirm its viability across GPU configurations, even for the europe-osm graph. On europe-osm, while the iteration count for cuMIS was 2,441 on a single GPU (as reported in Table 5), it drops to 25 on 2 GPUs and further to 12 on 8 GPUs.

Table 6: Iteration counts and MIS coverage (%) of cuMIS and MG-MIS

Input	# GPUs	MG-MIS		cuMIS	
		Iters	Cov. (%)	Iters	Cov. (%)
europe-osm	2	4	43.50	25	45.17
	4	5	43.50	15	44.18
	8	5	43.50	12	43.82
soc-twitter-2010	2	312	58.34	12	61.97
	4	322	58.34	12	60.67
	8	327	58.34	15	59.76
it-2004	2	–	–	35	59.73
	4	–	–	13	58.94
	8	–	–	13	58.02
uk-2007-05	2	–	–	31	60.94
	4	–	–	22	60.27
	8	–	–	20	59.44

Similar reductions are observed on it-2004 (e.g., 194 on a single GPU vs. 13 on 8 GPUs) and uk-2007-05 (e.g., 221 on a single GPU vs. 20 on 8 GPUs).

This iteration reduction is attributed to the hash-based partitioning strategy adopted in the multi-GPU implementation of cuMIS, as explained in Section 4. Vertices are assigned to GPUs by hashing their original IDs, which is independent of their degree. While priorities remain degree-dependent within a partition, the cross-partition processing order is determined by GPU IDs, effectively introducing pseudo-randomness into the execution schedule. As the GPU count increases, the execution schedule becomes increasingly randomized. This is particularly effective for the europe-osm graph. By distributing vertices across multiple GPUs, the hash-based partitioning successfully breaks the long monotonic tie-breaking chains that bottlenecked the single-GPU execution. It is worth noting that for soc-twitter-2010, the iteration count of cuMIS remains stable (12 to 15 iterations) across GPU counts because the power-law degree distribution already provides sufficient differentiation in vertex priorities, making the hash-induced randomization neither beneficial nor detrimental. In contrast, MG-MIS requires 312 to 327 iterations on the same dataset.

Finally, despite the hash-induced randomization of the execution schedule on multiple GPUs, cuMIS consistently produces larger independent sets than MG-MIS on the datasets where both complete. On soc-twitter-2010, cuMIS achieves 59.76% to 61.97% coverage compared to 58.34% for MG-MIS. On europe-osm, cuMIS coverage ranges from 43.82% to 45.17% compared to 43.50% for MG-MIS. Overall, the MIS coverage of cuMIS also remains stable as the GPU count increases. For instance, coverage shifts from 49.09% on a

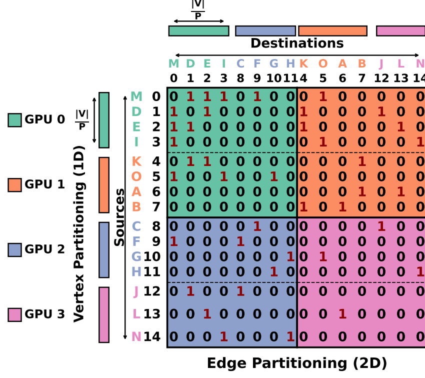


Figure 11: 2D graph partitioning.

single GPU to 43.82% on 8 GPUs for europe-osm, from 63.48% to 59.76% for soc-twitter-2010, from 60.37% to 58.02% for it-2004, and from 61.38% to 59.44% for uk-2007-05. These results demonstrate the effectiveness of cuMIS in maintaining high solution quality across diverse graph structures.

7 Scaling to Trillion-Edge Graphs

In this section, we discuss our 2D partitioning strategy to scale cuMIS to trillion-edge graphs in distributed multi-node environments and evaluate its performance compared to the hash-based 1D partitioning approach on a cluster of 32 compute nodes.

7.1 2D Partitioning for Extreme Scalability

While our hash-based 1D partitioning mitigates load imbalance on a single node with multiple GPUs by uniformly distributing edges, it limits scalability in multi-node settings due to its all-to-all communication pattern. As the GPU count P increases, the communication fan-out per device grows linearly, eventually saturating network bandwidth. To overcome this bottleneck, we adopt a 2D partitioning strategy [6, 22, 23, 36, 38] that decomposes the adjacency matrix into a $P_{row} \times P_{col}$ grid where $P = P_{row} \times P_{col}$ with $P_{row} \approx P_{col} \approx \sqrt{P}$. An edge from u to v is mapped to the GPU at coordinate $(row(u), col(v))$, where $row(u)$ is the row index of the GPU owning source u, and $col(v)$ is the column index of the GPU owning destination v. This mapping ensures that all outgoing edges of a vertex are distributed among GPUs within a single row. Similarly, an edge from v to u is mapped to the GPU at coordinate $(row(v), col(u))$. Note that since the input graph is undirected, both directed representations of each edge are stored and may reside on different GPUs. Thus, the 2D partitioning strategy limits inter-GPU communication to peers within the same row or column, reducing the number of synchronization partners by a factor of $\sqrt{P}$.

Figure 11 illustrates the 2D partitioning of the example graph from Figure 2 onto a 2×2 GPU grid. The grid positions are: GPU 0 at (0, 0), GPU 1 at (0, 1), GPU 2 at (1, 0), and GPU 3 at (1, 1). Recall from Section 4 that in the single-GPU case, vertices are sorted by descending degree and then by increasing original IDs. In the multi-GPU setting, this same sorting is performed independently within each GPU's local partition. This local ordering is then used to assign global IDs in consecutive ranges across the GPUs, with ties broken lexicographically in this example. Thus, vertex `M` (degree 5) receives the smallest ID 0, while vertex `N`, the last lexicographically among degree-2 vertices, receives the highest ID 14. For simplicity, we refer to vertices by their global IDs. The colored bar on the left indicates vertex ownership: GPU 0 owns vertices 0–3; GPU 1 owns 4–7; and GPUs 2 and 3 own 8–11 and 12–14, respectively. As described above, all outgoing edges of a vertex are distributed among GPUs within a single row. For example, all outgoing edges from vertex 0 are distributed to GPUs 0 and 1. The edge from vertex 0 to 5 is assigned to GPU 1 at (0, 1), and the edge from vertex 0 to 9 maps to GPU 0 at (0, 0). Similarly, the edge from 5 to 0 is assigned to GPU 0 at (0, 0), and the edge from vertex 9 to 0 maps to GPU 2 at (1, 0).

Similar to our implementation with the hash-based 1D partitioning, our 2D implementation maintains cached replicas to minimize data transfer during neighbor traversal. However, instead of caching priorities of boundary vertices from all P GPUs as in the 1D case, each GPU in 2D only exchanges priorities with its row or column peers. GPUs in the same column coordinate to maintain consistent priorities for the vertices mapped to that column. Consider the example in Figure 11, where GPU 0 at (0, 0) and GPU 2 at (1, 0) share column 0. Although GPU 0 owns vertices 0–3 and GPU 2 owns vertices 8–11, both GPUs maintain cached replicas of the priorities for all eight vertices. When GPU 2 updates vertex 8's priority, it broadcasts the change only to GPU 0 (its column peer), ensuring that both maintain a consistent view. For row peers, the communication pattern differs. Since vertex 0's outgoing edges are distributed between GPU 0 and GPU 1 (row 0), its neighbor list is split. During neighbor traversal, each GPU computes a local maximum neighbor priority for vertex 0. A row-wise reduction then follows to produce the global maximum needed to decide vertex 0's status. This structured communication reduces both memory overhead and synchronization cost from $O(P)$ to $O(\sqrt{P})$, enabling scalability to hundreds of GPUs.

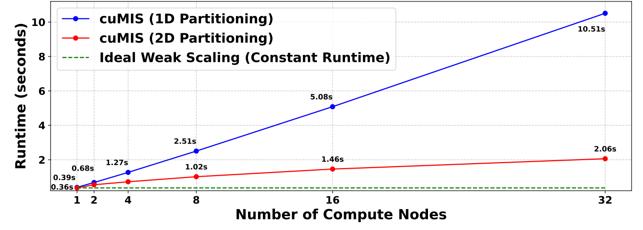


Figure 12: Weak scaling (Graph500 Scale 30–35).

7.2 Weak Scaling Analysis

We validate our 2D partitioning strategy through weak scaling experiments on a cluster of 32 compute nodes, with 8 H100 GPUs each. Weak scaling assesses the system's ability to handle larger problem sizes by increasing the workload proportionally with computational resources [15]. We evaluate the performance of cuMIS with 1D and 2D partitioning strategies using a sequence of Graph500 graphs, scaling from Scale 30 on a single compute node up to Scale 35 (34.4 billion vertices and 1.1 trillion edges) on 32 nodes.

As shown in Figure 12, the results reveal a substantial divergence in scalability between the 1D and 2D strategies. For smaller configurations up to 2 nodes, they deliver comparable performance, with the 1D approach benefiting from its lower overhead at smaller graph sizes. However, as the scale increases beyond 8 nodes, the

limitations of 1D partitioning become pronounced. The runtime for the 1D implementation degrades rapidly, increasing by nearly 27× across the scaling range. This decline in efficiency is a direct consequence of the all-to-all communication pattern. Each GPU must synchronize with up to $P - 1$ peers, which results in $O(P^2)$ total messages across the cluster, saturating the interconnect bandwidth.

In contrast, our 2D implementation demonstrates superior scalability. At the maximum scale of 32 nodes, it successfully computes the MIS for the trillion-edge graph in just 2.06 seconds—a 5.1× speedup over the 1D baseline. While the runtime does increase from 0.36 seconds to 2.06 seconds, this growth is much more moderate compared to the 1D baseline. This improvement is attributed to the $\sqrt{P}$ scaling property of our row and column communicators. For our largest configuration with $P = 256$ GPUs, the communication fan-out per device reduces from 255 potential peers to 15 row or column peers (i.e., $\sqrt{256} - 1$). This drastically lowers synchronization costs, establishing cuMIS with the 2D partitioning strategy as an effective solution for computing MIS on trillion-edge graphs.

8 Conclusion

We presented cuMIS, a high-performance framework for computing MIS on modern GPU architectures. Our unified design seamlessly spans single-GPU (with and without UVM), multi-GPU, and distributed multi-node configurations, addressing the scalability limitations of existing approaches. It is the first MIS implementation capable of processing trillion-edge graphs in a distributed multi-node environment with 256 GPUs. This extreme scalability stems from a data-driven execution model with read-only neighbor traversal, a hybrid-granularity workload distribution, and efficient data partitioning and inter-GPU communication.

Acknowledgments

This work was supported by the National Science Foundation under Grant Nos. 2209921 and 2209922, and an equipment donation from NVIDIA Corporation.

References

[1] Tyler Allen, Bennett Cooper, and Rong Ge. 2024. Fine-grain Quantitative Analysis of Demand Paging in Unified Virtual Memory. *ACM Trans. Archit. Code Optim.* 21, 1, Article 14 (Jan. 2024), 24 pages.

[2] Daniel Anderson, Guy E. Blelloch, Laxman Dhulipala, Magdalen Dobson, and Yihan Sun. 2022. The problem-based benchmark suite (PBBS), V2. In *Proceedings of the 27th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming* (Seoul, Republic of Korea) *(PPoPP '22)*. Association for Computing Machinery, New York, NY, USA, 445–447.

[3] Eugene T.Y. Ang, Prasanta Bhattacharya, and Andrew E.B. Lim. 2025. Estimating policy effects in a social network with independent set sampling. *Social Networks* 81 (2025), 17–30.

[4] James Alfred Ang, Brian W Barrett, Kyle Bruce Wheeler, and Richard C Murphy. 2010. Introducing the graph 500. In *Cray User Group (CUG)*. Sandia National Laboratories. https://www.osti.gov/biblio/1014640

[5] Scott Beamer, Krste Asanović, and David Patterson. 2012. Direction-optimizing breadth-first search. In *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis* (Salt Lake City, Utah) *(SC '12)*. IEEE Computer Society Press, Washington, DC, USA, Article 12, 10 pages.

[6] Erik G. Boman, Karen D. Devine, and Sivasankaran Rajamanickam. 2013. Scalable matrix computations on large scale-free graphs using 2D graph partitioning. In *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis* (Denver, Colorado) *(SC '13)*. Association for Computing Machinery, New York, NY, USA, Article 50, 12 pages.

[7] Martin Burtscher, Sindhu Devale, Sahar Azimi, Jayadharini Jaiganesh, and Evan Powers. 2018. A High-Quality and Fast Maximal Independent Set Implementation for GPUs. *ACM Trans. Parallel Comput.* 5, 2, Article 8 (Dec. 2018), 27 pages.

[8] Shuai Che, Bradford M. Beckmann, Steven K. Reinhardt, and Kevin Skadron. 2013. Pannotia: Understanding irregular GPGPU graph applications. In *2013 IEEE International Symposium on Workload Characterization (IISWC)*. 185–195.

[9] Steven Dalton, Nathan Bell, Luke Olson, and Michael Garland. 2014. CUSP: Generic parallel algorithms for sparse matrix and graph computations. http://cusplibrary.github.io/ Version 0.5.0.

[10] Andreas Eisenblätter. 2002. *Frequency assignment in GSM networks: Models, heuristics and lower bounds*. Cuvillier Verlag.

[11] David Eppstein. 2001. Small Maximal Independent Sets and Faster Exact Graph Coloring. In *Proceedings of the 7th International Workshop on Algorithms and Data Structures (WADS '01)*. Springer-Verlag, Berlin, Heidelberg, 462–470.

[12] Ilya Gertsbakh and Helman I Stern. 1978. Minimal resources for fixed and variable job schedules. *Operations Research* 26, 1 (1978), 68–85.

[13] Joseph E. Gonzalez, Yucheng Low, Haijie Gu, Danny Bickson, and Carlos Guestrin. 2012. PowerGraph: Distributed graph-parallel computation on natural graphs. In *Proceedings of the 10th USENIX Conference on Operating Systems Design and Implementation* (Hollywood, CA, USA) *(OSDI'12)*. USENIX Association, USA, 17–30.

[14] Graph500. 2017. Graph500 Benchmark Version 3.0.0. https://github.com/graph500/graph500. Accessed: 2025-02-23.

[15] John L. Gustafson. 1988. Reevaluating Amdahl's law. *Commun. ACM* 31, 5 (May 1988), 532–533.

[16] D. Gómez, J. Montero, J. Yáñez, and C. Poidomani. 2007. A graph coloring approach for image segmentation. *Omega* 35, 2 (2007), 173–183.

[17] Mark Harris. 2017. Unified memory for CUDA beginners. https://developer.nvidia.com/blog/unified-memory-cuda-beginners/. Accessed: 2025-12-13.

[18] Mark Harris. 2020. Fast, Flexible Allocation for NVIDIA CUDA with RAPIDS Memory Manager. https://developer.nvidia.com/blog/fast-flexible-allocation-for-cuda-with-rapids-memory-manager/. Accessed: 2025-12-13.

[19] Sungpack Hong, Sang Kyun Kim, Tayo Oguntebi, and Kunle Olukotun. 2011. Accelerating CUDA graph algorithms at maximum warp. In *Proceedings of the 16th ACM Symposium on Principles and Practice of Parallel Programming* (San Antonio, TX, USA) *(PPoPP '11)*. Association for Computing Machinery, New York, NY, USA, 267–276.

[20] Tomohiro Imanaga, Koji Nakano, Masaki Tao, Ryota Yasudo, Yasuaki Ito, Yuya Kawamata, Ryota Katsuki, Yusuke Tabata, Takashi Yazane, and Kenichiro Hamano. 2020. Efficient GPU Implementation for Solving the Maximum Independent Set Problem. In *2020 Eighth International Symposium on Computing and Networking (CANDAR)*. 29–38.

[21] Tomohiro Imanaga, Koji Nakano, Ryota Yasudo, Yasuaki Ito, Yuya Kawamata, Ryota Katsuki, Yusuke Tabata, Takashi Yazane, and Kenichiro Hamano. 2023. Simple iterative trial search for the maximum independent set problem optimized for the GPUs. *Concurrency and Computation: Practice and Experience* 35, 14 (2023).

[22] Seunghwa Kang, Alex Fender, Joe Eaton, and Brad Rees. 2020. Computing PageRank Scores of Web Crawl Data Using DGX A100 Clusters. In *2020 IEEE High Performance Extreme Computing Conference (HPEC)*. 1–4.

[23] Seunghwa Kang, Joseph Nke, and Brad Rees. 2022. Analyzing Multi-trillion Edge Graphs on Large GPU Clusters: A Case Study with PageRank. In *2022 IEEE High Performance Extreme Computing Conference (HPEC)*. 1–7.

[24] Richard M. Karp and Avi Wigderson. 1985. A fast parallel algorithm for the maximal independent set problem. *J. ACM* 32, 4 (Oct. 1985), 762–773.

[25] Brian Kelley and Sivasankaran Rajamanickam. 2022. Parallel, Portable Algorithms for Distance-2 Maximal Independent Set and Graph Coarsening. In *2022 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*. 280–290.

[26] Imran Khan and Ansaari. 2016. A New and Fast Approximation Algorithm for Vertex Cover Using Independent Set (VCUMI). *Operations Research and Decisions* 25, 1 (2016), 201–215.

[27] M Luby. 1985. A simple parallel algorithm for the maximal independent set problem. In *Proceedings of the Seventeenth Annual ACM Symposium on Theory of Computing* (Providence, Rhode Island, USA) *(STOC '85)*. Association for Computing Machinery, New York, NY, USA, 1–10.

[28] Leran Ma, Ke Chen, and Mingfu Shao. 2023. On the Maximal Independent Sets of k-mers with the Edit Distance. In *Proceedings of the 14th ACM International Conference on Bioinformatics, Computational Biology, and Health Informatics* (Houston, TX, USA) *(BCB '23)*. Association for Computing Machinery, New York, NY, USA, Article 42, 6 pages.

[29] Anju Mongandampulath Akathoott, Benila Virgin Jerald Xavier, and Martin Burtscher. 2025. A Multi-GPU Algorithm for Computing Maximal Independent Sets in Large Graphs. In *Proceedings of the 39th ACM International Conference on Supercomputing (ICS '25)*. Association for Computing Machinery, New York, NY, USA, 1162–1175.

[30] Maxim Naumov. 2015. Graph Coloring: More Parallelism for Incomplete-LU Factorization. https://developer.nvidia.com/blog/graph-coloring-more-parallelism-for-incomplete-lu-factorization/. *NVIDIA Developer Blog* (June 2015). Accessed: 2025-10-27.

[31] NVIDIA Corporation. 2023. NVIDIA Visual Profiler. https://docs.nvidia.com/cuda/profiler-users-guide/. Accessed: 2025-10-27.

[32] Sreepathi Pai and Keshav Pingali. 2016. A compiler for throughput optimization of graph algorithms on GPUs. In *Proceedings of the 2016 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications* (Amsterdam, Netherlands) *(OOPSLA 2016)*. Association for Computing Machinery, New York, NY, USA, 1–19.

[33] Keshav Pingali, Donald Nguyen, Milind Kulkarni, Martin Burtscher, M. Amber Hassaan, Rashid Kaleem, Tsung-Hsien Lee, Andrew Lenharth, Roman Manevich, Mario Méndez-Lojo, Dimitrios Prountzos, and Xin Sui. 2011. The tao of parallelism in algorithms. In *Proceedings of the 32nd ACM SIGPLAN Conference on Programming Language Design and Implementation* (San Jose, California, USA) *(PLDI '11)*. Association for Computing Machinery, New York, NY, USA, 12–25.

[34] Richard L. Rudell and Alberto Sangiovanni-Vincentelli. 1989. *Logic synthesis for VLSI design*. University of California, Berkeley. AAI9006491.

[35] Julian Shun and Guy E. Blelloch. 2013. Ligra: A lightweight graph processing framework for shared memory. In *Proceedings of the 18th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming* (Shenzhen, China) *(PPoPP '13)*. Association for Computing Machinery, New York, NY, USA, 135–146.

[36] Koji Ueno and Toyotaro Suzumura. 2012. 2D Partitioning Based Graph Search for the Graph500 Benchmark. In *2012 IEEE 26th International Parallel and Distributed Processing Symposium Workshops & PhD Forum*. 1925–1931.

[37] Xubo Wang, Dong Wen, Wenjie Zhang, Ying Zhang, and Lu Qin. 2023. Distributed Near-Maximum Independent Set Maintenance over Large-scale Dynamic Graphs. In *2023 IEEE 39th International Conference on Data Engineering (ICDE)*. 2538–2550.

[38] Andy Yoo, Edmond Chow, Keith Henderson, William McLendon, Bruce Hendrickson, and Umit Catalyurek. 2005. A Scalable Distributed Parallel Breadth-First Search Algorithm on BlueGene/L. In *Proceedings of the 2005 ACM/IEEE Conference on Supercomputing (SC '05)*. IEEE Computer Society, USA, 25.