ParAGraph: Parallel Agent-Based Graph Modeling

1st Jayanta Mukherjee
Department of Computer Science
Purdue University
West Lafayette, IN, USA
jmukher@purdue.edu

2nd Minhyuk Park
Siebel School of Computing
University of Illinois Urbana-Champaign
Urbana IL, USA
minhyuk2@illinois.edu

3rd George Chacko
Siebel School of Computing
University of Illinois Urbana-Champaign
Urbana IL, USA
chackoge@illinois.edu

4th Tandy Warnow
Siebel School of Computing
University of Illinois Urbana-Champaign
Urbana IL, USA
warnow@illinois.edu

5th Ananth Grama
Department of Computer Science
Purdue University
West Lafayette, IN, USA
ayg@cs.purdue.edu

***Abstract*—Simulating the growth of synthetic networks at the scale of tens of millions of nodes and hundreds of millions of edges presents a challenge for existing GPU-resident agent-based modeling (ABM) frameworks due to their prohibitive memory overhead, warp divergence, and host-device synchronization costs that constrain throughput as graphs scale. We present ParAGraph, a GPU-accelerated ABM framework for graph dynamics under Preferential Attachment and Erdős-Rényi growth models. We address algorithmic and implementation challenges through four co-designed technical contributions: (i) a compact sequential-ID vertex layout enabling direct $\mathcal{O}(1)$ indexing and coalesced memory access; (ii) staged micro-kernel scheduling with stage-local data residency; (iii) a warp-cooperative sparse-structure traversal algorithm that substantially reduces warp divergence; and (iv) *GPU-Std*[1], an open-source GPU-native data structure library providing `std`-style abstractions for GPU-resident graph structures. Through these contributions, ParAGraph achieves up to 24.3× speedup over a 16-thread CPU baseline under Preferential Attachment and 7.5× under the Erdős-Rényi model, across two architecturally different GPUs, a datacenter H100 and a consumer RTX 4090, to establish that the gains stem from algorithm-architecture co-design rather than GPU-specific hardware characteristics. Detailed profiling and ablation studies quantify the performance contribution of each optimization.**

***Index Terms*—GPU, Agent-based modeling, Preferential Attachment, Erdős-Rényi**

I. Introduction

Agent-based models (ABMs) are computational frameworks in which autonomous, heterogeneous *agents* follow local rules, with global behavior emerging from their network interactions [1], [2]. We implement two network growth models: (i) *Preferential Attachment (PA)*, in which connections depend on intrinsic fitness, accumulated in-degree, and node-arrival time, capturing preferential attachment, recency bias, and quality effects [3], [4]; and (ii) *Erdős-Rényi (ER)*, in which each directed edge (u, v) is formed independently with probability p, yielding expected out-degree $\mathbb{E}[\deg^+(u)] = np$ for a network with n nodes. While PA produces hub-dominated, irregular graphs, ER provides a uniform, unbiased baseline. Together, they represent complementary growth regimes that stress GPU computation, memory, and access patterns differently (Figure 1). PA exhibits extreme degree skew, causing load imbalance, warp divergence, and repeated access to a few adjacency rows. ER maintains more uniform degrees but grows the vertex set much faster, shifting pressure toward memory footprint and allocation overhead. Thus, PA and ER provide complementary stress regimes for **ParAGraph** across irregular and regular graph-growth workloads.

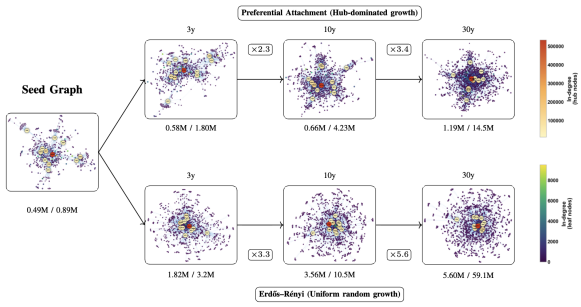


Fig. 1: **Preferential Attachment** (top) produces hub-dominated, heavy-tailed structures, while **Erdős-Rényi** (bottom) yields uniform edge densification. Values indicate *node / edge* counts.

ParAGraph aims at the generative graph workloads where the graph is constructed incrementally based on agent decisions. Graph frameworks such as Gunrock [5] and cuGraph [6] operate on a fixed topology; ABM growth instead mutates structure every epoch through node and edge insertion, disrupting locality, coalesced access, and warp balance. These challenges are tightly coupled: memory overhead limits batch size, reducing the parallelism needed to amortize irregular computation. To address these, we make the following main contributions:

This material is based upon work supported by the U.S. National Science Foundation under Grant Nos. 2513947 and 2402559. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect the views of the National Science Foundation.

[1]Source code available at https://github.com/jmukher1/ParAGraph.

1) A compact sequential-ID layout enabling lossless bitmap compression and coalesced access, reducing per-thread state by up to 32× while preserving GPU residency.
2) A staged microkernel model with stage-local memory management for immediate reclamation and overlapping CPU-GPU communication with computation.
3) GPU algorithms for incrementally constructed sparse graphs that combine warp-synchronous traversal, structure-of-arrays layouts, and register-resident sampling to reduce divergence and improve memory efficiency.
4) *GPU-Std*, an open-source CUDA library providing `std`-like GPU-resident containers (e.g., vectors, queues, sets, maps, and min-heaps) for high-performance irregular graph processing.

Together, these contributions render irregular, memory-bound ABM simulation tractable on GPUs, yielding up to **24.3**× speedup over a 16-thread CPU baseline for Preferential Attachment and up to **7.5**× for Erdős-Rényi models on both datacenter-class H100 and consumer RTX 4090 GPUs. On identical hardware, it also outperforms MASS CUDA by up to **6.5**×. To the best of our knowledge, **ParAGraph** is the first GPU framework to combine bitmap-based BFS compaction, stage-local memory management, coalesced incrementally constructed graph traversal, and reusable CUDA graph abstractions for scalable ABM execution.

II. Related Work

GPU-accelerated ABMs. Agent-based models capture decentralized interactions but are difficult to scale because of irregular communication, dynamic graph updates, and high memory demands [1], [2]. Existing GPU frameworks, including FLAME GPU [7] and MASS CUDA [8], improve programmability but retain persistent agent state and global synchronization, limiting scalability for graph-centric ABMs [9].

GPU graph processing. Frameworks such as Gunrock [5], Hornet [10], Enterprise [11], and cuGraph [6] achieve high performance on static graph workloads but do not support dynamic graph growth, per-agent BFS [12], and stage-local memory management required by ABMs. Bitmap-encoded frontiers have been used to reduce BFS overhead on both GPUs [13] and CPUs, the latter via Ligra's [14] sparse/dense frontier representation. **ParAGraph** extends this concept to GPU-resident visited sets and neighborhood buffers for *dynamically growing* graphs, achieving up to 32× state reduction while supporting PA growth, weighted-reservoir sampling, and incremental degree updates.

Table I highlights **ParAGraph** as the first framework to unify graph growth, bitmap-based per-agent BFS, staged memory management, and weighted-reservoir sampling. Its stage-local reclamation of temporary BFS state enables scalable, fully GPU-resident execution throughout graph evolution.

TABLE I: Comparison of GPU graph and ABM frameworks. ✓ = supported; × = not supported; ∼ = partial support.

System	Dynamic growth	Bitmap BFS	Per-agent BFS	Staged memory	WRS sampling
Gunrock [5]	×	×	×	×	×
Hornet [10]	∼	×	×	×	×
Enterprise [11]	×	×	×	×	×
Ligra [14]	×	✓	×	×	×
FLAME GPU [7]	∼	×	×	×	×
MASS CUDA [8]	∼	×	×	×	×
ParAGraph	✓	✓	✓	✓	✓

III. Graph Growth Models: Foundations and Formalization

Preferential Attachment and Erdős-Rényi models are commonly used to study the evolution of social networks, web graphs, and dynamic recommendation systems. While we focus on these network growth models, our algorithmic and implementation innovations apply broadly to other network growth models as well.

(i) **Preferential Attachment (PA) Model** [4], [15]: This model encodes the "rich-get-richer" phenomenon: the more connected a node is, the more likely it is to receive new links [3]. At each time step, the probability p_v that a new node u connects to an existing node v is proportional to a chosen statistical property; for example:
- *Structural*: in-degree, out-degree, etc.;
- *Temporal*: recency, age, etc.;
- *Intrinsic*: fitness, field similarity, impact factor.

(ii) **Erdős-Rényi (ER) Model** $G(n,p)$ [16]: A directed graph over n nodes is constructed by including each possible edge (u,v), from a new node u to an existing node v, independently with probability p. The expected out-degree of each new node is $\mathbb{E}[\deg^+(u)] \approx pn$.

A. Graph Representation

Let $\mathcal{G}^{(t)} = (\mathcal{V}^{(t)}, \mathcal{E}^{(t)})$ be a directed dynamic graph at time instant t, where $\mathcal{V}^{(t)}$ is the node set and $\mathcal{E}^{(t)} \subseteq \mathcal{V}^{(t)} \times \mathcal{V}^{(t)}$ is the edge set. In dynamic network models, nodes arrive continuously, but we process arrivals in parallel batches. Each batch defines an *epoch*. Each node v carries attributes $\{\tau_v, f_v^{\text{peak}}, \ell_v, \delta_v\}$, where τ_v is its arrival epoch, f_v^{peak} its intrinsic fitness peak, ℓ_v its fitness lag duration, and δ_v its fitness peak duration. The node count $N_t = |\mathcal{V}^{(t)}|$ grows each epoch by $\Delta_t = \lceil N_t \cdot \rho \rceil$, where $\rho \in (0,1)$ is the annual growth rate.

B. Preferential Attachment (PA) Model

We adopt a node scoring technique [17] that captures the temporal (recency), intrinsic (fitness), and behavioral (neighborhood) factors together.

Throughout this subsection, u denotes a newly inserted node, $g_u \in V_t$ its generator (Definition 2), and v an existing candidate vertex. Neighborhood exploration and scoring consider v in the one- and two-hop neighborhoods of g_u; all new edges have source u and destination v.

Definition 1 (Node Score)**.** *For each new node u and each candidate node $v \in V_t$ in the existing graph, we compute a multi-factor score:*

$$s_u(v) = w_\pi^u \cdot \phi_\pi(v) + w_r^u \cdot \phi_r(v) + w_f^u \cdot \phi_f(v), \qquad (1)$$

where $\phi_\pi(v)$, $\phi_r(v)$, and $\phi_f(v)$ denote the preferential attachment, recency, and intrinsic fitness scores, respectively; and w_π^u, w_r^u, w_f^u are the corresponding weights. Each new node u draws these weights i.i.d. from $\mathrm{Uniform}(0,1)$ *and normalizes them:*

$$w_\pi^u + w_r^u + w_f^u = 1, \quad w_\cdot^u \geq 0. \tag{2}$$

Definition 2 (Generator Node)**.** *Each new node u selects a generator node $g_u \in V_t$ uniformly at random. This generator provides the initial context for modeling connections.*

C. Erdős-Rényi (ER) Model $G(n,p)$

1) Edge Formation Rules: For each new node u arriving in epoch t, the kernel independently tests every existing node $v \in V^{(t-1)}$ as a potential connection target. A directed edge $(u \to v)$ is created if and only if a uniformly distributed random variate $U_{u,v} \sim \mathrm{Uniform}(0,1)$ falls below the edge probability p: $(u \to v) \in E^{(t)} \iff U_{u,v} < p$, with $U_{u,v} \sim \mathrm{Uniform}(0,1)$ i.i.d. where the random variates are drawn independently for every (u,v) pair across all agents in the batch. The expected out-degree of each new node is:

$$\mathbb{E}\big[\deg^+(u)\big] \;\approx\; p \cdot \Big|V^{(t-1)}\Big| \;\approx\; p \cdot N_{t-1}, \tag{3}$$

with no preferential, recency, or fitness bias: every existing node is an equally likely target, and the count fluctuates according to a Binomial distribution.

IV. Parallel GPU Algorithm

ParAGraph supports two GPU-parallel graph growth models that share a common epoch pipeline. In each epoch, the host computes model-specific metadata, inserts a batch of $\Delta_t = \lceil N_t \rho \rceil$ new agents, launches GPU kernels to materialize edges, and bulk-inserts the resulting edges into the graph. PA requires irregular two-hop BFS over a sparse power-law graph followed by weighted sampling, whereas ER replaces traversal with a fully parallel Bernoulli scan. Together they demonstrate that **ParAGraph** supports complementary execution patterns: irregular graph-traversal and regular data-parallel workloads.

A. Preferential Attachment ABM

Nodes are assigned contiguous integer IDs for compact storage and fast lookup. At each epoch, the host computes the PA, recency, and fitness score arrays and asynchronously transfers them to the GPU while initializing the new agents. Each agent is assigned an out-degree sampled from an empirical degree distribution, a weight triplet, a one-hop allocation ratio α_u, and a uniformly sampled generator node g_u.

1) Four-Stage GPU Pipeline and Adaptive Batching: The Δ_t new nodes are processed in mini-batches of size $B \leq B_{\max}$, sized to satisfy the memory-iso constraint $B \times \lceil N_t/32 \rceil \leq \mathcal{M}_{\mathrm{BFS}}$: as N_t grows, B shrinks proportionally, trading concurrency for GPU residency by bounding the BFS memory footprint $\mathcal{M}_{\mathrm{BFS}}$. Each mini-batch executes four kernels separated by synchronization barriers, allowing stage-local memory to be released immediately after use. A fused single-kernel implementation is evaluated as an ablation (§VI-F), while the staged pipeline is the default.

Algorithm 1: GPU-Parallel PA-based ABM

Input: Seed graph $G^{(0)}$; ρ, α_u; degree bag $\mathcal{D}$; recency map $\pi_{\mathrm{rec}}(\cdot)$; random-fill fraction f; same-epoch fraction σ; T epochs
Output: Grown graph $G^{(T)}$

Upload CSR; assign sequential IDs $0 \ldots N_{\max}-1$
$\forall u$: sample $k_u \sim \mathcal{D}$, weights w_u, split ratio α_u
for $t \leftarrow 1$ **to** T **do**
  Compute $\phi_\pi^{(t)}(v), \phi_r^{(t)}(v), \phi_f^{(t)}(v)$; upload to device
  Insert Δ_t nodes; draw g_u uniformly; sample $\mathcal{S}_t$ ($\lfloor \sigma \Delta_t \rfloor$ same-epoch sources)
  for *each mini-batch $\mathcal{B}_b$ of size B* **do**
    Alloc 3 bitmap slabs (cleared warp-strided in Stage 1)
    Stage 1 GPU $\|\, \forall u$: warp-cooperative bidirectional BFS from $g_u \Rightarrow \mathcal{N}^1(g_u), \mathcal{N}^2(g_u)$
    Sync; **free** bitmap slabs
    $k_u^{\mathrm{rnd}} \leftarrow \lfloor f k_u \rfloor$;
    $r_u \leftarrow k_u - 1 - [u \in \mathcal{S}_t] - k_u^{\mathrm{rnd}}$
    Stage 2 GPU $\|\, \forall u$: if $u \in \mathcal{S}_t$ add one same-epoch edge; $k_u^1 \leftarrow \min(\lceil \alpha_u r_u \rceil, |\mathcal{N}^1|)$; Weighted-Sampling over $\mathcal{N}^1$ via min-heap
    Sync; free 1-hop data
    Stage 3 GPU $\|\, \forall u$: $k_u^2 \leftarrow \min(r_u - k_u^1, |\mathcal{N}^2|)$; Weighted-Sampling over $\mathcal{N}^2$ via min-heap
    Sync; free 2-hop data
    Stage 4 GPU $\|\, \forall u$: deduplicated uniform fill of the residual budget; write (u, g_u) and all $(u, \cdot)$ to edge buffer
    Sync; transfer edges to host
  end
  Batch-insert edges (parallel fwd + bwd); update in/out-degrees, fitness
  if $\Delta_t > B_{\max}$ **then** $B_{\max} \leftarrow \lceil B_{\max}(1-0.5\rho) \rceil$
end

Stage 1 (BFS). Each agent expands its generator node using a bitmap-based BFS, producing the one-hop and two-hop neighborhoods, $\mathcal{N}^1(g_u)$ and $\mathcal{N}^2(g_u)$. A conventional thread-per-agent mapping suffers severe warp divergence because power-law neighborhoods vary widely in size; profiling shows only approximately 7% of warp lanes perform useful work. **ParAGraph** instead uses a **warp-cooperative BFS**, assigning one warp to each agent so lanes collaboratively expand a shared frontier using bitmap operations and `atomicOr`.

Stages 2-3 (Weighted Sampling). The one-hop and two-hop neighborhoods are processed independently. Threads compute the composite score (Eq. (1)) and use a numerically stable logarithmic formulation of the Efraimidis-Spirakis weighted sampling algorithm [18], maintaining register-resident heaps to select the top-k_u^1 and top-k_u^2 neighbors. Intermediate buffers are released immediately after each stage.

Stage 4 (Random Fill). Any remaining edge budget is filled using deduplicated uniform sampling over existing nodes. The completed edge lists are written to preallocated buffers and transferred to the host in a single bulk copy, after which the host updates the forward and reverse adjacency structures and adjusts the maximum batch size for the next epoch.

Since newly added nodes do not connect to one another within the same epoch, graph updates are deferred to the host,

Algorithm 2: GPU-Parallel ER $G(n,p)$ ABM

Input: Seed graph $G^{(0)}$; edge probability p; T epochs
Output: Grown graph $G^{(T)}$

Upload CSR; assign sequential IDs
for $t \leftarrow 1$ **to** T **do**
  Insert Δ_t nodes; alloc edge buffers (capacity $2p \cdot N_t$)
  `// All` Δ_t `nodes in one launch`
  GPU $\| \forall u \in \Delta_t$:
    Init per-thread RNG from ($t \times 12345$, `idx`)
    for $v \leftarrow 0$ **to** $N_t - 1$ **do**
      Draw $U_{u,v} \sim \mathrm{Uniform}(0,1)$
      if $U_{u,v} < p$ **then** append (u,v) to edge buffer
    end
  Sync; transfer edges to host
  Batch-insert edges (parallel fwd + bwd); update attributes
end

eliminating expensive dynamic GPU memory allocation and in-place CSR updates.

B. Erdős-Rényi ABM

ER reuses PA's host-side epoch pipeline but eliminates BFS, score computation, and weighted sampling. Since every existing node is an equally likely target, each GPU thread independently scans the current graph and inserts edge (u,v) with probability p. All new agents are processed in a single kernel launch, and memory usage is dominated only by the output edge buffers, remaining well within GPU memory.

C. Sequential Node ID Renaming and CSR Layout

ParAGraph begins with a one-time relabeling pass that maps every node to a contiguous identifier $0, \ldots, N-1$ via a bijection constructed in $\mathcal{O}(N)$ time during graph ingestion, where N is the graph size. This transformation enables every node-keyed data structure in *GPU-Std* (§IV-D) to use direct indexing instead of associative lookup: since a node's identifier *is* its array index, all node-ID lookups reduce to deterministic address computation, eliminating hashing, collision handling, and pointer chasing. The BFS *visited* set is one direct beneficiary, implemented as a bitmap rather than a hash set.

Each bitmap word stores the membership bits of 32 consecutive node identifiers, so node i maps directly to word $\lfloor i/32 \rfloor$ and bit position $i \bmod 32$ (as shown in Figure 2b). In the implementation, these indices are computed using the bitwise operations `i` $\gg$ `5` and `i` & `31`, replacing integer division and modulo with inexpensive bit operations. Membership is then determined by a single bit test at this location; insertion updates the same bit using an `atomicCAS`-based retry loop. Both operations therefore execute in $\mathcal{O}(1)$ time without hashing or probing. As shown in Figure 2, with sparse identifiers, a membership query requires computing a hash, probing a hash table (with possible collisions), and retrieving the corresponding entry. After sequential renaming, the same query reduces to two integer instructions that compute the bitmap word and bit position directly, followed by a single memory access.

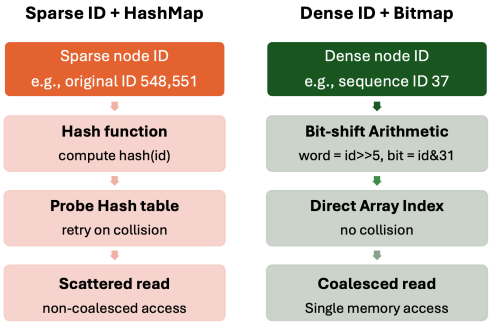
The same dense-ID invariant also enables *GPU-Std*'s CSR graph representation. The host-side adjacency structure is converted once into CSR arrays `offsets[`$N{+}1$`]` and `edges[`M`]`, which are transferred to the device with contiguous memory copies and maintained as persistent forward and reverse `DeviceGraph` instances requiring $\mathcal{O}(N+M)$ storage. Since the CSR row index is the node ID itself, the adjacency list of node i is obtained directly from `offsets[i]` in worst-case $\mathcal{O}(1)$ time. A single $\mathcal{O}(N)$ preprocessing pass enables direct indexing across the bitmap, CSR graph, neighborhood buffers, and all node-keyed data structures.

D. GPU-Std: GPU Data Structures

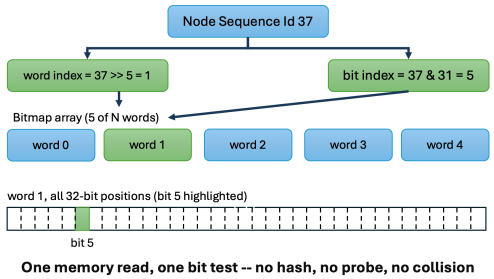
We provide *GPU-Std*, a portable, header-only, open-source CUDA library that encapsulates the low-level memory-management and synchronization primitives required by irregular graph algorithms into reusable `std`-style GPU-native data structures. *GPU-Std* provides deterministic layouts, allocation-free execution, and an explicit choice of lookup mechanism based on key-space density, enabling efficient reuse beyond **ParAGraph**.

- **`device_vector`**: A bitmap-backed set of dense node IDs supporting $\mathcal{O}(1)$ membership, erase, and amortized $\mathcal{O}(1)$ insert via direct 32-bit word addressing, eliminating hashing and probing. Positional access is supported but requires a bitmap scan. One bit per ID reduces memory by up to $32\times$ versus an integer array.
- **`device_vector_generic<T>`**: A templated, extensible, contiguous array for arbitrary typed payloads (e.g., scores, weights, and edge structures). This container provides a general-purpose abstraction for data that are not naturally node-ID keyed.
- **`device_min_heap`**: A register-resident, per-thread top-k priority queue implemented as a size-k min-heap. The heap retains the k highest-scoring candidates by evicting the current minimum only when a larger value arrives, supporting $\mathcal{O}(\log k)$ insertion and in-place $\mathcal{O}(k \log k)$ extraction. Because all heap state remains in registers throughout Stage 2/3 weighted sampling, candidate scoring requires no global-memory accesses.
- **`device_map`**, **`device_set`**: Wrappers over `cuco` [19] static hash tables providing amortized $\mathcal{O}(1)$ lookup and membership via open-addressed probing. The hash table maps keys to indices in a separate contiguous value array, introducing the indirection inherent to generic associative containers. *GPU-Std* reserves these containers exclusively for non-dense keys (e.g., `device_string`-valued attribute names), while dense node IDs bypass hashing entirely through direct indexing.
- **`DeviceGraph`**: A copyable CSR graph representation (`edges`, `vertices`, `offsets`) that exploits dense sequential node IDs as CSR row indices, enabling $\mathcal{O}(1)$ adjacency lookup and coalesced neighbor traversal via `__ldg`. By eliminating the node-ID-to-row mapping entirely, neighbor access requires only direct array indexing rather than associative lookup.

Dense node IDs are consistently exploited across all node-keyed structures including graph representation, visited sets,



(a) Sparse node IDs require hashing, collision handling, and associative lookup, whereas dense sequential IDs permit direct-indexed bitmap access through simple address arithmetic.



(b) Bitmap membership computation. Node ID 37 maps directly to word 1 and bit 5 using bit-shift arithmetic; testing or setting this bit completes the membership operation.

Fig. 2: Dense sequential node IDs replace hash-table lookup with direct-indexed bitmap operations.

and neighborhood buffers, so that a single renaming pass at ingestion enables direct-indexed $\mathcal{O}(1)$ access throughout the pipeline. Only inherently non-dense keys (e.g., attribute names or arbitrary payloads) use generic or hash-based containers. *GPU-Std* eliminates dynamic allocation, reduces global memory traffic, and improves warp efficiency through coalesced direct-indexed memory accesses. Together, sequential renumbering and GPU-native data-structures replace associative-container lookups and pointer chasing with direct array indexing, reducing global memory traffic, and maximizing memory coalescing.

V. Implementation Details

A. Bitmap BFS-State Compaction

Each agent's BFS state comprises three bitmap slabs (`visited`, `frontier_curr`, `frontier_next`) of $W{=}\lceil N_t/32\rceil$ `uint32_t` words, node v mapping to bit $(v \bmod 32)$ of word $\lfloor v/32\rfloor$. All slabs for a mini-batch are bulk-allocated once as three contiguous arrays and sliced by pointer offset, avoiding per-agent allocation. Under the default warp-cooperative path the slabs are cleared inside Stage 1 in a warp-strided loop, overlapping initialization with the traversal itself; the thread-per-agent path instead zeroes them host-side with three `cudaMemset` calls. At $B{=}24{,}000$, $N{=}491{,}532$, the slabs occupy **4.1 GB** vs. **132 GB** for a naïve layout - a **32**× per-agent reduction (**180 KB** vs. **5,760 KB**).

B. Adaptive Batch Sizing

As N grows, the per-batch slab cost increases. Whenever $\Delta_t > B_{\max}$, the ceiling is updated as $B_{\max} \leftarrow \lceil B_{\max}(1{-}0.5\rho)\rceil$, throttling concurrency to preserve the memory-iso invariant $B \times \lceil N/32\rceil \le \mathcal{M}_{\text{BFS}}$ and keep the simulation fully GPU-resident.

C. Out-Degree Calibration

To enable direct comparison between PA and ER, we match the expected out-degree. Under ER, choosing $p = 0.05\%$ on the approximately 0.5M-node seed graph yields an expected out-degree of about 250, matching PA's out-degree cap ($k_{\max}$) and providing the calibration used in §VI-H.

VI. Experiments

We evaluate **ParAGraph** against three implementations: (i) a 16-thread OpenMP CPU implementation (CPU-16T), (ii) our GPU-optimized implementation, and (iii) MASS CUDA [8] as a representative general-purpose GPU ABM framework. The CPU implementation is functionally equivalent to the GPU pipeline, implementing the same PA/ER scoring and growth semantics, and is verified to produce statistically indistinguishable output distributions (§VI-C). It is an optimized OpenMP implementation representative of competent multicore CPU execution, providing a consistent baseline for evaluating the benefits of GPU acceleration.

A. Experimental Setup

We evaluate **ParAGraph** on two representative GPU/CPU platform pairs rather than using a single CPU baseline across GPU generations: (i) an **NVIDIA H100 SXM5** (Hopper, 132 SMs, 80 GB HBM3, 2.0-3.4 TB/s memory bandwidth) paired with an **Intel Xeon Platinum 8480+** (56 cores @ 3.8 GHz), and (ii) an **NVIDIA RTX 4090** (Ada Lovelace, 128 SMs, 24 GB GDDR6X, 1.0 TB/s memory bandwidth) paired with an **Intel Core i9-14900KF** (24 cores/32 threads, 36 MB L3, 64 GB RAM). The H100/Xeon platform represents cluster-scale HPC systems optimized for memory capacity and bandwidth, while the RTX 4090/Core i9 platform represents a widely accessible, lower-cost desktop workstation. The H100 provides substantially higher memory capacity and bandwidth, whereas the RTX 4090 delivers higher peak FP32 throughput (∼82.6 TFLOPS versus ∼51-67 TFLOPS), allowing us to evaluate **ParAGraph** across bandwidth and compute-oriented architectures. To ensure fair GPU-CPU comparisons independent of host hardware, each platform uses its own CPU baseline capped at 16 threads (CPU-16T), despite the differing total core counts of the Xeon and Core i9 processors.

B. Seed Graph

We evaluate **ParAGraph** using the Stahl-Johnstone citation network [17], a hub-dominated graph that presents a challenging workload for GPU-accelerated graph simulation and has

previously been studied only in sequential settings [17], [20]. Starting from this common seed graph, we evaluate PA and ER models. The ER model is evaluated at three probabilities ($p \in \{0.01\%, 0.05\%, 0.1\%\}$), spanning sparse to dense growth, with $p = 0.05\%$ calibrated to match PA's expected out-degree.

C. Degree-Distribution Fidelity

To verify GPU-accelerated **ParAGraph** and its MASS CUDA port faithfully reproduce the reference CPU ABM, we ran all three for 10 independent runs with identical seeds, comparing node-level in-degree, out-degree, and fitness-conditioned degree distributions rather than aggregate statistics alone. Nodes are grouped by type (*seed* or *agent*) and degree bin ($f1$-$f6$), reporting the pooled distribution plus mean±std of per-run means to separate stochastic variability from implementation differences.

Figure 3 pools in-degree ($f1$-$f6$) and out-degree ($f1$-$f3$) across 10 runs for all three implementations; all reproduce the CPU reference to within run-to-run noise, statistically indistinguishable across every bin and node type, including the extreme $f6$ bin (n=10/run, mean in-degree $\approx 1.57 \times 10^5$).

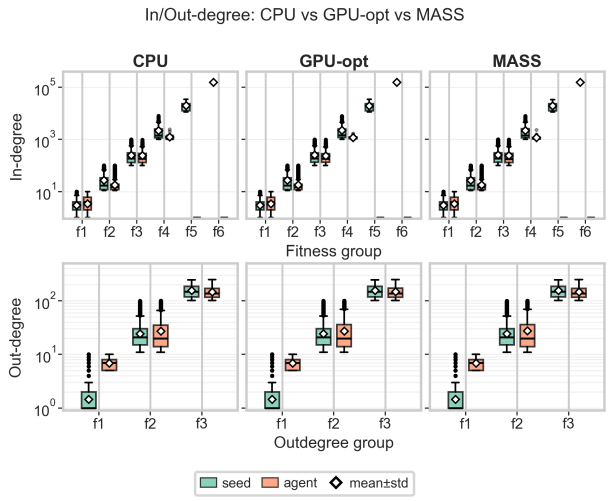


Fig. 3: Pooled in-degree (top) and out-degree (bottom) distributions across 10 runs for CPU, **ParAGraph**, and MASS CUDA, grouped by fitness/degree bin and node type (seed/agent). Diamond markers show mean ± std. deviation across runs.

D. Performance Results

On each platform we compare against its own CPU-16T baseline and a MASS CUDA implementation using fixed-size batches, versus ParAGraph's adaptive batching (§IV-A1). Figure 4 reports runtime and speedup at 1%/3%/6% growth over 30 epochs.

Execution time. At 6% growth, CPU-16T requires 16.0 hours (Xeon) and 11.2 hours (i9); **ParAGraph** completes the same workload in **40.7 minutes** (H100) and **27.7 minutes** (RTX 4090). MASS CUDA is consistently slower (57.0, 54.1 minutes).

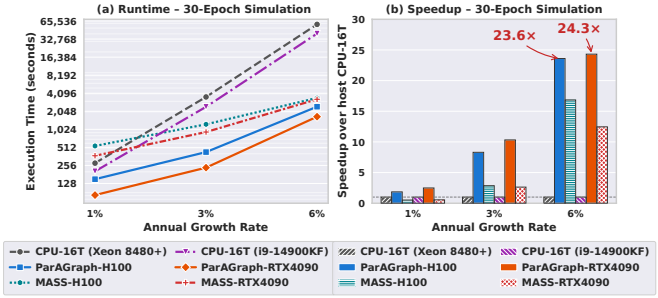


Fig. 4: **Runtime (a) and speedup over each platform's CPU-16T host baseline (b)**, 30-epoch horizon. Peak speedup of **23.6**× (H100) and **24.3**× (RTX 4090) is reached at 6% growth.

Speedup. ParAGraph reaches 2.8×, 8.3×, and **23.6**× speedup on H100 and 2.5×, 10.3×, and **24.3**× on RTX 4090 at 1%, 3%, and 6% growth, respectively. The nearly identical peak speedups across these two GPUs indicate that the gains stem primarily from the algorithmic formulation (§IV) rather than GPU-specific characteristics. **ParAGraph** outperforms MASS CUDA at every growth rate on both platforms, although the gap *narrows* with increasing growth as MASS's fixed per-batch overhead is amortized. At 6% growth, ParAGraph reduces the most demanding 30-epoch run from 11 to 16 hours on CPU to under 41 minutes on both the datacenter-class H100 and consumer RTX 4090.

E. Scalability Analysis

ParAGraph's speedup rises monotonically with horizon at both growth rates on both platforms, peaking at 30-epoch/6% (23.6× H100, 24.3× RTX 4090). MASS CUDA trails throughout, by up to 6.5× at 10-epoch/6% on H100, and dips briefly at 10-epoch/3% before recovering by 30 epochs, consistent with MASS's extra indirection over native CUDA kernels as shown in Figure 5. To compare speedup across devices with very different core counts, we define efficiency (ε) as $\varepsilon = \text{speedup} \times 16/p$, where p is the GPU's SM count; dividing by p and rescaling by the CPU-16T thread count normalizes the CPU-16T baseline to $\varepsilon = 1$ regardless of platform. By this measure, ε only exceeds the CPU-16T-equivalent line ($\varepsilon > 1$) from 10-epoch/6% growth onward, reaching 3.31× (H100) and 3.04× (RTX 4090) at 30-epoch/6%; MASS stays below this line except at the same largest scenario (2.36× H100, 1.56× RTX 4090).

Cross-platform portability. ParAGraph's speedup range is nearly identical across two very different GPUs, 2.13-23.6× on H100 versus 2.29-24.3× on RTX 4090, under 3% apart at peak, evidence the advantage comes from algorithm-architecture co-design, not GPU-specific properties. MASS is far less portable: its ranges (1.11-16.8× H100, 1.14-12.5× RTX 4090) trail substantially, and the ParAGraph/MASS gap on *identical* hardware varies more (1.19-6.54× H100, 1.81-3.93× RTX 4090) than ParAGraph's own speedup varies across GPUs; GPU choice matters less to ParAGraph than software architecture matters to MASS.

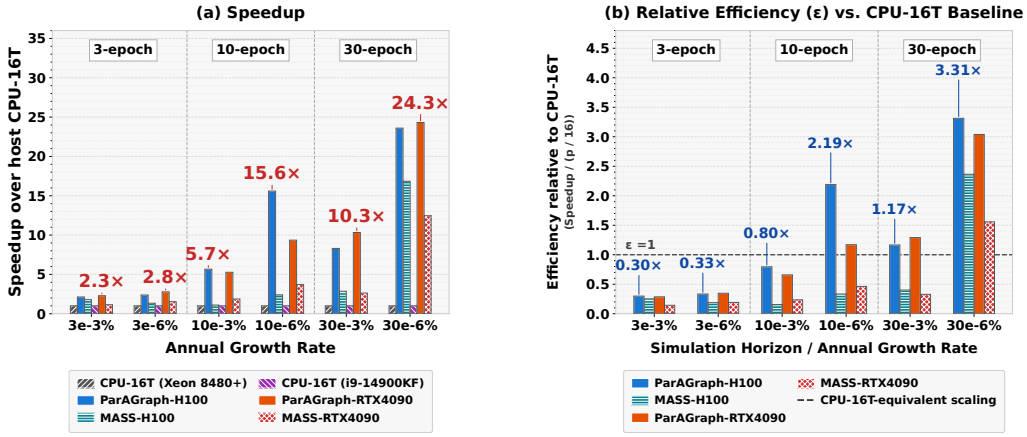


Fig. 5: **Strong-scaling speedup (a) and Relative efficiency (ε) vs. CPU-16T baseline (b)**, on H100 and RTX 4090, across horizons (3/10/30 epochs) and growth rates (3%, 6%). Relative Efficiency (ε) is rescaled so CPU-16T equals 1.0 by normalization (Speedup / (SM count / 16)); bars above the dashed line beat naive core-count scaling.

F. Ablation Studies

To isolate each design decision's contribution (§IV), we evaluate six configurations on H100 at a 10-epoch horizon, one axis at a time, each running at $\rho \in \{1\%, 3\%, 6\%\}$ against CPU-16T (Xeon 8480+): (i) **Baseline** uses the full default pipeline - warp-cooperative BFS, adaptive mini-batching with an initial ceiling $B_{\max} = 100{,}000$, and the staged four-kernel execution model; (ii) **Default Batch** is identical to Baseline but with a smaller batch-size ceiling ($B_{\max} = 20{,}000$), isolating the effect of the ceiling itself; (iii) **No Warp** reverts Stage 1 to thread-per-agent dispatch, keeping the same $B_{\max} = 20{,}000$ batching and staged pipeline as Default Batch, isolating the warp-cooperative BFS's contribution; (iv) **No Batching** disables mini-batching entirely, processing each epoch's full cohort in a single pass, while retaining warp-cooperative BFS and the staged pipeline, isolating adaptive batching's contribution; (v) **Large Batching** relaxes the ceiling to $B_{\max} = 1{,}000{,}000$ while retaining warp-cooperative BFS and the staged pipeline, representing a large-batch regime; and (vi) **Monolithic** fuses all four kernel stages into a single launch and reverts to thread-per-agent dispatch, with the same $B_{\max} = 20{,}000$ batching as Default Batch, isolating the staged-vs-fused pipeline design.

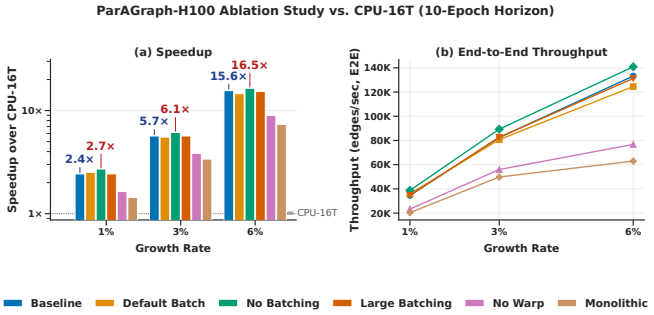


Fig. 6: **Ablation Study: speedup over CPU-16T (a) and throughput (b) vs. growth rate**, six H100 configurations. *No Warp* and *Monolithic* form a lower band in both panels.

Throughput and speedup. Figure 6 shows speedup ranging from $1.44\times$ (*Monolithic*, 1%) to $16.46\times$ (*No Batching*, 6%) across all 18 pairs. *Baseline*, *Default Batch*, *No Batching*, and *Large Batching* cluster within $\sim 13\%$ of each other at every growth rate; *No Warp* and *Monolithic* fall well below - isolating the warp-cooperative BFS (whose gap to *No Warp* widens from $1.48\times$ to $1.74\times$ as ρ grows, exposing more divergence).

Batching is a memory-safety mechanism, not primarily a throughput one, here. *No Batching* leads at every growth rate, but its margin over Baseline *narrows* as ρ grows ($11.5\% \to 8.1\% \to 5.5\%$) rather than widening: an unbounded pass forfeits the concurrency-for-residency trade mini-batching makes, and that trade grows more valuable as Δ_t grows. At this 10-epoch horizon, Δ_t never exceeds the GPU's memory budget, so *No Batching*'s advantage is real but not evidence mini-batching is unnecessary in general; we recommend *Large Batching* as a practical default beyond the regimes tested, trading a small throughput cost against an OOM risk this horizon is too short to trigger.

Warp divergence, thread utilization, and occupancy. Table II confirms that the warp-cooperative rewrite outperforms thread-per-agent dispatch on every metric, across both GPUs and growth rates. The clearest gain is grid utilization: assigning one warp per agent instead of one thread expands Stage 1's grid by 32-34×, increasing waves/SM by the same factor. The no-warp kernel never reaches one-third of a wave across either device's SMs, whereas the warped kernel reaches 1.9-9.7× waves/SM. Thread packing improves more modestly: active threads/warp increase by 27-32% on H100 and 32-44% on RTX 4090, while non-predicated threads increase by 34-37% and 49-53%, respectively, still well below full 32-lane utilization. Branch uniformity, already above 99.4% without warping, reaches 99.98-99.99% with warping, confirming that lane inactivity is dominated by predication rather than branch divergence. Achieved occupancy moves in opposite directions

TABLE II: Stage 1 BFS comparison, warp-cooperative (*baseline*) vs. original (*no-warp*), on H100 and RTX 4090 at a 10-epoch horizon, 3% and 6% growth.

	H100				RTX 4090			
Metric	no-warp(3%)	warp(3%)	no-warp(6%)	warp(6%)	no-warp(3%)	warp(3%)	no-warp(6%)	warp(6%)
Grid size (blocks)	35.6	1181.2	75.4	2547.8	36.0	1229.6	76.4	2487.3
Waves/SM	0.135	4.474	0.286	9.652	0.056	1.921	0.118	3.887
Active threads/warp (/32)	2.26	2.98	2.93	3.72	2.26	3.25	2.86	3.99
Non-pred. threads/warp (/32)	2.11	2.89	2.72	3.63	2.04	3.12	2.56	3.82
Branch uniformity (%)	99.81	99.99	99.46	99.99	99.83	99.98	99.53	99.98
Stage 1 time share (%)	88.41	82.40	87.11	82.03	77.36	71.59	76.29	69.25

across architectures—higher for no-warp on H100 and higher for warped on RTX 4090—without changing the register-bound ceiling (5 blocks/SM on RTX 4090 and 2 blocks/SM on H100). This is consistent with the waves/SM results, since achieved occupancy measures only the density of active warps on resident SMs. Stage 1's share of kernel time also decreases after the rewrite (H100: $88.4\% \rightarrow 82.4\%$, $87.1\% \rightarrow 82.0\%$; RTX 4090: $77.4\% \rightarrow 71.6\%$, $76.3\% \rightarrow 69.3\%$), not because Stage 1 regresses, but because it becomes proportionally faster while downstream costs remain fixed.

G. GPU profiling

To identify what actually drives runtime at scale, we instrumented the full ParAGraph (*baseline*) pipeline across all growth rates on a **10-epoch parallel GPU run**, breaking wall-clock time into eleven stages.

TABLE III: Time breakdown by growth rate, parallel GPU run, 10-epoch horizon (ms, % of total).

Component	**1%**	**3%**	**6%**
Host preprocess	5523 (19.3%)	5806 (14.7%)	6041 (10.7%)
CSR build	7042 (24.6%)	9768 (24.8%)	14121 (25.0%)
Stage1 BFS	5857 (20.5%)	7110 (18.1%)	8372 (14.8%)
Stage2+3	1221 (4.3%)	1531 (3.9%)	2085 (3.7%)
Stage4	24 (0.1%)	32 (0.1%)	43 (0.1%)
Transfers	1079 (3.8%)	3488 (8.9%)	8013 (14.2%)
Host graph update	1313 (4.6%)	3249 (8.3%)	6142 (10.9%)
ABM construction	968 (3.4%)	966 (2.5%)	966 (1.7%)
Graph parse/init	3301 (11.5%)	3347 (8.5%)	3306 (5.8%)
Output write	951 (3.3%)	1293 (3.3%)	1850 (3.3%)
Other Cost	1311 (4.6%)	2784 (7.1%)	5643 (10.0%)
Pipeline total	**28590**	**39374**	**56582**

As shown in Table III, GPU kernel time (Stage1+2+3+4) is **comparable in order of magnitude to the major host-side stages, not dwarfed by them**: kernels run 18.6-24.8% of total time versus 35.6-43.9% for Host preprocessing+CSR build (a $1.8\text{-}1.9\times$ gap) and 8.4-25.0% for Transfers+Host graph update (narrowing to $1.35\times$ at 6% growth) - the signature of a pipeline already optimized near its ceiling, where further GPU tuning has limited room left to move total runtime. CSR build holds a stable $\approx 25\%$ share throughout while doubling in absolute cost; Transfers and Host graph update grow 643% and 368%, far outpacing the pipeline's own 98% growth, together approaching CSR build's share by 6% growth. ABM construction and Graph parse/init are fixed one-time costs that shrink relatively as the pipeline grows around them; Stage4 stays negligible at

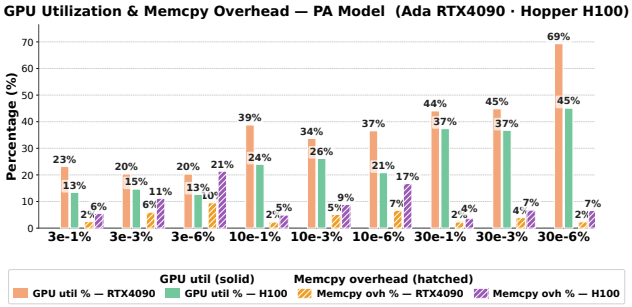


Fig. 7: GPU utilization (%) and memcpy overhead (%) on RTX 4090/H100 across 9 configurations (3e, 10e, 30e for growth rates 1%, 3%, 6%). Utilization peaks at **69%** (RTX 4090) and **45%** (H100) at 30e-6%; memcpy overhead falls to **2%** on RTX 4090 but stays at **7%** on H100.

0.1% throughout. This trend is consistent with H100 utilization remaining below 50% at the largest scale (Figure 7).

Utilization. Utilization rises monotonically with horizon at every growth rate on both GPUs, reaching 69% (RTX 4090) and 45% (H100) at 30e-6%; the growth-rate trend at fixed horizon is noisier (e.g. H100 at 30 epochs dips $37.4\% \rightarrow 36.8\%$ before recovering to 45.1%). Memcpy overhead falls to 2% on RTX 4090 at 30e-6%, a clear compute-bound transition H100 does not show at the same point (7%). This scaling is driven by `Stage1`; memory overhead stays low throughout.

Occupancy. As shown in Figure 8, at 30e-6%, `Stage1`+`Stage3` account for 89.2% of kernel time on H100 and 90.9% on RTX 4090, register-limited on both GPUs, though H100 weights `Stage1` $2.75\times$ over `Stage3` while RTX 4090 splits nearly evenly ($1.25\times$). Register-limited occupancy is *higher* on RTX 4090 (83%) than H100 (62%) for these kernels not because RTX 4090 has more register headroom (both share a 65,536-register/SM budget), but because `Stage1`'s register count (44 H100, 48 RTX 4090) rounds up to the same 1536-register/warp allocation unit on both, yielding an identical 40 resident warps/SM; RTX 4090's smaller warp ceiling (48 vs. H100's 64) turns that into a larger percentage. Utility kernels reach 100% occupancy on both GPUs, confirming that BFS stages, not the auxiliary pipeline, remain the bottleneck on both. Together, these results demonstrate that **ParAGraph** scales efficiently: utilization increases monotonically with scale, execution remains compute-bound despite graph irregularity, and occupancy is consistent

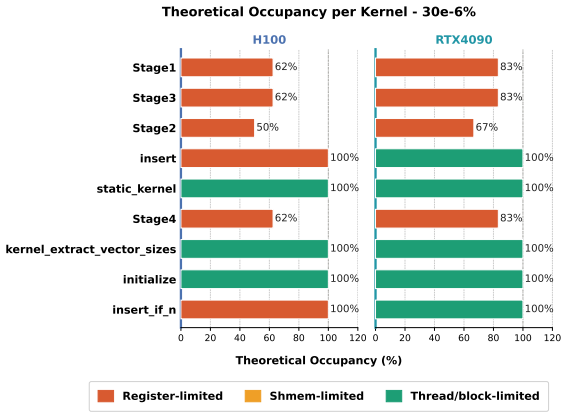


Fig. 8: Theoretical SM occupancy per kernel at 30e-6% for H100 and RTX 4090. Bar colour is the occupancy limiter (register/shmem/thread-block). `Stage1-Stage4` are register-limited on both GPUs; utility kernels reach 100%.

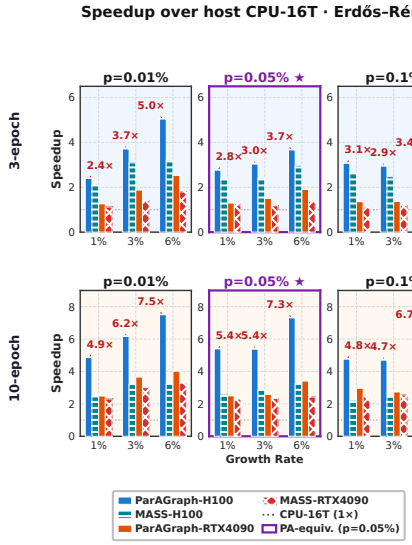


Fig. 9: **GPU speedup of ER-p for ParAGraph and MASS CUDA, on H100 and RTX 4090**, vs. each platform's own CPU-16T baseline, across two horizons, three edge probabilities ($p \in \{0.01\%, 0.05\%, 0.1\%\}$), and three growth rates. The p=0.05% column ($\star$) is the PA-equivalent setting ($\mathbb{E}[\deg^+] \approx 250$).

across GPU generations, with BFS register pressure reflecting an architectural constraint rather than an algorithmic limitation.

H. Erdős-Rényi Model: Speedup Across Edge Probabilities

For ER, we evaluate only GPU-optimized **ParAGraph** and MASS CUDA. ER eliminates the irregular two-hop BFS, yielding a uniform, divergence-free workload that exposes the pipeline's peak throughput, an upper bound on parallelism absent graph-traversal overhead. Per §V-C, p=0.05% yields the same mean degree as PA, enabling direct comparison between the two models.

Figure 9 shows that **ParAGraph** outperforms MASS CUDA on both GPUs across all configurations. On H100, **ParAGraph** achieves 2.4-**7.5**× speedup, peaking at 7.5× for 10 epochs and p=0.01% at 6% growth, while MASS remains below 3.25×. At the PA-equivalent point, ParAGraph reaches 7.3× versus 3.2× for MASS, a **2.3**× advantage on identical hardware. RTX 4090 shows the same trend at a lower absolute level, with **ParAGraph** peaking at 4.0× versus 3.3× for MASS, both at p=0.01%. Increasing edge probability does not improve speedup: at 10 epochs and 6% growth on H100, speedup decreases from 7.5× at p=0.01% to 6.7× at p=0.1%. At the PA-equivalent setting, ER achieves 1.9-2.5× higher speedup than PA because its sampling avoids BFS traversal and the associated register-pressure bottleneck (§VI-G).

I. Limitations and Future Work

ParAGraph currently supports append-only graph growth: nodes and edges can be inserted but not deleted. Its dense sequential-ID layout enables the bitmap compaction described in §V, but deletion would require tombstoning with periodic compaction or a sparse-ID indirection layer, reducing the benefits of direct-indexed bitmap access. Supporting dynamic graph updates and sparse identifiers, extending evaluation to additional graph models, and maintaining the graph entirely on the GPU are natural directions for future work. Extending **ParAGraph** to multiple GPUs is another natural next step, requiring dynamic load balancing for uneven graph growth, communication-aware partitioning (e.g., METIS-style [21]), and computation-communication overlap to preserve locality and scalability across devices.

VII. Conclusions

We presented **ParAGraph**, a GPU-native framework for large-scale ABMs that co-designs data representation, scheduling, and parallel algorithms to transform an irregular, memory-bound workload into a high-throughput GPU pipeline. **ParAGraph** consistently outperforms MASS CUDA on identical hardware, by up to 6.5× on H100, an advantage attributable to algorithmic co-design rather than hardware alone, while delivering up to 24.3× speedup over a 16-thread CPU baseline and exceeding CPU-16T-equivalent efficiency by up to 3.3× at scale, with peak speedups reproducing within 3% across a datacenter H100 and a consumer RTX 4090. The primary bottleneck in PA is irregular two-hop BFS traversal, a cost the Erdős-Rényi model eliminates entirely: **ParAGraph** reaches up to 7.5× speedup under ER, within 3% of its speedup at the PA-equivalent setting (p=0.05%). At matched expected degree, ER is itself 1.9-2.5× faster than PA, directly quantifying BFS traversal's overhead. A clear hierarchy emerges: CPU parallelism is bandwidth-limited; general-purpose GPU frameworks cannot exploit irregular graph structure; **ParAGraph**, through co-design, addresses both these drawbacks.

REFERENCES

[1] E. Bonabeau, "Agent-based modeling: Methods and techniques for simulating human systems," *Proceedings of the National Academy of Sciences*, vol. 99, no. suppl_3, pp. 7280–7287, 2002. [Online]. Available: https://doi.org/10.1073/pnas.082080899

[2] C. M. Macal and M. J. North, "Tutorial on agent-based modeling and simulation," in *Proceedings of the 37th Conference on Winter Simulation*, ser. WSC '05. Winter Simulation Conference, 2005, p. 2–15. [Online]. Available: https://doi.org/10.1109/WSC.2005.1574234

[3] A.-L. Barabási and R. Albert, "Emergence of scaling in random networks," *Science*, vol. 286, no. 5439, pp. 509–512, 1999. [Online]. Available: https://doi.org/10.1126/science.286.5439.509

[4] D. J. d. S. Price, "Networks of scientific papers," *Science*, vol. 149, no. 3683, pp. 510–515, 1965. [Online]. Available: https://doi.org/10.1126/science.149.3683.510

[5] Y. Wang, A. Davidson, Y. Pan, Y. Wu, A. Riffel, and J. D. Owens, "Gunrock: a high-performance graph processing library on the gpu," in *Proceedings of the 21st ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, ser. PPoPP '16. New York, NY, USA: Association for Computing Machinery, 2016. [Online]. Available: https://doi.org/10.1145/2851141.2851145

[6] RAPIDS Development Team, *RAPIDS: Libraries for End to End GPU Data Science*, 2023. [Online]. Available: https://rapids.ai

[7] P. Richmond and M. K. Chimeh, "Flame gpu: Complex system simulation framework," in *2017 International Conference on High Performance Computing and Simulation (HPCS)*, 2017, pp. 11–17. [Online]. Available: https://doi.org/10.1109/HPCS.2017.12

[8] L. Kosiachenko, N. Hart, and M. Fukuda, "Mass cuda: A general gpu parallelization framework for agent-based models," in *Advances in Practical Applications of Survivable Agents and Multi-Agent Systems: The PAAMS Collection*, Y. Demazeau, E. Matson, J. M. Corchado, and F. De la Prieta, Eds. Cham: Springer International Publishing, 2019, pp. 139–152. [Online]. Available: https://doi.org/10.1007/978-3-030-24209-1_12

[9] J. Xiao, P. Andelfinger, D. Eckhoff, W. Cai, and A. Knoll, "A survey on agent-based simulation using hardware accelerators," *ACM Comput. Surv.*, vol. 51, no. 6, Jan. 2019. [Online]. Available: https://doi.org/10.1145/3291048

[10] F. Busato, O. Green, N. Bombieri, and D. A. Bader, "Hornet: An efficient data structure for dynamic sparse graphs and matrices on GPUs," in *Proceedings of the IEEE High Performance Extreme Computing Conference (HPEC)*, 2018. [Online]. Available: https://doi.org/10.1109/HPEC.2018.8547541

[11] H. Liu and H. H. Huang, "Enterprise: breadth-first graph traversal on gpus," in *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, ser. SC '15. New York, NY, USA: Association for Computing Machinery, 2015. [Online]. Available: https://doi.org/10.1145/2807591.2807594

[12] R. Neff, M. E. Zarch, M. Minutoli, M. Halappanavar, A. Tumeo, A. Kalyanaraman, and M. Becchi, "Fuseim: Fusing probabilistic traversals for influence maximization on exascale systems," in *Proceedings of the 38th ACM International Conference on Supercomputing*, ser. ICS '24. New York, NY, USA: Association for Computing Machinery, 2024, p. 38–49. [Online]. Available: https://doi.org/10.1145/3650200.3656621

[13] D. Merrill, M. Garland, and A. Grimshaw, "Scalable gpu graph traversal," *SIGPLAN Not.*, vol. 47, no. 8, p. 117–128, Feb. 2012. [Online]. Available: https://doi.org/10.1145/2370036.2145832

[14] J. Shun and G. E. Blelloch, "Ligra: a lightweight graph processing framework for shared memory," in *Proceedings of the 18th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP)*, vol. 48, no. 8. New York, NY, USA: Association for Computing Machinery, Feb. 2013, p. 135–146. [Online]. Available: https://doi.org/10.1145/2517327.2442530

[15] D. J. de Solla Price, "A general theory of bibliometric and other cumulative advantage processes," *Journal of the American Society for Information Science*, vol. 27, no. 5, pp. 292–306, 1976. [Online]. Available: https://doi.org/10.1002/asi.4630270505

[16] P. Erdös and A. Rényi, "On the evolution of random graphs," *Publications of the Mathematical Institute of the Hungarian Academy of Sciences*, vol. 5, pp. 17–61, 1960. [Online]. Available: https://doi.org/10.1515/9781400841356.38

[17] G. Chacko, M. Park, V. Ramavarapu, A. Grama, P. Robles-Granda, and T. Warnow, "An agent-based model of citation behavior," *Applied Network Science*, vol. 11, no. 10, 2026. [Online]. Available: https://doi.org/10.1007/s41109-025-00766-z

[18] P. S. Efraimidis and P. G. Spirakis, "Weighted random sampling with a reservoir," *Information Processing Letters*, vol. 97, no. 5, pp. 181–185, 2006. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S002001900500298X

[19] NVIDIA Corporation, "cuCollections: GPU-accelerated concurrent data structures library," 2026. [Online]. Available: https://github.com/NVIDIA/cuCollections

[20] G. Chacko, M. Park, H. Yi, and T. Warnow, "Modeling the global citation network using the scalable agent-based simulator for citation analysis with recency-emphasized sampling (SASCA-ReS)," Zenodo, 2025. [Online]. Available: https://doi.org/10.5281/zenodo.17772113

[21] G. Karypis and V. Kumar, "A fast and high quality multilevel scheme for partitioning irregular graphs," *SIAM Journal on Scientific Computing*, vol. 20, no. 1, pp. 359–392, 1998. [Online]. Available: https://doi.org/10.1137/S1064827595287997