LARGE-SCALE GRAPH ANALYSIS AND MATCHING ON PARALLEL ARCHITECTURES

Michael Mandulak

Submitted in Partial Fullfillment of the Requirements
for the Degree of

DOCTOR OF PHILOSOPHY

Approved by:
George M. Slota, Chair
Sayan Ghosh
Mark Shephard
Boleslaw K. Szymanski

Department of Computer Science
Rensselaer Polytechnic Institute
Troy, New York

[December 2025]

CONTENTS

LIST OF TABLES

LIST OF FIGURES

ACKNOWLEDGMENTS

I would first like to thank my advisor George Slota for his continued support, guidance and assistance throughout my PhD and career-aimed endeavors. Without the opportunities, platforms and advice provided by him, none of my PhD work would have been possible. I also thank you for supporting my wide range of projects over the years, giving me the freedom to explore the interesting problems contained in this thesis. I thank you sincerely for all you have done for me throughout my PhD.

I would like to give a special thank you to my advisors at Pacific Northwest National Laboratories: Sayan Ghosh, S M Ferdous and Mahantesh Halappanavar. Throughout three internships and many years of projects in between, your dedication to me from the very start has been invaluable far beyond our project accomplishments together. I look up to each of you and am forever thankful for your contributions to myself, my work and my career developments. I especially want to thank Sayan for his endless mentorship and willingness to support me – from long nights of writing to conference experiences, you have been there through it all. I cannot thank you enough.

Thank you as well to all of my friends and family who have supported me throughout my time as a PhD student, especially my parents who have always been there for me no matter the circumstances. I would also like to thank my PhD committee: Boleslaw Szymanski, Mark Shephard and Sayan Ghosh, for their advice and support throughout the completion of my PhD milestones.

My sincerest thank you goes to my wonderful life partner Amy Xie, whose endless support, love and dedication to me as a human being has changed my life and enabled me to complete my PhD work. Despite the USA–Australia distance, your patience, kindness and care throughout all of our adventures has made you my shining light. I am so lucky to be supported by you and could not have asked for a more wonderful partner to be with me through it all.

The work contained in this thesis is in parts supported by: the National Science Foundation under Grant No. 2047821; the U.S. Department of Energy (DOE), Office of Science, Office of Advanced Scientific Computing Research, Scientific Discovery through Advanced Computing (SciDAC) Program through the FASTMath Institute under Contract No. DE-SC0021285 at Rensselaer Polytechnic Institute, Troy NY; the U.S. DOE ASCR End-to-end

co-design for performance, energy efficiency, and security in AI-enabled computational science (ENCODE) project at Pacific Northwest National Laboratory (PNNL); the U.S. DOE ExaGraph project; Data-Model Convergence Initiative (DMC); the Laboratory Directed Research and Development program at PNNL. PNNL is operated by Battelle Memorial Institute under Contract DE-AC05-76RL01830.

ABSTRACT

Graphs are relational data structures used to represent a wide variety of scientific domains, ranging from biological and chemical interactions, network security, machine learning, social networks, web data and many more. As these real-world instances rapidly grow in size, scalable methods on high-performance computing architectures are required to perform efficient data analysis. However, the development of these methods is notably challenging due to several factors: the irregularity of real-world graph data, the complexity of relevant analysis algorithms and the difficulties in communication and synchronization overheads at scale.

To tackle these challenges, this thesis explores the design of parallel graph analysis methods ranging from vertex ordering to matching, with an emphasis on scalability and performance. Grouped into three primary sections of general graph analysis, matching-based methods, and a combination of the two, this thesis discusses algorithmic and implementation-based optimizations on multicore, manycore and distributed topologies. The primary contributions are as follows.

For general graph analysis, this thesis discusses contributions in both multithreaded vertex ordering and in network sensing on GPUs. First, a parallel refinement-based ordering algorithm is presented to improve cache efficiency in popular graph analysis methods, such as PageRank. In network sensing, the computation of graph properties is performed across multiple GPUs using emerging `C++26` standard parallelism. Optimizations made in both cases are shown to outperform comparison methods, while offering experimental insights into new approaches and forthcoming technologies, respectively.

In the realm of maximum weighted matching (finding vertex disjoint edge sets with maximum weight), methods are presented to efficiently implement approximation algorithms on multi-GPU topologies, with optimizations to communication and data movement. This is extended to applications in set similarity computations on web-based text data, specifically in data join operations. The multi-GPU methods yield the first results on billion edge graphs, outperforming state-of-the-art parallel implementations by up to 45×. Matching-based sequential data join results yield 20× improvement upon state-of-the-art filter-verify set similarity frameworks.

Finally, this thesis combines the notions of graph analysis and matching on distributed frameworks, relying on 2D graph processing. This aims to minimize communication over-

heads at scale, applying sparse communication patterns within popular graph analytics. These optimizations allow for the scaling of simple and complex analytics on up to 400 GPUs. The results show near theoretical scaling on graphs in the billions of edges while significantly outperforming similar distributed graph processing frameworks.

CHAPTER 1
INTRODUCTION

This thesis presents work in the domain of *large-scale graph analytics* through parallel processing, applying a wide range of techniques to innovate methods and enhance the performance of graph-based data analysis across multithreaded and manycore systems. For our purposes, we consider graph-based data as relational data represented by vertices and edges, where each edge models the relationship between two objects. These methods serve a wide range of use cases from applications to biology, chemistry, linear algebra, social networks, physics and data science, to name a few [26, 33, 38, 86, 101]. While challenging in principle through method complexities and domain applications, the requirement of solutions to be *scalable* as modern data sizes are ever increasing (graph edge counts scaling beyond the *trillions*), there is an ever increasing demand for efficient solutions to large-scale graph processing [23, 90, 127].

Coupled with typical large data challenges comes additional challenges in devising efficient parallel methods to process these large-scale graphs. Namely, real-world graph structures are often irregular in nature, leading to inefficient memory access patterns and challenging workload distributions when scheduling work among processing elements. Furthermore, different processing schemes can lead to additional overheads in synchronization and data movement, which is highly dependent on system hardware (GPU, interconnect, etc.) and algorithmic dependencies. These challenges significantly limit the performance yield from evolving modern hardware, requiring developers to carefully craft algorithms and their implementations to push the boundaries of performance in order to process graphs at such a scale.

Considering these challenges, this thesis aims to explore these demands through efficient parallel methods for a variety of graph analytic problems at a large scale. We specifically focus on problems related to basic graph analytics such as vertex ordering, network sensing and graph connectivity/traversal while including a special focus on the classical problem of weighted matching. In each instance, we explore applications of parallel architectures through the use of manycore, multicore and distributed processing to push the bounds of performance for a variety of applications.

1.1 Thesis Topics

Throughout this thesis, we explore a number of different algorithms and implementations popular in the domain of "graph analytics," with an extra emphasis on weighted matching based on relevant contributions. Thus, the topics of this thesis are split between general graph analysis methods and weighted matching, while providing a brief description of each.

1.1.1 General Graph Analytics

1.1.1.1 Vertex Ordering

The vertex ordering problem [55], which aims to rearrange vertices for more cache efficient memory access, highlights a crucial consideration in data locality among the irregularity of real-world graphs. When processing graphs at a large scale, the overheads associated with inefficient memory access severely limit scalability [89], making reordering algorithms crucial within parallel graph analytics. Since the problem is NP-hard, popular solutions to reordering take advantage of heuristics relative to cache mapping and alternative locality measures. Thus, an initial focus of the work in the problem scope targeted alternatives to the popular heuristic methods to yield performance improvements with minimal reordering.

1.1.1.2 Graph Statistics

Within alternative analysis contexts, such as with anonymized data in network security and privacy settings, the computation of basic graph statistics sensitive to data permutations requires efficient computations at a large scale [136]. In a general sense, this problem focuses on the development of graph processing methods at a large-scale with sensitivity to varying input data formats while maintaining efficiency. Towards this, this thesis explores the application of emerging technology in new C++26 standard parallelism on GPUs as a means for calculating graph statistics efficiently and with limited GPU-specific code for the most flexibility among parallel platforms.

1.1.2 2D Graph Processing

Popular frameworks for large-scale distributed graph processing partake in tradeoffs of algorithm flexibility, scalability or performance by targeting specific aspects of an analysis method's parallel translation. Problems of workload distribution of irregular graph data, the

associated communication overheads and overall algorithm complexity limit the performance of general graph analytics frameworks at scale [143]. To address this problem, this thesis presents work on a 2D edge-based graph framework [20], which offset the typical "vertex-centric" 1D approaches of graph parallelism to improve communication overheads and overall scalability.

1.1.3 Weighted Matching

Matching problems are widespread in usage, such as within admissions and assignment problems, network packet switching, data mining, graph partitioning, computer vision and countless more [9, 13, 53, 111, 120]. The maximum weighted matching problem seeks to find a disjoint subset of the graph's edge set such that the overall weight is maximized. In parallel, optimal methods are infeasible in implementation due to augmenting path dependencies and overall algorithmic complexity, leading to approximate methods for greater efficiency [64]. Relative to this, this thesis explores the parallel implementation and application of approximate weighted matching on GPUs, while experimenting with the integration of these methods within popular set similarity workflows in data analytics.

1.2 Thesis Outline

This thesis is comprised of five content chapters focused on a range of topics within large-scale graph analytics. Chapters 2 and 3 focus on parallel methods for general graph analytics, primarily targeting the vertex ordering and network sensing problems using multithreaded and multi-GPU platforms. Chapters 4 and 5 discuss the parallel translation of the approximate maximum weighted matching problem and its applications to the notion of set similarity. Chapter 6 then follows with a work combining the notions of the previous chapters, including a distributed, multi-GPU graph analytics platform consisting of general graph analysis methods, including weighted matching.

CHAPTER 2
EXPLICIT ORDERING REFINEMENT FOR ACCELERATING IRREGULAR GRAPH ANALYSIS

2.1 Introduction

This chapter discusses an experimental study into a CPU shared-memory refinement-based approach to the vertex ordering problem. We evaluate this degree-based refinement method relative to a number of initial orderings using three shared-memory graph analytic algorithms: PageRank, Louvain and the Multistep algorithm. Applying refinement, we observe runtime improvements of up to 15x on the ClueWeb09 graph and up to 4x improvements to cache efficiency on a variety of network types and initial orderings, demonstrating the feasibility of an optimization approach to the vertex ordering problem at a large scale.

2.2 Vertex Ordering

Graph analysis at a large scale has become increasingly applicable given modern advancements in high-performance computing methods and the sheer size of real-world networks [55, 82]. Despite this, inefficient memory access patterns often limit the performance and feasibility of such analytic algorithms, especially as edge totals surpass the trillions. This problem is generalized to a vertex locality issue, prompting concerns regarding the frequency of vertex access and the order at which this occurs. These concerns are shown to limit the scalability of graph analytic algorithms in parallel [89].

To improve upon vertex locality, ordering methods have been explored to generate hierarchical relationships in large-scale graphs through the consideration of vertex labels relative to cache memory access patterns among certain classes of networks. Layered Label Propagation (LLP) [18], the Shingle ordering heuristic [30] and the Rabbit ordering algorithm [4], among others, all show results towards this goal. In an application setting, these methods demonstrate greater efficiencies within common analysis algorithms such as PageRank and community detection while prompting usage in graph compression and the development of compression-friendly orderings [87, 114]. While showing promising results, such reordering

This chapter has previously appeared as: M. Mandulak, R. Hu, and G. Slota, *Explicit ordering refinement for accelerating irregular graph analysis*, in 2022 IEEE High Performance Extreme Computing Conference (HPEC), IEEE Computer Society, 2022, pp. 1–8.

algorithms are often complex and focus on heuristics or greedy methods for approximation.

In this work, we seek to conduct an experimental study on the application of optimization to the vertex ordering problem for the improvement of CPU shared-memory parallel graph analysis methods. Specifically, we take inspiration from graph partitioning methods and propose a novel optimization method focusing on the explicit refinement of low-degree vertices within a graph. While known ordering methods achieve efficient cache access patterns and analysis runtimes using heuristics, we note that our optimization approach is easily applied to these generated orderings and shows promising results for optimization as a solution to the vertex ordering problem.

2.2.1 Contributions

Our degree-based refinement method builds upon any input initial ordering and improves ordering quality relative to our locality metrics in the Linear Gap Arrangement (LinGap) and Log Gap Arrangement (LogGap) problem towards improved analysis runtimes and cache efficiencies. We focus on three CPU shared-memory parallel graph analysis algorithms: PageRank, Multistep connectivity [128] and Louvain [14]. We apply and compare our refinement method to three heuristic ordering methods in the LLP, Shingle ordering and Rabbit ordering algorithms. Using an AMD system, we demonstrate runtime improvements of up to 15x on PageRank and over 2x on Multistep relative to the ClueWeb09 graph's natural ordering and 10-15x speedups on a number of graphs for the Louvain algorithm. On algorithm-generated orderings, we observe speedups between 1.1-3x and up to 2x improvement to cache efficiency. From this data, we state the main observations of our experimental study as follows:

- The LinGap and LogGap metrics show a strong positive correlation with PageRank analysis measures and slight correlations for the Multistep and Louvain measures.
- Spikes in the improvement of analysis measures occur at singular points within refinement progression, dependent on the graph structure.
- Refinement upon an initial Rabbit ordering shows the most overall improvement of analysis measures within our test data.
- The application of optimization methods to the vertex ordering problem shows promising improvements upon heuristic methods.

2.3 Related Works and Background

2.3.1 Related Works

A number of existing vertex ordering algorithms take intuitive approaches to reordering vertices towards efficient graph analysis in main memory. Presented in [30], the Shingle ordering heuristic focuses on ordering vertices relative to the measured commonality between two vertices' neighborhoods. This ordering scheme shows high compression rates relative to the natural ordering for social networks in particular. The Layered Label Propagation ordering method in [18] applies typical label propagation methods for compression optimization while considering global label states. This algorithm shows noticeable improvements to the compression rates of web graphs compared to the Apostolico and Drovandi algorithm in [2].

Alternatively, the Rabbit Order algorithm proposed in [4] focuses on runtime efficiency by optimizing towards cache efficiency directly rather than compression rates. This algorithm focuses on developing a hierarchy within real-world graphs and connecting it to cache hierarchies for an intuitive sense of cache locality. Communities are then developed and extracted from this hierarchy and a new vertex ordering is generated. This method shows notable speedup in parallel relative to compression-based ordering schemes while improving the runtimes of sparse matrix-vector multiplication (SpMV) analytic algorithms, such as PageRank and Label Propagation [151]. This focus on SpMV extends generally into matrix ordering contexts. Specifically, we see applications to the similar fill-reducing ordering problem, where methods attempt to apply heuristics in the fill-reduction of matrices for graph structures, partitioning and general matrix orderings [25, 74].

2.3.2 Graph Ordering Problem

Defining the ordering problem as in [30], we consider an undirected graph $G = (V[0, n), E \subseteq V \times V)$ and seek a permutation $\pi : V \to N$ such that the chosen metric is minimized. To set our optimization goal, we reference some accepted metrics in graph ordering that, while not robust, provide a means of judging ordering quality based on a notion of vertex label locality.

2.3.3 Metrics

Considering the focus on vertex-centric models later justified in §2.5.1, we use this approach in our choice of metrics for refinement: the Linear Gap Arrangement and Log Gap

Arrangement problems. We choose these metrics due to their demonstrated correlation with the runtimes and cache efficiencies of our parallel graph analytic algorithms. We discuss this in detail in §2.6.1.

First, we reference the Minimum Linear Arrangement (MinLA) problem in [30], which considers the sum of the vertex label differences across G. This is defined formally as $LA(G, \pi) = \sum_{u,v \in E} |\pi(u) - \pi(v)|$. We use the extension of this, the Linear Gap Arrangement (LinGap) problem, which considers the sum of the differences of each vertex's sorted neighborhood, excluding itself. Given a vertex u and its sorted neighborhood $sN(u) = \{v_1, v_2, ..., v_d\}$, this is defined as $LinGap(G, \pi) = \sum_{u \in N} \sum_{v_i \in sN(u)} |v_i - v_{i+1}|$.

Similarly, we reference the Minimum Log Arrangement (MinLogA) problem in [30], which considers the log of the sum of the vertex label differences across G, formally defined as $LA(G, \pi) = \sum_{u,v \in E} log(|\pi(u) - \pi(v)|)$. We again reference the Log Gap Arrangement (LogGap) problem, which considers the log of the sum of the differences of each vertex's sorted neighborhood, excluding itself. Under the same conditions as LinGap, LogGap is defined as $LogGap(G, \pi) = \sum_{u \in N} \sum_{v_i \in sN(u)} log(|v_i - v_{i+1}|)$.

Shown in [58] and furthered in [30], both MinLA and MinLogA are determined to be NP-hard over most types of graphs, warranting the usage of either heuristics or greedy approaches [30, 148] to the problems in application. This is extended to the LinGap and LogGap problems. Graph compression schemes have also been studied for the approximation of ordering quality [42]. Approximation methods have also been explored for the MinLA problem and related metrics, such as the Minimum Containing Interval Graph and the Minimum Storage-Time Product in [27].

2.4 Methods

As our primary method to evaluate explicit order refinement, we propose a degree-based refinement method for the explicit optimization of the vertex labels of low-degree vertices across a graph, given some initial vertex ordering. We consider our approach as a proof-of-concept that helps motivate, via our experimental study, further study into explicit order refinement algorithms. A more in-depth development of highly scalable and efficient methods for optimization is reserved for future work.

2.4.1 Degree-based Refinement

Based on our optimization approach, we employ a degree-based refinement method in parallel to improve an initial ordering towards one of the chosen metrics. Note that the initial ordering can simply be the natural ordering or any algorithm-generated ordering.

2.4.1.1 Metric Computation

For the explicit refinement of a chosen metric, we utilize a global and local calculation of metrics. In the global case, we proceed in parallel through G's vertex set V with each vertex calculating its gap metric among its sorted adjacency list. Note that the gap metric and the sorting of the adjacency list proceeds relative to the input label map, determined as the initial ordering. Since we perform this calculation among vertices in parallel, each vertex reduces its local metric value into a global metric value for the ordering. Local metric calculations focus on the two-hop neighborhood of two input vertices for performance improvements.

Algorithm 2.1 Log Gap Arrangement Refinement by Degree

```
1: function LogGap Degree Refine(G,p)
2:     S = sort(V) ascending by degree
3:     for each vertex u in the first p percent of S in parallel do
4:         for each vertex v in u's adjacency list do
5:             bs = evalLogGapArrLocal(G, u, v)
6:             as = evalLogGapArrLocalSwap(G, u, v)
7:             if as < bs and as < desiredSwapVal_u
8:                 desiredSwap_u = v
9:                 desiredSwapVal_u = as
10:    for each vertex u in the first p percent of S do
11:        bs = evalLogGapArr(G)
12:        swap(G, u, desiredSwap_u)
13:        as = evalLogGapArr(G)
14:        if bs < as
15:            swap(G, u, desiredSwap_u)
```

Considering the Log Gap Arrangement Refinement algorithm in Algorithm 2.1, the algorithm proceeds in parallel as follows: given an undirected graph G defined as previously, we first sort V in ascending order based on each vertices' degree. We then focus on two major operations: the Desired Swap (DS) step and the Swap Completion (SC) step. Note that the translation to a LinGap-based implementation for each step is trivial.

2.4.1.2 Desired Swaps

In the DS step, we have each vertex within an input fraction of G compute its most preferred label swap along each edge among its neighbors, decided by calculated improvement upon the LogGap metric. Thus, each vertex, in parallel, simulates a swap with each of its neighbors and compares the metric value before and after the simulated swap. Each vertex then saves its desired swap and continues onto the SC step.

2.4.1.3 Swap Completion

In the SC step, we sequentially iterate through the list of desired swaps and perform the swap if the metric improvement still holds. We are required to check this condition due to the possible infringement of previous confirmed swaps on future desired swaps. Note that the we perform the metric test locally. This continues up to the set fraction of low-degree vertices in G.

Table 2.1: Basic graph properties for experimentation.

Graph	Class	#Vertices	#Edges	Cite
com-Friendster	Social	66 M	1.8 B	[145]
twitter-2010	Social	41.7 M	1.5 B	[81]
LiveJournal	Social	4.8 M	69 M	[7]
web-ClueWeb09	Web Graph	1.7 B	7.9 B	[22]
enwiki-2013	Web Graph	4.2 M	101.3 M	[18]
web-BerkStan	Web Graph	685 K	7.6 M	[84]
it-2004	Web Graph	41.3M	1.2 B	[19]
ant1km	Mesh	13.5 M	53.8 M	[134]
trianglemesh1	Mesh	1.9 M	1.9 M	[119]
USA-road-d	Road	24 M	58.3 M	[121]

2.5 Experimental Setup

With the goal of comparing our refinement algorithm's performance relative to metric improvement, analysis runtimes and cache efficiency, we measure cache efficiency utilizing the Linux perf tool on runs of three shared-memory parallel graph analytic algorithms: PageRank, Louvain [14] and the Multistep[1] connectivity algorithm [128]. We define cache-efficiency relative to cache miss percentages of L1 and L3 cache in relation to overall cache accesses for each. Towards the calculation of these measures, we proceed to describe our experimental setup.

[1]https://github.com/HPCGraphAnalysis/Connectivity

2.5.1 Memory Access

In terms of memory access patterns in graph analysis, we consider general patterns alongside our employed algorithms: PageRank, Multistep connectivity and Louvain. One of the more general approaches considers the "think like a vertex" (TLAV) framework for patterns of vertex-centric access. Such a method, by nature, improves vertex locality while allowing for scalable means of processing. A comprehensive survey of TLAV frameworks and methods is included in [96]. For our analysis algorithms, we note the SpMV basis of the PageRank algorithm yields cache misses due to poor locality within the compressed sparse row (CSR) format of adjacency storage. Further definition of CSR locality within SpMV is mentioned in [4]. For the Multistep connectivity algorithm, focus lies in the initial traversal-based approach with a secondary propagation phase, utilizing BFS-based patterns of memory access among label propagation schemes. The Louvain algorithm accesses the neighbor information for all vertices in a graph and focuses on graph coarsening, which is relative to vertex locality and ordering dependent.

Within our experimental study, we note that all of our graph analysis algorithms function under a vertex-centric approach with CPU-based shared-memory parallelism. We make this distinction to denote the focus of our study on such parallel models of graph analysis methods despite algorithmic differences.

2.5.2 Data

We present results using a variety of network topologies, specifically social networks, web graphs, meshes and a road network. The properties of each network are included in Table 2.1. Our graphs are collected from well known data sources such as SNAP, DIMACS and WebGraph.

2.5.3 Architecture

We use an AMD system for the collection of cache miss and runtime results. The AMD system has 2TB of DDR4 RAM and dual-socket NUMA AMD EPYC 7742 64-core processors and 2 threads per core with a clock speed of 1500MHz. Each core has a 4MiB L1 instruction cache, 4MiB L1 data cache, a 64MiB L2 cache and 256MiB of shared L3 cache per socket. The AMD system operates on Ubuntu version 20.04.4 LTS.

Table 2.2: Relative metric correlation coefficients for algorithm analysis metrics.

Metric	LinGap	LogGap
PageRank Cache	0.2568	0.3196
PageRank Timing	0.7041	0.8796
Multistep Cache	-0.0005	-0.0006
Multistep Timing	0.0088	0.0090
Louvain Cache	-0.0093	-0.0007
Louvain Timing	0.0387	0.0477

2.5.4 Experimentation

Our degree-based refinement method, alternative ordering algorithms (Layered Label Propagation, Shingle and Rabbit[2]) and our parallel graph analytic algorithms (PageRank, Multistep connectivity[1] and Louvain) are implemented in C++ and compiled using GCC version 9.4.0 and OpenMP version 4.5 for shared-memory parallelism. The Linux perf tool is used for collecting L1 cache and overall cache miss rates with all reported results being the arithmetic average of ten runs per graph analytic algorithm. The perf tool events used are cache-references, cache-misses, L1-dcache-load and L1-dcache-load-miss. We run PageRank on each graph for twenty iterations and we set the max iterations for our Louvain runs at five. Mesh generation for the trianglemesh1 graph occurs on Python3 version 3.8.10 and Pygalmesh version 0.10.6.

2.6 Results

In this section, we present and evaluate experimental results. Our contributions here are two-fold: (1) an experimental study into LinGap and LogGap as metrics for cache efficiency and analysis times; (2) a comparative evaluation of our presented degree-based refinement method to improve ordering quality relative to these metrics. We first discuss results pertaining to the relationship between the amount of degree-based refinement and graph analytic efficiencies. We then perform a comprehensive suite of tests using our refinement method and the ordering algorithms discussed in §2.3.1 and discuss the results.

2.6.1 A Motivating Case for LinGap and LogGap

We begin evaluation by considering the LinGap and LogGap problems as our metrics for refinement. The choice is intuitive considering their quantified notion of vertex locality

[2]https://github.com/araij/rabbit_order

through the measured label gaps within a neighborhood. Similarly, each problem provides an explicit refinement focus for optimization and serves as a general measure for how "different" our refined ordering is from an input initial ordering. Thus, we consider the correlation between the quality of an ordering relative to the measures we are interested in: cache efficiency and analytic runtime. We take the relative measure of ordering quality relative to that graph's worst metric while relative time and cache efficiency follow. Results are collected on the AMD system and are taken across all graphs and orderings. The results for LinGap and LogGap correlation are shown in Table 2.2. The PageRank results, alongside the intuitive notion of the LinGap and LogGap metrics as a vertex locality measure, show promise in the exploration and application, even if the Multistep and Louvain algorithm results suggest a potentially complex relationship.

Table 2.3: Improvement and speedup results for each ordering method taken as the geometric average across all graphs and across all analytic algorithms using the AMD system.

Ordering	Cache	L1 Cache	Time
LLP	0.991	1.002	1.637
LLPLinRefine	1.053	1.005	1.623
LLPLogRefine	1.056	1.007	1.593
Rabbit	1.002	0.999	1.933
RabbitLinRefine	**1.144**	1.017	**2.025**
RabbitLogRefine	1.137	**1.031**	1.973
Shingle	1.017	1.018	1.317
ShingleLinRefine	1.043	0.986	1.336
ShingleLogRefine	1.050	1.026	1.340
LinRefine	1.054	1.007	1.479
LogRefine	1.058	0.992	1.458

2.6.2 Degree-based Approach

Building on the correlation in §2.6.1, we collect results using each degree-based refinement method to observe metric growth progression within refinement. Specifically, we conduct degree-based refinement for each metric using the LiveJournal graph on intervals between 0.1% and 5.0% of the low-degree vertices and analyze the metric improvement for each interval. A similar trend in these results exists for both metrics in improvement with significant improvement starting around the 1% marker, or approximately 48,000 low-degree vertices refined. We use this marker to collect our comprehensive results. From this, we draw one main observation: the largest improvement in analytic measures tend to occur directly

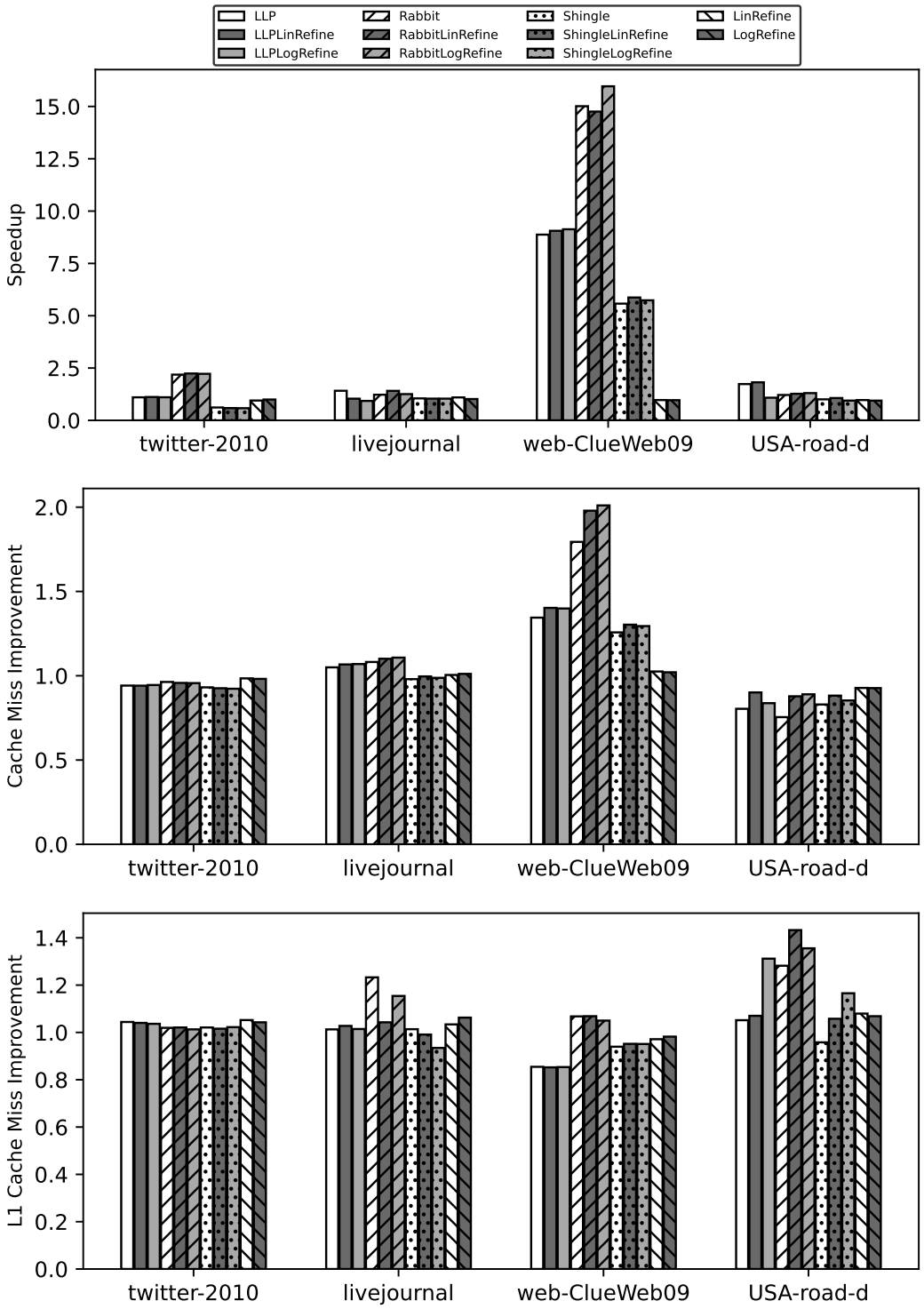

Figure 2.1: Cache miss improvement and algorithm speedup relative to the natural ordering for the PageRank algorithm. Improvement and speedup is taken as a fraction of the performance of the natural ordering over each ordering method's performance as recorded using the AMD system.

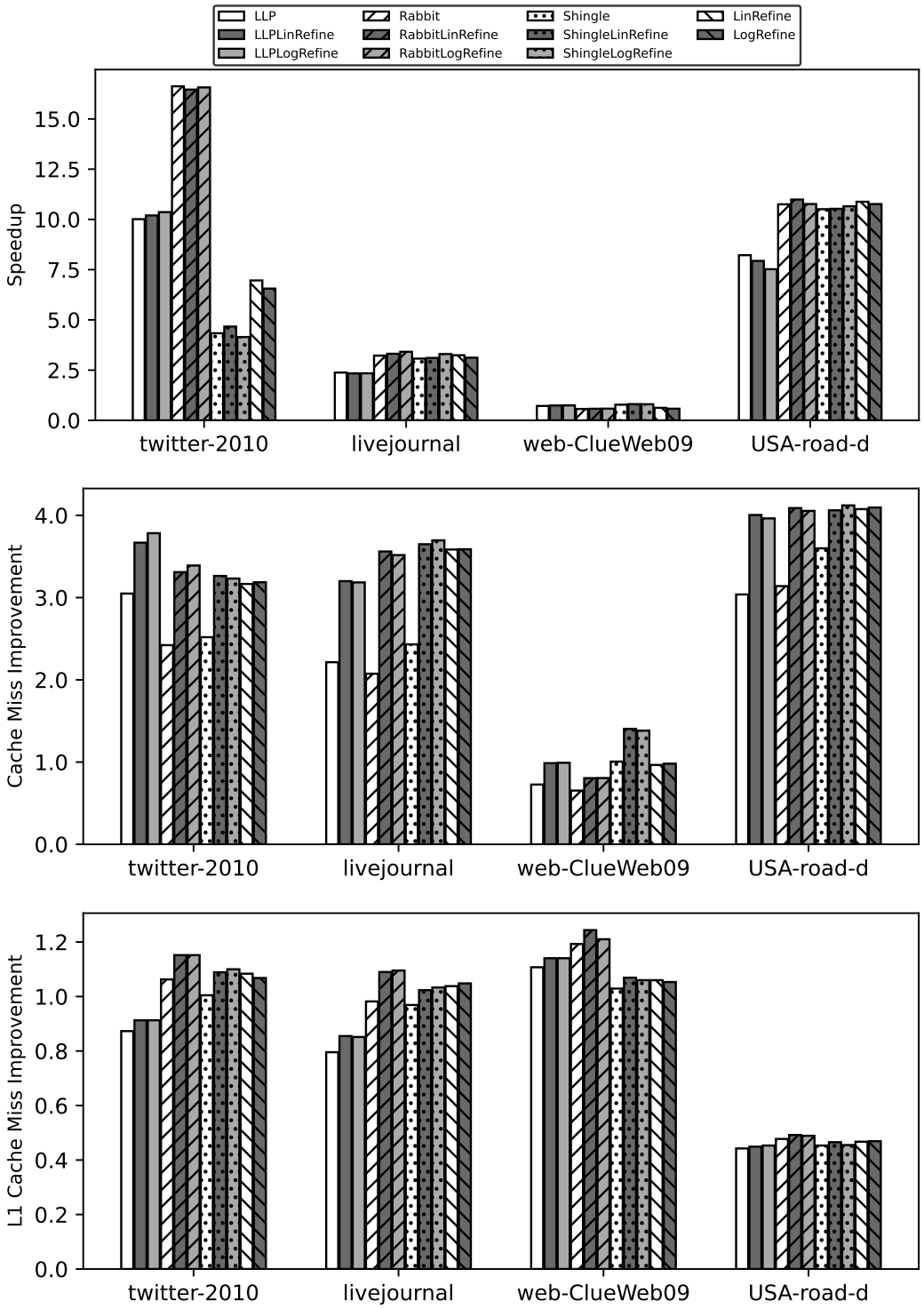

Figure 2.2: Cache miss improvement and algorithm speedup relative to the natural ordering for the Louvain algorithm. Improvement and speedup is taken as a fraction of the performance of the natural ordering over each ordering method's performance as recorded using the AMD system.

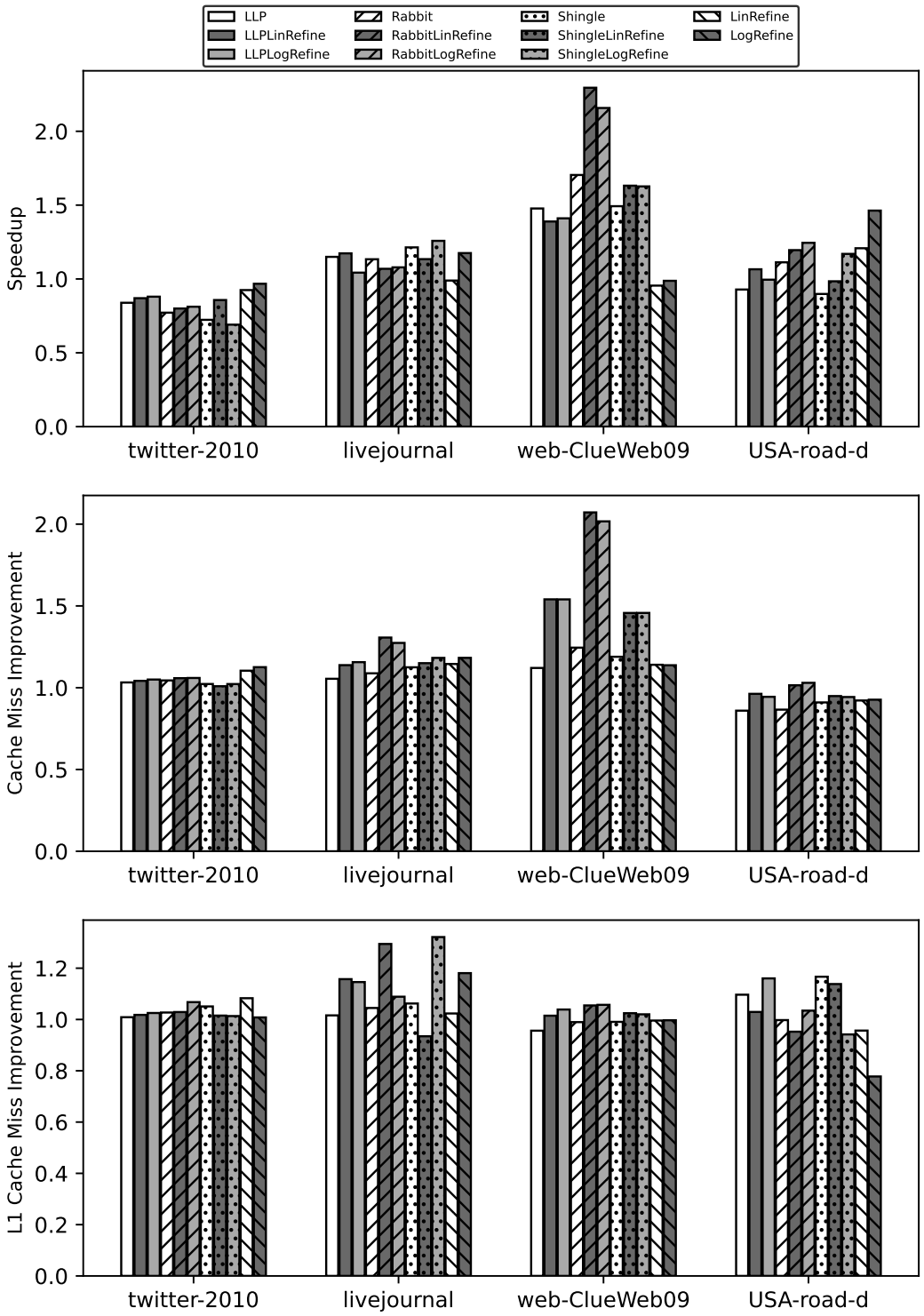

Figure 2.3: Cache miss improvement and algorithm speedup relative to the natural ordering for the Multistep algorithm. Improvement and speedup is taken as a fraction of the performance of the natural ordering over each ordering method's performance as recorded using the AMD system.

at a single threshold percentage, with minimal improvement otherwise. We attribute this to the existence of key vertex-label pairings within an ordering, where the refinement of such pairings yields high analysis metric improvements, suggesting potential in the identification of such pairings before refinement.

2.6.3 Analytic Performance

First, we describe our refinement process for experimentation. We perform degree-based refinement on each graph in Table 2.1 using between 10% and 0.001% of the graph's low-degree vertices, determined based on the threshold marker referenced in §2.6.2. This method is applied to each graph once per initial ordering and once per metric. We collect improvement and speedup results relative to the natural ordering of each ordering method for each of the graph analytic algorithms. The results using the AMD system for the PageRank, Multistep connectivity and Louvain algorithms are presented in Figs. 2.1, 2.2, 2.3. For conciseness, we provide results of improvement taken as the geometric mean across all of our analytic algorithms in Table 2.3. We also omit comprehensive results in Figs. 2.1, 2.2 and 2.3 for conciseness and instead display the most notable results.

Examining the results for the PageRank algorithm in Fig. 2.1, we notice cache miss improvements remaining relatively consistent with that of the natural ordering for all graphs except for ClueWeb09, where a near 2× improvement is reached using LogGap refinement on a Rabbit ordering. Similarly, PageRank timing shows up to a 15× speedup. We note that ClueWeb09 is the largest graph in our study and attribute such an improvement to the limitations of graph access in main memory, resulting in the large impact of improvement upon ordering quality. Similarly, the application of the hierarchical method of a Rabbit ordering alongside LogGap refinement showing significant results for the PageRank algorithm is likely due to the optimization of outliers within the hierarchical communities towards the centralized adjacency matrix under a SpMV-focused analysis. The Multistep connectivity algorithm results in Fig. 2.3 shows similar trends, but with timing and cache improvements closer to 2×. An interesting distinction from the PageRank results occurs in the maximum improvement through the LinGap refined Rabbit ordering instead of the LogGap refinement. This is likely due to the traversal-based nature of the Multistep connectivity algorithm, which focuses on the gaps among neighboring vertices rather than a scaled measure. In terms of refinement efficacy, we again see a fair distribution of maximized cases that include an applied

refinement across the analytic metrics.

The Louvain algorithm's results in Fig. 2.2 show distinct trends compared to those of the previous analytic algorithms, yielding higher levels of cache miss improvement - up to 3–4×. L1 cache misses show significantly higher miss rates for the graphs that demonstrate high overall cache miss improvement. Speedup values achieve around 15× for our Twitter graph while including 10× speedups for a mesh and the road network. We can observe that the focus on refining low-degree vertices under a coarsening model shows high levels of cache miss improvement for graphs of moderate density and can result in analytic speedups.

To obtain a general measure for overall performance per analytic metric, we take the geometric mean of improvement values across all graphs and graph analytic algorithms for each ordering method. The results from runs on the AMD system are compiled in Table 2.3. Highlighted are the highest improvement totals across each of analysis metrics. We observe that applying refinement achieves the highest overall improvement for all cases across the three analytic measures, especially under a Rabbit ordering. Since our analysis algorithms all consider a vertex-centric approach through SpMV, traversal and coarsening, it holds that a community-based approach with explicit refinement within communities would show locality improvements.

2.7 Concluding Remarks

This chapter has explored a new area within vertex ordering that considers the explicit optimization of the Linear Gap Arrangement and Log Gap Arrangement problem metrics. We showed up to 15× performance improvements using refinement relative to heuristic methods. Following from the vertex ordering problem, we explore further extensions of general graph analytics through parallel platforms in the next chapter, where we discuss a multi-GPU implementation of network traffic analysis methods using emerging standard C++26 technologies.

CHAPTER 3
ANONYMIZED NETWORK SENSING USING C++26 STD::EXECUTION ON GPUS

3.1 Introduction

In this chapter, we explore another avenue of large-scale graph analytics through network sensing, primarily targeting GPU platforms and emerging technologies. These GPU-based implementations commonly face start-up challenges in host-device memory management and porting complex workloads on devices, among others. To mitigate these challenges, composable frameworks have emerged using modern C++ programming language, for efficiently deploying analytics tasks on GPUs. Specifically, the recent C++26 `Senders` model of asynchronous data operation chaining provides a simple interface for *bulk pushing tasks* to varied device execution contexts. Applying this, we discuss the practical aspects of developing the Anonymized Network Sensing Graph Challenge on dense-GPU systems using this recently proposed C++26 `Senders` model. Adopting a generic and productive programming model does not necessarily impact the critical-path performance (as compared to low-level proprietary vendor-based programming models). Our commodity library-based implementation achieves up to 55× performance improvements on 8× NVIDIA A100 GPUs as compared to the reference serial GraphBLAS baseline.

3.2 Network Sensing

Network sensing at a large-scale [77] plays a critical role in a variety of applications, such as in network traffic characterization, recommender systems, sensor dynamics and network data analytics [72, 75, 152]. This is further emphasized through the recently proposed Anonymized Network Sensing Graph Challenge [136], highlighting contributions to the generation and analysis pipeline of anonymized traffic matrix data from network packet captures (in the form of a standardized file format, namely, PCAP). Focusing on the analysis portion of this Graph Challenge, GPU-based analytics have become increasingly relevant within large-scale network data workloads [60, 142, 147]. Consequently, dense-GPU systems (i.e., 4–16

This chapter has previously appeared as: M. Mandulak, S. Ghosh, S. M. Ferdous, M. Halappanavar, and G. Slota, *Anonymized network sensing using c++26 std::execution on gpus*, in 2025 IEEE High Performance Extreme Computing Conference (HPEC), IEEE Computer Society, 2025, pp. 1–7.

GPUs on a single node) are commonplace, with terabytes of aggregate memory and proprietary GPU interconnection network to enhance the overall bandwidth. However, designing codes to exploit several GPUs must contend with the challenges associated with effective workload distribution, host-device memory traffic, GPU interconnection performance, and complex algorithms [47], to name a few.

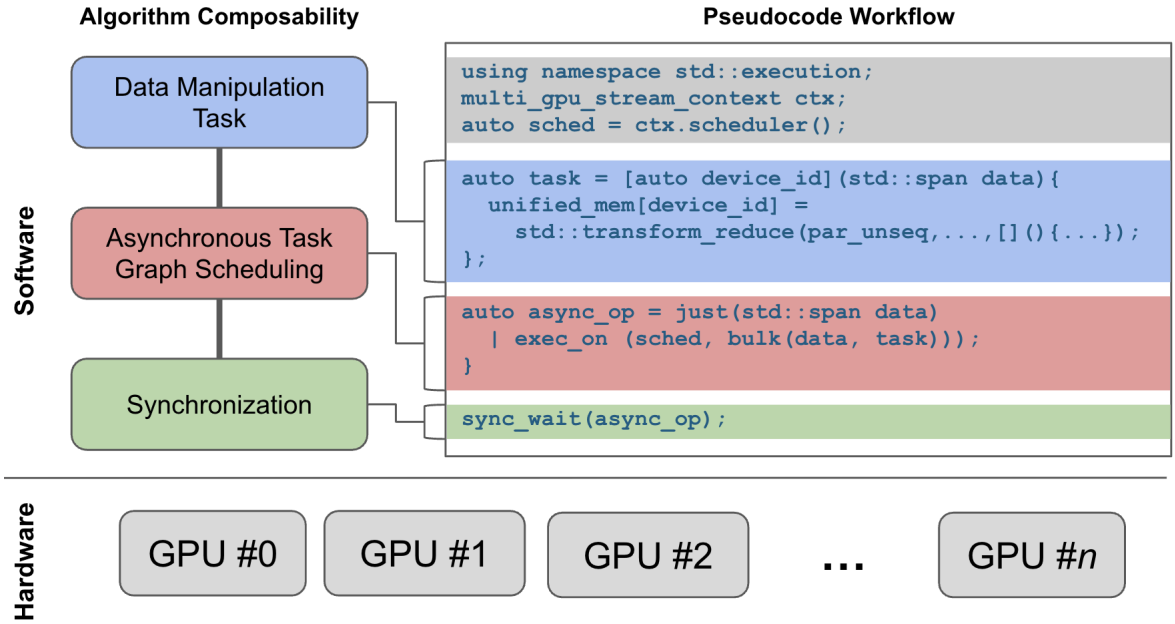

Figure 3.1: C++26 asynchronous `Senders` programming model in `std::execution`. Using this standardized model, asynchronous workloads can be composed over diverse execution environments, for e.g., scheduling data manipulation tasks on multiple GPUs, captured here.

These challenges were traditionally mitigated through vendor-based library solutions coupled with implementations explicitly composing the underlying data structures and associated operations. However, to enhance the portability of workloads targeting contemporary device/accelerator environments, driving the development using vendor-agnostic standardized programming models is key. Towards that goal, the C++ programming language has made significant inroads in improving performance and productivity to span across a wide variety of applications and hardware spectrum. C++26 is the "next" generation of the C++ standard (considered a landmark release since C++11 ushering in the modern C++ era, a decade ago), including significant changes to the library itself relative to the core language features[3]. Specifically, C++26 Execution control library (referred as `std::execution` or

[3]https://en.cppreference.com/w/cpp/compiler_support/26

`Senders` framework) provides composable building blocks for managing asynchronous execution on generic execution environments, as depicted in Fig. 3.1. This library allows transparent composition of task execution graphs or Directed Acyclic Graphs (i.e., DAGs similar to CUDA graphs[4]) and asynchronous scheduling on queues associated with specific execution contexts. It also introduces explicit syntax for launching the tasks (and invoking subsequent callbacks or "receivers"), thus, providing enhanced flexibility for common "many-tasks" scenarios compared to the existing C++ `std::future` or `std::packaged_task` interfaces.

These developments propose explicit high-level asynchronous semantics for GPU-based paradigms, allowing for the translation of "many-tasks" scenarios to many-threaded device contexts through vendor-optimized standardized library functions such as transform, reduce, scan, etc. Following this, we explore the efficacy of the C++26 `Senders` model on GPUs for the large-scale data analytics workload depicted in the Anonymized Network Sensing Graph Challenge.

This work is one of the early works to explore C++26 `std::execution` workflows within large matrix workloads on multiple GPUs. Our contributions are as follows:

- Develop Anonymized Network Sensing Graph Challenge using recently proposed C++26 `Senders` model, leveraging data structures built on top of Standard Library containers for platform-agnostic analytics.
- Incorporated generic data batching scheme to subpartition network data between host and GPU (for handling partition sizes exceeding the available GPU memory).
- Achieved performance improvements of up to 55× compared to the serial GraphBLAS-based reference implementation using 8× NVIDIA A100 GPUs.

3.3 Background

3.3.1 Graph Challenge

The Anonymized Network Sensing Graph Challenge [136] highlights contributions towards improved network traffic analytics through multiple facets. These include the improvement of PCAP I/O, packet data extraction methods, IP anonymization, traffic matrix construction, matrix storage, and matrix analytics. This work focuses on improving the final step of network traffic analytics through the application of dense-GPU processing. At

[4]https://developer.nvidia.com/blog/cuda-graphs/

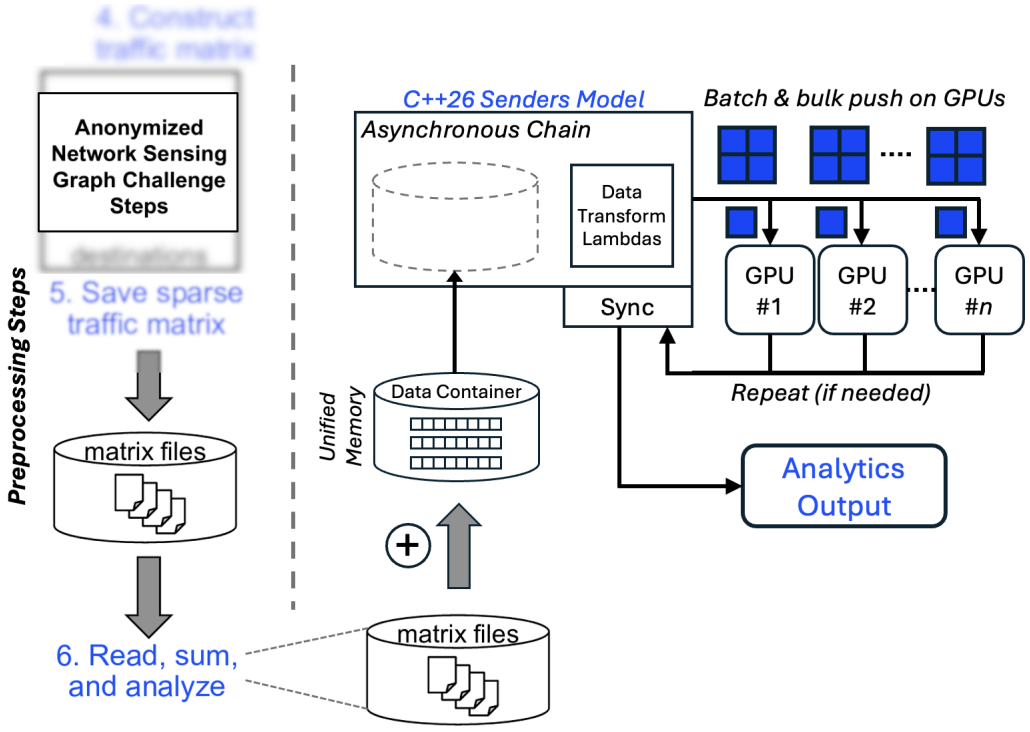

Figure 3.2: Graph Challenge workflow overview with our proposed processing method using the C++26 Senders Model. The "Preprocessing Steps" graphics are adapted from [136]. Under the C++26 Senders Model, we load and aggregate traffic matrix files, forming an asynchronous workflow, comprised of batching and bulk pushing data operations to multiple GPUs.

a high level, we outline our method in Fig. 3.2. We begin with the pre-processing steps adapted from the Graph Challenge proposal, extending the analysis portion to the loading of matrix files into generalized data containers. Using these, we follow propositions in the *C++26 Senders Model* workflow, performing data analytics by pushing asynchronous chains of workloads to a multi-GPU setup and retrieving the analytics output. Next, we describe the components of the *C++26 Senders Model*.

3.3.2 C++26 Execution Model

Within the P2300 proposal for standard C++ [66], the authors propose an asynchronous task management workflow under the `std::execution` namespace for composable task-based parallel workloads launched on generic execution resources (i.e., an abstraction for a hardware component), handled by a scheduler. Overall, this proposal builds upon the popular asynchronous callback-promise workflow under a "senders-receivers" (`Senders`) model, associating an asynchronous unit of computation to a *sender* with a completed state

being processed under a *receiver*. The *execution resource* of this model is expressed via an explicit scheduler abstraction; scheduler represents the place where execution happens (with options to compose senders into task graphs), which could be a thread-pool, single thread or GPU(s).

Vendor-specific implementations of `std::execution` are in active development to extend the `Senders` model to accelerators, such as GPUs. Specifically, NVIDIA is enhancing integration of proprietary CUDA with the C++ Standard Library (STL) functionalities on device through `libcudacxx` [85], backporting several libraries relative to data structures and operations under the `cuda::std` namespace. The integration of STL data structures and basic operations (such as `span, mdpsan, reduce,` etc.) is extended within the `Senders` model through `nvexec` [105], NVIDIA's device-based schedulers and functionalities for the relevant asynchronous abstractions. There are some generalizations, such as a default allocation of all used device memory to unified memory under `nvexec`. Overall, the `stdexec` library is a reference implementation of the C++26 Senders model, whereas NVIDIA C++17 Standard Parallelism (`stdpar`) allows GPU acceleration of the standard algorithms, both are integrated within the NVIDIA HPC SDK suite.

3.4 Methodology

We introduce the key components in implementing network traffic analytics challenge using C++26 *senders*-based programming model for containerized data processing, in §3.4.1 and §3.4.2. In §3.4.3, we discuss subpartitioning the input data for concurrent batch processing on GPUs.

3.4.1 Programming and Execution Model

From the perspective of our workload, high-level components of the C++26 asynchronous senders model are captured in Fig. 3.1. We first distinguish between the hardware and software layers of this model, followed by task composition.

Software Model: We first provide an overview of the pseudocode workflow for a generic transformation and reduction operation from Fig. 3.1, making generalizations for conciseness. The model relies on the usage of a vendor-specific device scheduler built on top of a single or multi-device context (depicted as `sched` in Fig. 3.1). Using this abstraction, programming on single vs. several devices (i.e., execution resources) are almost indistin-

guishable; we rely on a multi-GPU context to bulk push operations on a single dense-GPU node.

We consider the data manipulation tasks launched on devices as *lambda* functions (lambda expressions or lambdas are unnamed functions used to define operations where they are invoked), with `std::span` to represent a non-owning view of the contiguous data for processing. This allows for application extensibility, as we can process data of arbitrary contiguous sequences (i.e., trivially copyable) through `span`. Since vendor extensions such as `nvexec` (which provides NVIDIA specific schedulers and runtime support) assumes unified memory (i.e., single shared address space between host and device), we perform operations (in this case, `transform_reduce`) within the `std` namespace and commit results to unified memory relative to the device index (see Fig. 3.1).

The asynchronous data manipulation tasks are chained and associated with specific data spans. While supporting complex task workflows under `std::execution` for asynchronous chaining, for our purposes we can simply push bulk executions to multiple devices (reusing `sched`) and the set data manipulation lambda accordingly. We then perform a synchronization on this asynchronous chain, retrieving the result at the end of the workflow. This can be further customized with collective operations or other synchronization measures (e.g., multi-node scenarios).

Execution Environment: Traditional device-based applications rely on vendor-specific programming models, with separate multi-node or multi-GPU abstractions. In this model, various options exist to instantiate the underlying execution resource (i.e., `sched` in Fig. 3.1), comprising of multiple CPU/host threads, single GPU or dense-GPUs (all GPUs in a node), while keeping the asynchronous task descriptions uniform. With appropriate back-end support, this model can be expanded to distributed implementations (over network) considering partitioned data.

Tasks Composability: The basis of the programming model relies on the generalization of any simple or complex parallel computation into three basic components: a data manipulation task, asynchronous data operations and bulk synchronizations. In Fig. 3.1, we provide an example of a generic transform and reduction operation (which transforms each element in the container in place and then accumulates the results), a common data manipulation task in a myriad of data analysis workflows. For our purposes, we demonstrate extensibility to common matrix-based network analysis measures, such as link counts, source

counts and max degrees, among others. For more complex algorithms (i.e., which cannot be recast into a combination of containerized scan, reduce and transform operations), GPU integration with `stdpar` allows inclusion of device kernel within the data manipulation lambdas, allowing greater flexibility.

3.4.2 Standardized Containers

The usage of `std::span` within the programming model allows expansive data backends for portable analytics. Popular frameworks, such as cuGraph [35] or Gunrock [141], considers variety of input data formats (dedicated data ingestion frameworks such as cuDF [34] can alleviate preprocessing overheads), but often resort to containerized operations internally for implementing workloads. This method espouses the composable and generic aspects of C++ metaprogramming. Using this programming model, we adhere to similar principles, using `std::vector` container for the relevant data structures and passing an `std::span` for device invocations. Complex sparse representations, such as Compressed Sparse Row (CSR), can be managed through a collection of `std::vector` containers (e.g. consider a graph as a vector of a vector of vertex neighborhoods; operations such as triangle counting become a sequence of set intersections).

3.4.3 Concurrent Batching

Regarding data distribution across devices using multi-device schedulers, the usage of unified memory simplifies data access and allocation using on-demand page migration [115]. This, however, assumes a simple device-data partitioning occurs to split data evenly among devices. For our purposes, we rely on this even split of matrix data per device, which allows us to further take advantage of data *batching* on device.

We consider *batching* as a technique to subpartition input data into b_n batches, where each batch is processed in parallel one after the other. We detail this workflow in Fig. 3.3. The batch count b_n is input specified, where we assume $b_n = 1$ is the default case (where the batch is the entire device data partition), and $b_n = k$ splits the input data into k portions per device. This allows us to leverage sequential processing for better scalability and lower page faults through reduced data sizes at processing time. We further discuss the impact on performance in §3.6.

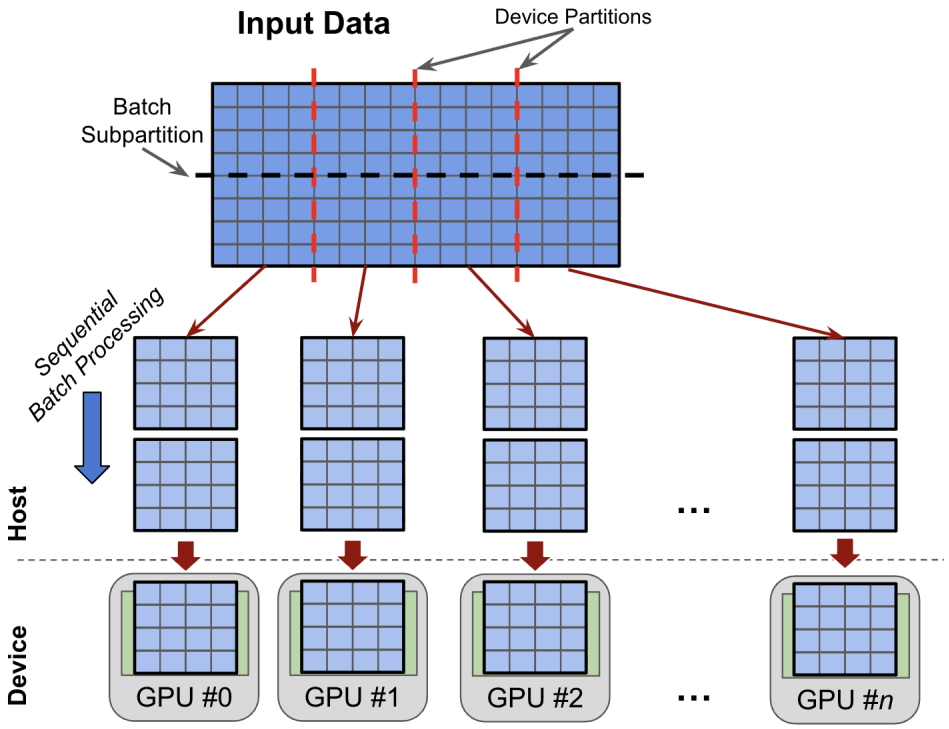

Figure 3.3: Demonstrates input data batching from host to GPUs, where individual device partitions are sub- partitioned into fixed-size batches, sequentially moved on GPUs by the host during computation.

3.5 Implementation Details

In this section, we discuss our implementation alongside the design choices for Graph Challenge analytics, listed in Table 3.1.

3.5.1 Software Dependencies

We utilize a number of vendor-based libraries that integrate C++26 `Senders` model execution workflows on GPUs, specifically NVIDIA CUDA-based implementations of `std::execution` workflows through `nvexec` [105] for the translation of asynchronous workflows and operation lambdas to device code. `nvexec` also provides the multi-GPU scheduler context used to designate the execution resource. Our containerized data structures are STL-based and uses `libcudacxx` or `libcu++` [85] for automatic unified memory management. We further use `libcu++` for device implementations of the STL operations, such as `std::reduce` available in `cuda::std` namespace.

Table 3.1: Graph Challenge packet analysis measures, with the aggregate properties and summation notations adapted from [136]. The relevant programming model data operation is listed for each property. A_t represents a network traffic matrix at time t, with $A_t(i,j)$ as the number of packets between source i and destination j.

Aggregate Property	Notation	C++ function
Valid packets N_V	$\sum_i \sum_j A_t(i,j)$	`reduce`(*weights*)
Unique links	$\sum_i \sum_j \lvert A_t(i,j)\rvert_0$	`size`(*edges*)
Unique sources	$\sum_i \left\lvert \sum_j A_t(i,j)\right\rvert_0$	`size`(*row_sums*)
Max source fan-out	$\max_i \left\lvert \sum_j A_t(i,j)\right\rvert_0$	`max`(*degrees*)
Unique destinations	$\sum_j \lvert \sum_i A_t(i,j)\rvert_0$	`size`(*col_sums*)
Max destination fan-in	$\max_j \lvert \sum_i A_t(i,j)\rvert_0$	`max`(*degrees*)

Algorithm 3.1 Pseudocode for *max reduction*

Input: Container: *data*, Multi-GPU Context: *ctx*, Batch Count: $b_n : n \in \mathbb{R}_{>0}$
Output: Maximum value

```
1: procedure MAX_LAMBDA(cuda::std::span(data), cuda::std::span(result)
2:     d ← device_id
3:     result[d] ← max(result[d],
4:         cuda::std::reduce(data, std::max))
5: sched ← ctx.GET_SCHEDULER()
6: cuda::std::span result ← ∅
7: for each batch b of data do
8:     subspan ← cuda::std::span(b)
9:     sndr ← stdexec::just(subspan, result)
10:    | exec_on(sched, stdexec::bulk(size(b),
11:                MAX_LAMBDA)
12:    stdexec::sync_wait(std::move(sndr))
13: return result
```

3.5.2 Data Representation

In the processing of the network traffic matrix files, we refer to the sequential GraphBLAS-based implementation discussed in [136]. Instead of GraphBLAS data representations, we use generic STL-based containers, requiring intermediate transformation of the CSR into flat containers comprising of the *edges*, *degrees* and *weight* data for the respective nonzero entries. Using these source containers, we can build derivative containers for specific data operations, such as *row_sums* and *col_sums* for out– and in–degrees, respectively.

3.5.3 Analytics and Operations

The associated analytics include source, destination and total packet counts, maximum fan-in/fan-out and maximum link counts. For all the required measures, a combination of sum reductions and maximum scans suffice, of which we generalize for repeated data processing among the containers. The Graph Challenge properties are listed in Table 3.1, alongside data manipulation operations used to implement the specific functionality. Each operation can be successively invoked on subsequent batches of data, analyzing sub-partitioned portions of the packet data on device (this option is discussed in §3.4.3). Details of the two primary operations (i.e., scan and reduce) are as follows:

Maximum Scan: A concise example of the maximum reduction implementation is listed in Algorithm 3.1, represented as the lambda expression MAX_LAMBDA, which performs maximum reduction on device using `cuda::std::reduce` on a unified memory-backed `span`. The data can be optionally processed in more than one batches, designating an even number of sub-partitions of the data relative to the number of batches in *subspan*. The asynchronous tasks are encapsulated through the *sender sndr*, passing the input span and result. Next, we specify the resource context (passing the scheduler and lambda) on which the *sender* will be executed (multiple senders can be successive chained/scheduled, i.e., utilizing overloaded `operator|`). Finally, a blocking synchronization on sender completes the operation.

Sum Reduction: For similar reduction or single buffer type operations, we follow the exact same structure as in Pseudocode 3.1. We can simply replace the data manipulation operation in Lines 3 and 4 with the relevant operation, and perform the same workflow.

3.6 Evaluations

We first describe our software and hardware setup, followed by execution time performance results and an overall analysis. We compare our implementation to the sequential GraphBLAS-based Python implementation provided as part of the Graph Challenge [136].

Platform: Our primary GPU platform in our evaluations is an NVIDIA™ DGX system with 8 GPUs (DGX-A100). The DGX A100 is the third generation server node from NVIDIA™, and consists of 8 "Ampere" A100 GPUs (with 108 SMs) with 40GB HBM2 memory/GPU and two-way 64-core AMD EPYC 7742 CPUs at 2.25GHz, 256MB L3 cache, 8 memory channels, and 1TB DDR4 memory. NVIDIA GPUs either come in the PCIe or proprietary NVLink/NVSwitch based form factors. Proprietary SXM module allows NVIDIA

GPUs to directly communicate through NVLink interconnect (DGX-A100 uses SXM4).

Our implementation is built using CUDA version 12.2, GCC version 13.3.0 and version 25.5-0 of the NVIDIA High Performance Computing Software Development Toolkit[5] for `nvc++`. We use `std=c++20`, as C++26 features are experimental using the `--experimental-stdpar` flag. Experimental NVIDIA device support features are available using the `-stdpar=gpu` flag. Our source code is openly available from GitHub: https://github.com/mmandulak1/stdexecANSGC.

Dataset: We use the specific data required for this Graph Challenge [136], which are GraphBLAS matrices derived from randomized network packet data of 2^{30} synthetic packets.

3.6.1 Baseline Execution

We separate baseline execution time assessments into two tasks: *analysis time* and *end-to-end time*. Analysis time is the time taken to calculate the analysis measures, excluding preprocessing (e.g., data structure construction and file I/O). End-to-end time is the entire program execution time for the analysis. For each case, we collect the results of our implementation, scaling up from 1–8 GPUs, and compare to the sequential Graph Challenge baseline. The least of 5 test runs is reported in the results. We present the analysis and end-to-end results in Figures 3.4, 3.5 and 3.6, respectively, including batching variation in batch counts of 1, 5 and 10 (see §3.4.3). We summarize our results as follows, followed by a discussion on the impact of concurrent batching.

Analysis Time: In Figures 3.4 and 3.5, we present execution time scaling per GPU count and overall performance improvement relative to the sequential GraphBLAS-based implementation. We observe the lowest execution time using a batch count of 10 and 8 GPUs of 1.17 seconds, compared to the sequential baseline of 64.23 seconds. At its peak, we observe a 55× performance improvement upon the sequential in analysis timing, with a geometric mean performance improvement at 8 GPUs of 47× across all three batch counts. We attribute performance improvements to improved data distribution using higher device counts, leveraging parallel performance.

Between batch counts, we observe the highest improvement using a batch count of 10, which noticeably outperforms the other batch counts across all GPU counts. On average, the larger batch count yields up to 170% improvement in performance upon a batch count

[5]https://developer.nvidia.com/hpc-sdk

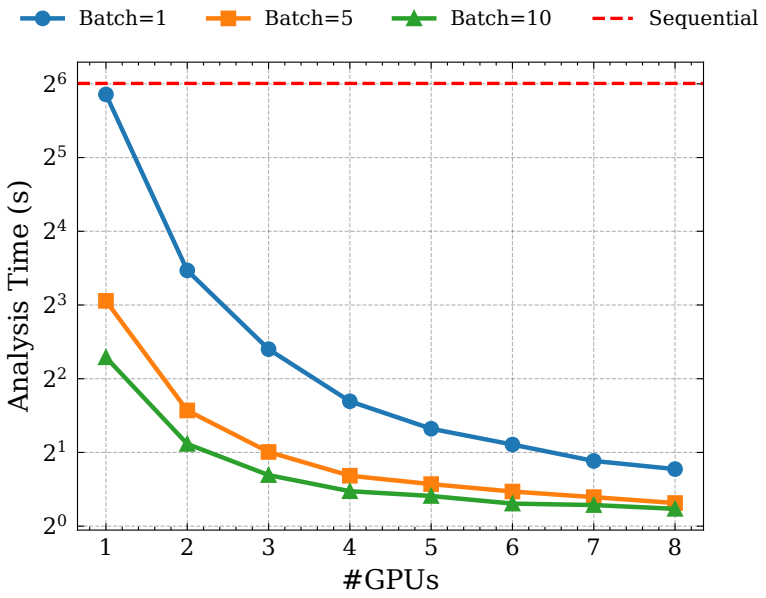

Figure 3.4: Scalability (analysis time, lower is better) on 1–8 GPUs with varying batch counts. Best performance is observed using 8 GPUs and 10 batches at ∼1 seconds compared to ∼64 seconds for sequential baseline.

of 1 and up to 20% at a batch count of 5. On average, using batching in this regard yields a 140%improvement over the default case of a batch count of 1. In the single GPU case, the limitations in flexibility of even data distribution on device are very apparent in performance impacts. While the usage of higher device counts alleviates workloads for better distribution, the combination of batching and higher device counts yields the best flexibility for performant data distribution with lower data sizes at a given processing point.

End-to-End Time: We present the end-to-end performance results in Fig. 3.6, depicting the execution times with varied batch and GPU counts. Aside from the analytics time, we note that the data loading task is relatively expensive, taking approximately 40 seconds. Consequently, host to device data movement costs are nontrivial (about 100 seconds on a single GPU considering a single batch). Still, we demonstrate improvements relative to the sequential baseline by ∼4×, with the highest improvement using a single batch at 8 GPUs by about 9×. By geometric mean, we observe 8× execution time performance improvement across all data points. We further compare implicitly with the 2024 Graph Challenge champion [149], with approximately 2× improvement in end-to-end execution time relative to their multithreaded implementation.

As shown in Fig. 3.6, for #GPU >3, a single batch outperforms that of 5 and 10 batch counts, while a batch count of 10 shows the best performance for #GPUs $\leq$ 3. This can

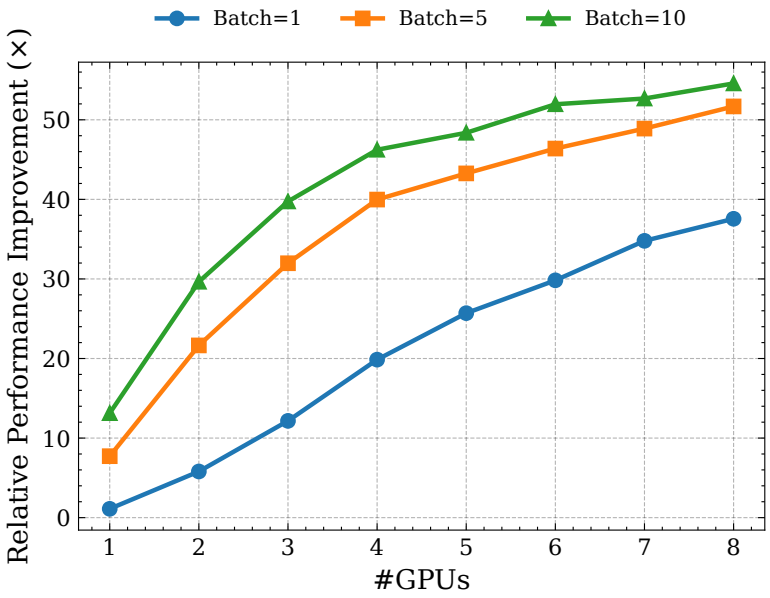

Figure 3.5: Relative performance improvement (compared to serial reference implementation, higher is better) on 1–8 GPUs with varying batch counts. Best performance observed at 8 GPUs using 10 batches: 55×.

be attributed to the relatively expensive data movement and initial container building costs combined with uneven workload distributions owing to the input data. Similarly, we see minimal improvement using 10 batches over 5, with a 1.2× improvement and an average improvement with a batch count >1 of 1.1×.

Batching: Relative to performance, batching serves as a means to balance workload distribution as GPU counts increase. In practice with C++26 `std::execution` device workload scheduling, resources appear to be scheduled efficiently relative to data sizes. Using the `nvexec` scheduler, if a workload fits in the capacity of a single GPU, resources are scheduled accordingly. This does not necessarily balance towards workloads, which is remedied through batching, providing a pre-scheduler workload distribution before bulk pushes to devices. This, alongside the sequential processing loop coupled with batching, results in variable performance improvements relative to tradeoffs in device counts and workload distributions. For our purposes, we express the usefulness of batching in large data instances with low device counts within the non-complex analytics workloads presented.

3.6.2 Packet Processing

We further summarize our end-to-end execution time performance relative to the sequential in terms of network packets processed per second. This metric is calculated by

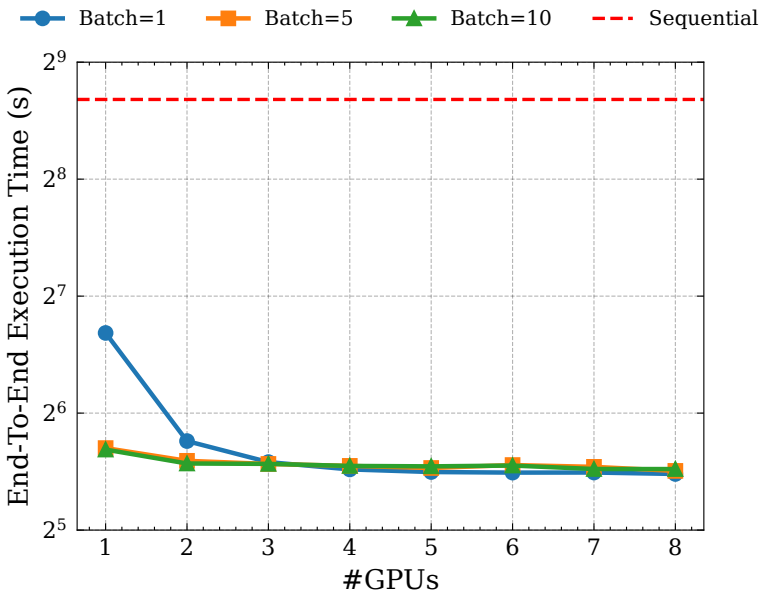

Figure 3.6: Scalability (end-to-end time, lower is better) on 1–8 GPUs with varying batch counts. Best performance observed using 8 GPUs and default batch count of 1, leading to 9× improvement vs. sequential.

Table 3.2: Best packet rate (*higher* is *better*) per batch and #GPUs.

Batch Count	Best Packet Rate (packets/s)	#GPUs
Sequential	2,614,183	-
1	**24,061,441**	8
5	23,626,502	8
10	23,372,598	8

recording the best end-to-end execution time relative to the number of packets in the randomized network traffic dataset. We highlight the best rate per batch count in Table 3.2, observing approx. 9× improvement as compared to the reference baseline, with analysis times following at approximately 50× over the same.

3.6.3 Observations

We discuss the observed productivity and challenges of C++26 `std::execution` model for multi-device execution.

Productivity: Primary benefit of the C++26 execution workflows is flexible device oriented programming through standardized solutions enhanced with vendor optimizations (e.g., device memory management via standardized containers and unified memory). Our implementation is approximately 20 lines of code (LoC) for each of the maximum scan and sum reduction operations, and in total about 100 LoC including the driver functions. We

further note that our implementation includes no explicit CUDA code and relies fully on vendor functionalities supplied within the standard. This is beneficial for free performance gains across GPU generations, without the need to adapt entire workflows to be device-aware, as long as the methods conform to a set of standard operations. Complex operations can be included through explicit device code within the lambdas.

Challenges: Notable challenges exist in the conversion of complex algorithms to fit within the C++26 `std::execution` workflows, especially in the domain of graph problems. The straightforward bulk pushing of operations to devices limits user flexibility in controlling data distribution (which led us to consider batching), which is critical for irregular and hierarchical data instances. Furthermore, de facto unified memory requires careful examination of device data accesses for cache efficiency. These considerations, alongside those typical in parallel graph problems (e.g., load imbalance), still require interventions, as the scheduler does not act as an all-encompassing distributor across expansive problem domains.

3.7 Concluding Remarks

This chapter presented the results of utilizing the recently proposed C++26 `std::execution` model on dense-GPU platforms, splitting network traffic analytics workloads into composable set of data manipulation operations as asynchronous task graphs. In addition, we showed that explicit control of the data distributions across GPUs (i.e., batching) is still relevant for modern programming abstractions. Despite using high-level abstractions, we achieve up to 55× execution time improvement relative to the sequential linear-algebra baseline and about 2× improvement in implicit comparison of end-to-end execution time to the streaming-based multithreaded implementation by the 2024 Graph Challenge champion. In the coming chapters, we explore a separate but classical graph analysis problem in weighted matching.

CHAPTER 4
EFFICIENT WEIGHTED GRAPH MATCHING ON GPUS

4.1 Introduction

In this chapter, we discuss the weighted matching problem, which identifies a maximal subset of edges in a graph such that these edges do not share any vertices in common with each other. This problem faces notable challenges in GPU parallel settings due to common complexities in general graph processing, such as irregular memory access patterns and load imbalances. Furthermore, increasingly massive graph sizes and resultant intermediate data commonly exceeds available GPU memory. To tackle these challenges, this section discusses work in efficient approximation algorithms for *locally dominant matching*, exhibiting 2-45× performance improvements compared to state-of-the-art single-GPU and multithreaded CPU matching implementations on a variety of real-world and synthetic graphs.

4.2 Weighted Matching

Given a graph $G(V, E, w)$, where $w : E \rightarrow \mathbf{R}_{>\mathbf{0}}$ is a weight function with a positive real number associated with each edge. A *weighted matching* $M \subseteq E$ is a set of edges such that no two edges in M are incident on the same vertex, and the sum of weights of matched edges, $\sum_{e \in M} w(e)$, is the maximum among all possible matchings in G. Matching is a fundamental graph problem with numerous applications in diverse fields. Also known as the linear assignment problem, matching has applications in assigning or mapping one set of entities (e.g., residents) to another (e.g., hospitals) [95], numerical linear algebra [36, 46], computer vision and pattern recognition [9], and a variety of scheduling, resource allocation and facility location problems [1, 21].

Optimal algorithms for matching exploit the approach of *augmentation*, where paths that alternate between matched and unmatched edges are iteratively found from current solutions. By swapping the matched edges along these paths, more edges can be matched [80]. However, such an iterative approach limits the amount of work that can be done in parallel. In contrast, approximation algorithms that do not require computing long augmenting

This chapter has previously appeared as: M. Mandulak, S. Ghosh, S. M. Ferdous, M. Halappanvar, and G. Slota, *Efficient weighted graph matching on gpus*, in International Conference for High Performance Computing, Networking, Storage and Analysis (SC'24), IEEE Computer Society, 2024, pp. 1–16.

paths are amenable to parallelization and therefore perform significantly better on parallel systems [64]. An approximation algorithm is required to generate a solution with a provable bound to the optimal one (detailed in §4.3).

With the steady rise in graph sizes and ubiquity of dense-GPU nodes on modern HPC platforms, it has become crucial to develop efficient computational methods and identify trade-offs for graph processing on multiple GPUs. For graph workloads, sustainable (strong) scalability is impacted by severe and unbalanced data movement bottlenecks, brought on due to inherent irregularity in the real-world graph structure and limited computation within many graph algorithms. Linear algebraic methods continue to demonstrate the significant performance advantages of GPUs over multicore CPUs. Although graph algorithms can be algebraically expressed [76], and past research proposed efficient linear algebra based parallel algorithm for finding a perfect matching in a weighted bipartite graph [6], implementing weighted matching on general graphs using sparse linear algebra methods can be prohibitive in terms of the computation costs (currently, no known methods exist). By contrast, efficient approximation algorithms for weighted matching are known [110].

The scalability issues of the graph workloads can be alleviated, to a certain extent, by limiting the data movement within a compute node. This is achieved through leveraging faster GPU interconnects and vendor-optimized collective operations. Nodes with several GPUs and relatively large main memories are becoming mainstream, allowing for processing massive graph workloads, which would have previously required distributed-memory systems and the concerns associated with network communication and load imbalance. Current support exists for up to 72 latest NVIDIA™ Blackwell™ GPUs interconnected within a rack using NVLink™ [106], which translates to an order-of-magnitude increase in the GPU-GPU bandwidth relative to contemporary RDMA/Infiniband interconnects.

However, memory requirements of graph workloads can still easily surpass the available global memory within a GPU (tens of GBs). Even with multiple GPUs, an arbitrary partitioning of a graph can push the workload to its memory limit, leading to out-of-memory and silent errors. In distributed-memory, this problem can be sidestepped by using more resources at startup or by considering fixed-size buffers—both strategies increase communication overheads. In a single node context, this mismatch of the available data and GPU global memory is mitigated by considering a local partitioning of the graph, where a "partition" roughly corresponds to the maximum #edges that can be stored on each device. Further

enhancing the notion of this partitioning is the use of logical "batches" associated with the per-device partitions, with synchronization at the end of every processed batch. The intuition behind batching is about selecting a working set (vertex and corresponding edge ranges) and synchronization interval, to mitigate load imbalances within partitions. Although the device synchronization and batch transfer overheads can be expensive for certain graphs, they can be offset by improved data buffering, thread parallelism, memory access locality, thread occupancy, faster data reductions, and ultimately, multi-GPU parallelism [113, 122]. Even though graph processing workloads are known for irregular data movement overheads leading to implementation and scalability challenges [124], by processing in batches, graph algorithm are able to regularize the synchronization requirements and thus exploit vendor-optimized GPU collective libraries such as NCCL™[70] for inter-GPU communication.

To the best of our knowledge, this work is the first-of-its-kind multi-GPU implementation of weighted approximate matching. Primary contributions are summarized.

- To accommodate large graph partitions on device and control the working set size to enhance scalability, we propose a flexible batch processing scheme in the context of weighted matching on multi-GPU systems.
- We demonstrate 2–45× performance improvement over optimized OpenMP-based CPU graph matching implementation over multiple GPUs.
- We detail performance and quality analysis using several billion-edge real-world and synthetic graphs on two GPU platforms (comprising of NVIDIA™ A100 and V100 GPUs). For small graphs in which the optimal matching could be performed, we show close to the optimal quality (∼6% lower in weight on geometric mean).

We believe that this work will advance both the development of new matching algorithms and matching-based applications to accelerate a large number of domain problems.

4.3 Background and Related Work

4.3.1 Preliminaries

Notations Let $G(V, E, w)$ be a simple undirected graph, where V and E are the set of vertices and edges, respectively, and $w : E \rightarrow \mathbf{R}_{>0}$ is a positive weight function defined on the edges. We define $n = |V|$ and $m = |E|$. A subset $F \subseteq V$ induces a subgraph of G with

F as vertex set and edge set $\{\{u,v\} \in E : u \in F \text{ and } v \in F\}$. Similarly, a subset $F \subseteq E$ induces a subgraph where the vertices are the endpoints of F along with edgeset F. For a vertex v, $\mathcal{N}(v)$ may represent the edges incident on v ($\{e \in E : v \cap e \notin \emptyset\}$) or vertices adjacent to v ($\{u \in V : \{u,v\} \in E\}$), and which definition is used will be clear from the context. For two integers x, y, where $x \leq y$, $[x,y]$ represents the consecutive integers from x to y including themselves. We denote $f(X) = \sum_{e \in X} f(e)$, where f is a function defined on the set X.

Maximum Weight Matching (`MWM`) Problem Given a graph $G(V,E,w)$, a *matching* is a subset of edges, $M \subseteq E$, where every vertex of G has *at most* one endpoint in M. A maximum weight matching (`MWM`) is a matching M^* of maximum $w(M^*)$ among all matching. A matching M is *maximal* if it can not be extended without violating the matching constraint. For an $\alpha \in (0,1]$, M is an α-approximate `MWM` if $w(M) \geq \alpha\ w(M^*)$.

4.3.2 Locally Dominant Algorithm

Definition 4.3.1 (Locally dominant matching)**.** Given a matching M, an edge $e \in E \setminus M$ is available if it does not share any endpoints with any other edge of M, i.e., $M \cap e = \emptyset$. e is locally dominant w.r.t M if $w(e)$ is greater or equal to all available adjacent edges of e. A matching M is locally dominant if every edge of M is locally dominant when it is added to M. In Fig. 4.1, the edges (1,0) and (3,4) locally dominating while (2,3) and (5,4) are not.

We restate the approximation result of a locally dominant algorithm by Preis [111].

Lemma 4.3.1 ([111])**.** *Any algorithm that produces a maximal locally dominant matching is $\frac{1}{2}$-approximate for maximum weight matching.*

The locally dominant algorithm provides us with a framework to design highly concurrent algorithms for matching, since it avoids global sorting as needed for the traditional greedy algorithm. The LocalMax [12] and Suitor [94] algorithms described in the literature are two examples of locally dominant frameworks. We next discuss a pointer-based locally dominant algorithm in Algorithm 4.1, which will provide a base for our multi-GPU algorithms described in the subsequent section. Each iteration of Algorithm 4.1 consists of two phases: a *pointing* and a *matching* phase. In a pointing phase for each vertex, v of G, we identify and point to a neighboring vertex (mate) with the highest weight. In the next phase,

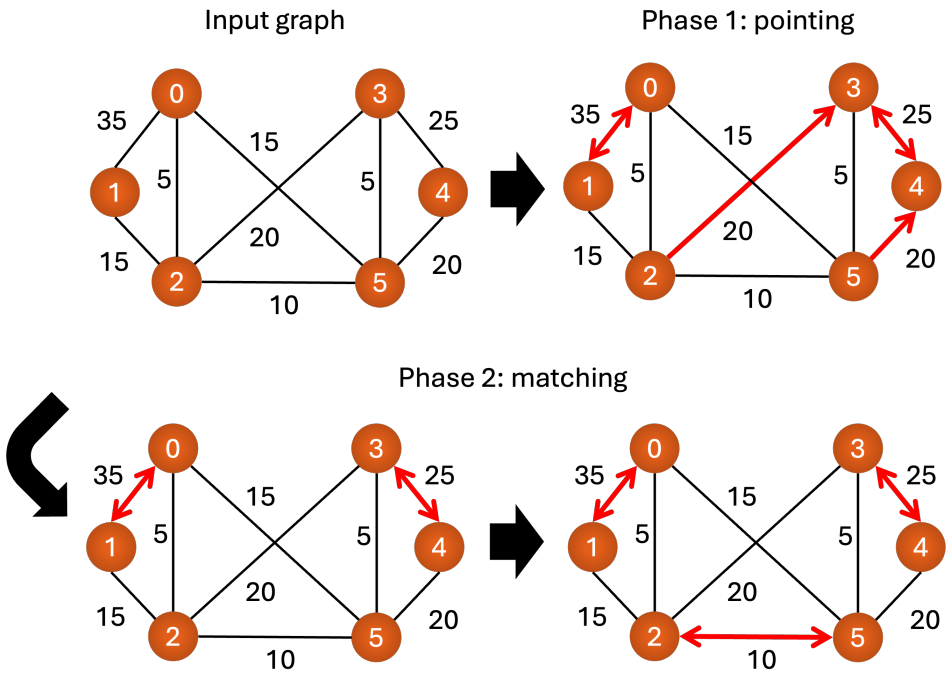

Figure 4.1: One iteration of the LD-SEQ algorithm: *pointing*: for each vertex, choose the heaviest neighbor, and, *matching*: if two vertices point to each other, add the edge to M; remove all edges incident on M, repeat.

Line 6 checks for an edge if the two endpoints mutually point to each other. If this is the case, then e is added to matching, and all the adjacent edges of e (including e) are removed from G. This continues until the graph becomes empty. We show an iteration with the two phases of LD-SEQ algorithm in Fig. 4.1.

Algorithm 4.1 LD-SEQ matching

Input: Graph: $G(V, E, w)$
Output: A locally dominant matching in *mate* array

1: $M \leftarrow \emptyset$
2: **while** G is not empty **do**
3: **for all** $v \in V$ **do** ▷ *Phase 1: Pointing*
4: $mate(v) = \arg\max_{u \in \mathcal{N}(v)} w(\{u, v\})$
5: **for all** $e(u, v) \in E$ **do** ▷ *Phase 2: Matching*
6: **if** $mate(v) = u$ and $mate(v) = v$ ▷ *LD edge*
7: $M = M \cup e$
8: $G = G \setminus \{e \cup \mathcal{N}(e)\}$

Lemma 4.3.2. *The matching M generated by Algorithm 4.1 is maximal and locally dominant.*

Proof. Let M be the current matching. An edge e is inserted into M iff the condition in

line 6 is true, which means u and v mutually point to each other. So, e is a locally dominant edge w.r.t. M. Since each edge, when inserted to M, is locally dominant according to Definition 4.3.1, M is also locally dominant. We continue until G is empty, which renders a maximal matching. □

The following corollary immediately follows from Lemma 4.3.2 and Lemma 4.3.1.

Corollary 4.3.1. *The matching M generated by Algorithm 4.1 is $\frac{1}{2}$-approximate.*

4.3.3 Related Work

Although MWM is solvable in polynomial time [48, 49], the high computational complexity and sequential nature of the optimal algorithm is prohibitively expensive for even moderate-sized graphs. As a result, in the last few decades, there have been several efficient approximation algorithms designed with different guarantees [5, 44, 45, 109, 111]. The breakthrough result for designing practical parallel algorithms is the locally dominant algorithm by Preis [111]. The locally dominant algorithmic framework is used in a number of shared and distributed memory algorithms for approximate matching. These include the pointer-based (aka LocalMax) algorithms [12, 93] and stable matching-based suitor algorithms (henceforth, SR-OMP) [94]. We refer to [110] for a detailed description of many of these algorithms.

Fagginger et al. [50] adapt the bipartite auction algorithm to implement a non-bipartite greedy matching by randomly coloring the eligible vertices blue or red, and they show that this algorithm can be implemented to GPU. However, the quality of the matching from this algorithm is shown to be subpar to subsequent work [12, 103]. Birn et al. [12] uses the pointer-based approach, while Naim et al. [103] employ the stable matching-based suitor algorithm (henceforth, SR-GPU). SR-OMP and SR-GPU are the state-of-the-art practical parallel approximate matching algorithms for shared memory parallel and single GPU methods. None of the existing GPU algorithms can be executed on multi-GPU systems, which is the primary contribution of this work.

To motivate our work, we consider recent works towards the culmination of multi-GPU methods across linear solvers and the applications of maximum weighted matching therein, requiring a scalable approximate matching in practice [10, 133] Recent works pertaining to scalable graph-related computations on multiple GPUs have been shown to utilize randomized or naturally ordered partitions across multiple devices [29, 67, 123]. For our purposes,

we draw from methods that employ batching and sampling strategies[32, 62, 150] to balance edge counts across devices while considering scalability relative to the number of devices allocated for computation.

4.4 GPU Implementation

We now discuss the development of our *locally dominant pointer-based* method implementation on GPUs, referred to as LD-GPU in the rest of the chapter. Considering a single node and multiple GPU configuration, we detail optimizations made through *batching*, graph structure implementation, and kernel design to improve the performance across a wide range of input graphs. For the following, we assume that we have N GPUs indexed through 1 to N.

4.4.1 Graph Distribution

We utilize the Compressed Sparse Row (CSR) format to store the nonzero elements in the graph, using separate vertex, edge, and value (edge weights) arrays, where edge information is stored as 64-bit integers. We distribute the graph G across N GPUs, where the i'th GPU has the subgraph $G_i(V_i, E_i)$ as input. To achieve that, we first form a partition of the vertex set, $V = \{V_i \subseteq V : i \in [1, N]\}$, where $V_i \cap V_j = \emptyset$ for $i \neq j$, and $\bigcup_{i\in[1,N]} V_i = V$. For each V_i, we compute E_i by choosing the subset of edges that have at least one endpoint in V_i. Formally, $E_i = \{\{u, v\} \in E : u \in V_i\}$. Note that the edge set E does not form a disjoint partition across the devices, since an edge can reside in multiple devices.

We partition the vertices with an attempt to assign similar #edges across the partitions (#vertices can be dissimilar) for improved load balance across devices, ensuring contiguous vertex IDs among partitions for coalesced global memory accesses on device. Each device is managed by an individual OpenMP thread that acts as its device index.

4.4.2 Batching

For many of the massive graphs in our benchmark, even the subgraph representations (i.e., G_is) do not fit into GPU memory. To tackle the memory limitations on devices and the irregularity of graph structured data, we adopt a *batching* scheme in our implementation. This batching scheme logically groups vertices assigned to a device for processing one group at a time. Given a device i, we further create a set of subgraphs (called batches) of G_i by

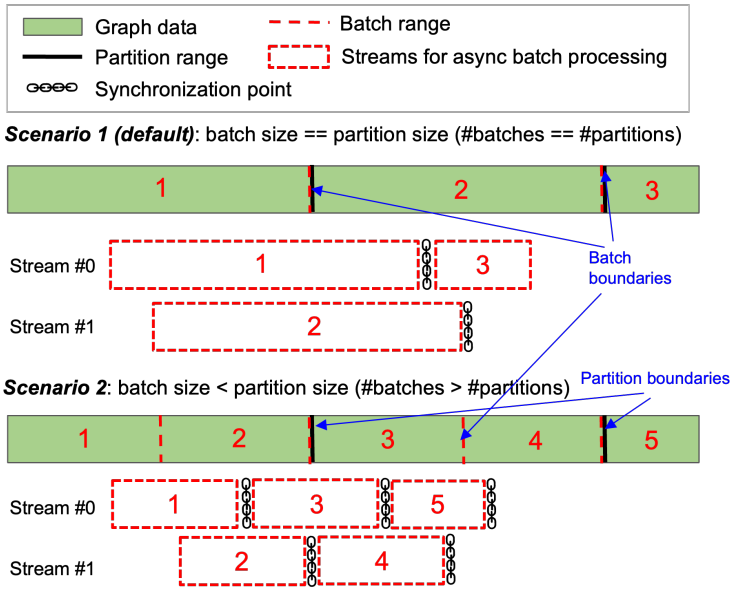

Figure 4.2: Scenarios concerning batches and partitions (on a single device) and depicting asynchronous batch processing through CUDA streams.

partitioning the vertex set V_i. We assume the ith device has N_i batches, and represent them as a set of integers, $\{1, \ldots, N_i\}$. The b'th batched subgraph is denoted by $G_i^b(V_i^b, E_i^b)$, where $b \in [1, N_i]$. Similar to the graph distribution, in the batched subgraph, the edge set E_i^b for the ith device contains all the adjacent available edges to the vertices (V_i^b) assigned to batch b. We use contiguous ranges of vertices to form a batch, following the device partition in the initial distribution of the graph data, as shown in Fig. 4.2. The purpose of batching is twofold: (i) considering single-node multi-GPU platforms, large graphs (as long as there is sufficient memory on node) can be accommodated on a variable number of devices (implicit or default scenario), and (ii) allowing logical control of task distribution on device for better balance on workload spreads (explicit working set control via batches less or greater than a partition).

The batch formation follows an edge-based scheme, implemented as a binary search on the prefix sums within our CSR representation. Coalesced allocations are maintained in batches given the contiguous nature of our initial partitioning. In the formation of batches, there can be multiple scenarios; two of them are outlined in Fig. 4.2. In both the cases, we maintain the initial device partition and consider batches of varying sizes relative to the original partition. We attempt to minimize the number of batches to reduce initial overheads associated with data transfer between the host and device. We also adopt a double buffering scheme for batch processing, and we use two GPU streams per device to asynchronously load

Algorithm 4.2 LD-GPU matching

Input: Graph: $G(V, E)$, $pointers[0 : |V|\,]$
Output: Matching in $mate[0 : |V|\,]$ array

1: **for** each GPU in parallel **do**
2: **while** there exist available matching edges **do**
3: **for** each batch b per GPU **do** ▷ *Pointing*
4: $stream \leftarrow b \bmod 2$
5: loadBatch<$stream$>(G_{id}, b)
6: setPointers<$stream$>(G^b_{id},$pointers$, $mate$)
7: `nccl_AllReduce`($pointers$)
8: setMates($pointers$, $mate$) ▷ *Matching*
9: `nccl_AllReduce`($mate$)
10: **procedure** LOADBATCH<$stream$>($G_i(V_i, E_i)$, b)
11: $v\_batch \leftarrow V_i^b$
12: `cudaMemcpyAsyncHtoD`(v_batch, $stream$)

data and compute. Ideally, #batches can be optimized relative to scalability or to exploit underlying program logic. We discuss the scalability aspect of multiple batches in §4.5.

4.4.3 Intermediate Data Sharing

Multi-GPU implementations can consider a peer-to-peer approach for sharing intermediate data between devices, supported by unified virtual addressing in contemporary GPUs. There are peer-access APIs to bypass host for inter-device transfers; on-demand paging is another option. These options are convenient for regular data sharing scenarios, irrespective of applications, when the data must be shared after certain synchronization points, usually following independent computation. However, in our case, devices work on different batch ranges concurrently (#batches are the same per device), and there can be a situation during matching (see Algorithm 4.1), where there is a dependency on a previous or next batch. Hence, we had to adopt a conventional bulk-synchronous approach in batching, without which we would have to contend with numerous conflicting scenarios of device memory loads, especially as batch counts increase.

Primary conflicts revolve around instances where required edge information is not present on concurrently-loaded batches in the devices. Such scenarios require either the host to retrieve this information or to impose restrictions on batches based on inter-dependencies, which can substantially increase with the rising #batches. Thus, we adopt a vertex-based approach to impose independence in the setting of vertex pointers, no matter the batch distribution. This further allows us to tweak batch counts without restriction for optimal data

distribution given the irregularities within edge information. One trade-off for this method, however, is the requirement to store global matching information on each device. For our purposes, this requires two arrays of size $|V|$ to be allocated on each device. Given the usage of batching and the relative memory complexity of vertices to edges being trivial, we accept this trade-off for ease of implementation and device communication.

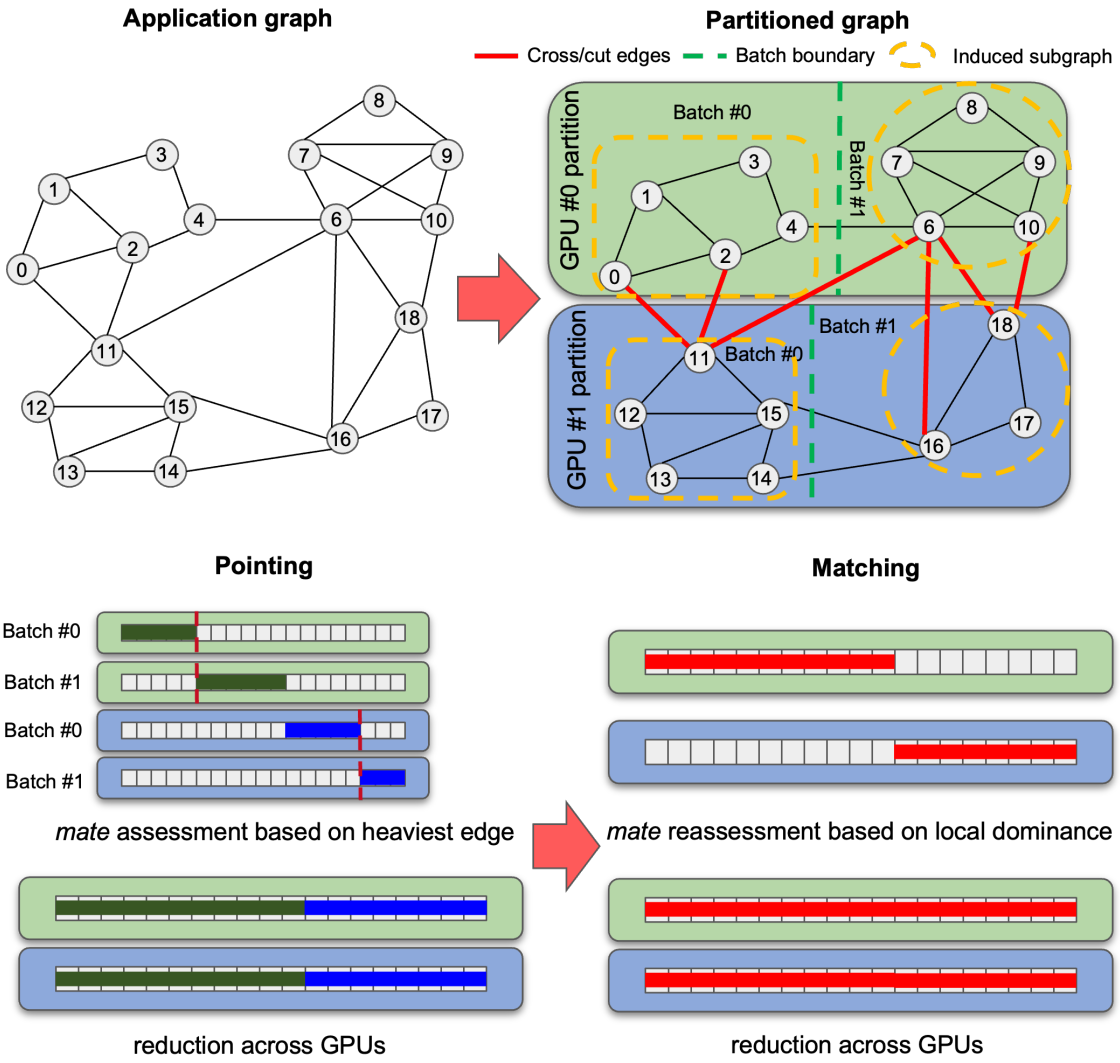

Figure 4.3: LD-GPU algorithm illustration considering partitions and batches using multiple GPUs. A graph is first partitioned among devices and logically arranged into ranges of vertices (and adjacent edges) called *batches*. Each batch is processed independently through the pointing phase, followed by a global reduction, the matching phase and another global reduction to synchronize device matching information.

4.4.4 GPU Implementation

We now discuss the details of our GPU implementation, LD-GPU. We provide a general overview of the algorithm in Fig. 4.3, which depicts the *pointing* and *matching* phases, as introduced in Algorithm 4.1. We first provide a high-level description in Algorithm 4.2 followed by specific kernels in Algorithm 4.3.

Algorithm 4.3 Matching Kernels

```
1:  procedure SETPOINTERS<stream>(G^b(V^b, E^b),
    pointers, mate)                                  ▷ Batched Graph Data G^b
2:      buffer ← V^b[stream]
3:      for u ∈ buffer in warp do
4:          p ← ∅
5:          if mate[u] = ∅
6:              for v ∈ N(u) and mate[v] = ∅ per thread do
7:                  p ← arg max_{x∈{v,p}}{w({u, x})}
8:              shuffle_reduce(p)                    ▷ Across Warp
9:              pointers[u] ← p
10: procedure SETMATES(pointers, mate)
11:     for u per thread do
12:         if pointers[pointers[u]] = u             ▷ Mutual Check
13:             mate[u] ← pointers[u]
```

4.4.4.1 Algorithms

After graph loading and distribution, our algorithm proceeds as follows: each GPU's host thread iterates through its batches, sequentially loading and processing batch data for the initial *pointing* phase. We utilize a dual-buffer method to overlap communication and computation of successive batches over CUDA streams (shown in Fig. 4.2). Specifically, we allocate two buffers per device, such that the loading and processing of batches can occur concurrently using asynchronous CUDA streams (denoted by *stream* in lines 4-6). Thus, we only have to synchronize between successive batch invocations when the #batches are greater than two. When there are one or two batches, there is no extra synchronization between the respective batch invocations due to the separate buffers. In the cases of a higher number of batches, we sequentially perform these load and processing steps, interleaving batches between the buffers and performing host-device synchronization after determining the heaviest *available* edge information for the vertices in each batch (lines 5-6). Recall from Definition 4.3.1, an available edge is an edge that can be added to the current matching

without violating the matching constraint. This *pointing* phase identifies and sets pointers along these highest weighted neighbor edges for each vertex, independently. Then, we invoke a reduction of the pointer information across the GPUs using NCCL reduction routines [70], ensuring that all devices contribute and obtain the global pointer information and synchronize, before moving on to the next phase (line 7).

For the *matching* phase, we maintain global matching information on device and perform mutual checks independently using the pointer information obtained from the *pointing* phase (line 8). Any mutually-pointing vertices are committed to the matching. We do not require batching in this phase, since we only reference the aggregated pointer information for mutual checks. We perform another NCCL `allreduce` to synchronize the matching information across devices (line 9). Given the global matching information for an iteration, a device can then decide to terminate if no new edges were added to the matching. This process repeats until no more available matching edges exist.

4.4.4.2 Kernels

For the *pointing* phase, we distribute contiguous groups of vertices within the current batch across warps (a *warp* is 32 threads). These groups are assigned a stream in our dual buffer allocation based on the batch number (line 2). Each warp then sequentially processes its assigned vertices, with the threads concurrently iterating over the neighborhood/ adjacencies of the current vertex. Each thread performs a reduction on its subset of the neighborhood to determine the heaviest active edge (lines 5-7), which is further reduced using a bandwidth-efficient warp-level shuffle reduction utilizing registers, communicating the heaviest active neighboring edge across the warp (line 8). This edge is stored in an array at the global device memory (`pointers` in Algorithm 4.3), and the process continues for each vertex assigned to the warp.

In the *matching* phase, we check the list of vertices, without scanning the individual neighborhoods. We can perform the mutual pointer check (line 12) after distributing the vertices evenly among the threads to limit load imbalance, assigning contiguous groups of vertices to each thread, and performing a global memory check and a subsequent write if a mutual pointer exists (locally dominant edge). Although the global memory check can lead to suboptimal performance due to non-coalesced memory accesses arising from indirect indexing, in practice we found the pointing phase to be more expensive, as discussed in §4.5.

We further invoke a device-wide reduction on the global matching information, to ensure consistency of the mutual checks across the iterations of the *matching* phase.

Next, we show that our LD-GPU algorithm provides the $\frac{1}{2}$ approximation guarantee as the LD-SEQ algorithm.

Lemma 4.4.1. *The matching produced by LD-GPU is $\frac{1}{2}$-approximate for MWM.*

Proof. It is sufficient to show that the edges committed to matching (Line in Algorithm 4.3) in LD-GPU are locally dominant w.r.t. the current matching and the final matching is maximal. The proof then follows from Lemma 4.3.1. We note that the only difference between the sequential LD-SEQ algorithm (Algorithm 4.1) and LD-GPU (Algorithm 4.2) is that, in LD-GPU the graphs are distributed across the devices by using a non-overlapping vertex partition scheme. However, since we include all the adjacent edges of the set of vertices assigned to the device, the edge distribution may overlap. In a device for a vertex u pointing to v in the *pointing* phase there are two cases : (i) v is in the induced subgraph (the edges inside the yellow dashed box in Figure 4.3), and (ii) v is in the cross/cut edges (the red edges in Figure 4.3). For the first case, we can immediately decide whether v also points to u; for the second case, since v does not reside in the particular GPU, we do not know who v decides to point. However, after the *pointing* phase, we are synchronizing the *pointers* array across all devices (Line 7 in Algorithm 4.2). Since the vertices form a non-overlapping partition, this reduction is unambiguous. After the reduction, if u and v point to each other, $\{u, v\}$ must be a locally dominant edge, which we are checking in setMates function. Furthermore, after the *matching* phase, we are also synchronizing the *mate* array globally to reflect the current matching across all devices. The algorithm continues until we have any more edges to match, which renders a maximal matching. □

4.5 Evaluations

In this section, we present detailed quality and performance assessments of LD-GPU on two NVIDIA GPU platforms, comparing the performance/quality with state-of-the-art CPU/GPU implementations. We have excluded graph related I/O, allocations (host/device), CSR construction and host-device partition transfer times from the reported execution times, and only include the time (in seconds) for the *pointing* and *matching* phases on GPUs. We report the best times over ten runs. We acknowledge that for large graphs, graph I/O/

Table 4.1: (Left) Graph datasets and properties, where $|V|$ and $|E|$ are the graph vertex and edge cardinalities, d_{max} and d_{avg} are the graph maximum and average degrees, and B, M, and K refer to $\times 10^9$, $\times 10^6$, and $\times 10^3$, respectively. (Right) Best execution times (s) over ten runs per algorithm. LD-GPU demonstrates better performance relative to existing CPU/GPU implementations (SR-OMP/SR-GPU) for 9/14 graphs, depicting 2–45× speedup for billion-edge graphs relative to SR-OMP. '-' refers to tests that failed due to out-of-memory errors.

	Properties				Best Execution Time (s)				LD-GPU Vs.	
Graphs	$\|V\|$	$\|E\|$	d_{max}	d_{avg}	SR-OMP	SR-GPU	LD-GPU (#GPUs)	LEMON	SR-OMP	SR-GPU
AGATHA-2015	184 M	5.8 B	12.6 M	63	36.07	-	**16.04(8)**	-	2.2×	-
uk-2007-05	105 M	3.3 B	975 K	62	N/A	-	**2.44(8)**	-	-	-
webbase-2001	30 M	3.3 B	2.1 M	220	N/A	-	**49.29(8)**	-	-	-
MOLIERE_2016	134 M	2.1 B	68	32	46.08	-	**11.16(8)**	-	4.1×	-
GAP-urand	134 M	2.1 B	1.5 M	31	17.66	-	**0.319(8)**	-	45.4×	-
GAP-kron	118 M	1.9 B	816 K	17	9.53	-	**0.389(4)**	-	24.4×	-
com-Friendster	65 M	1.8 B	5 K	55	8.40	**0.661**	0.693(6)	-	12.1×	0.95×
Queen_4147	4 M	317 M	81	79	0.332	**0.008**	0.018(4)	323.5	18.4×	0.44×
mycielskian18	196 K	301 M	98 K	1530	0.113	0.025	**0.019(1)**	488.6	5.9×	1.32×
HV15R	2 M	283 M	484	140	0.240	0.047	**0.032(4)**	217.5	7.5×	1.47×
com-Orkut	3 M	234 M	33 K	76	4.351	**0.036**	1.215(4)	221.8	3.6×	0.03×
kmer_U1a	68 M	139 M	70	4	0.798	**0.048**	0.152(4)	323.5	5.2×	0.32×
kmer_V2a	55 M	117 M	30	2	0.636	**0.058**	0.131(1)	271.8	3.6×	0.44×
mouse_gene	45 K	28 M	8 K	642	0.041	0.016	**0.013(1)**	488.6	3.1×	1.23×

preparation times can be significant relative to the overall times spent in the phases, but this is unavoidable, regardless of the matching algorithm. Details on the experimental platforms and input datasets are below.

4.5.1 Datasets

We perform evaluations using fourteen graphs with varying sizes (from 28M to 5.8B edges) and structural properties. Most of these graphs are collected from the SuiteSparse Matrix Collection [39], except uk-2007-05 and webbase-2001, which are web-crawl graphs from the LAW collection [19]. In cases where natural edge weights were absent from the datasets (weights not present or assigned 1), we sample weights from a uniform distribution range. Our datasets are listed in Table 4.1, organized into groups: (i) Small: graphs with #edges $\leq$ 1B, and (ii) Large: graphs with #edges $>$ 1B.

4.5.2 Platforms

We use two GPU platforms in our evaluations: NVIDIA™ DGX systems with 8/16 GPUs (DGX-A100 and DGX-2). The NVIDIA™ DGX-2 V100 platform consists of a single

node with 16 "Volta" V100 GPUs (with 80 symmetric multiprocessors a.k.a SMs) with 32GB HBM2 memory/GPU and two-way 24-core Intel Xeon P-8168 CPUs ("SkyLake" or SKL) at 2.7GHz, 33MB L3 cache, 6 memory channels, and 1.5TB DDR4 memory. DGX A100 is the third generation server node from NVIDIA™, and consists of 8 "Ampere" A100 GPUs (with 108 SMs) with 40GB HBM2 memory/GPU and two-way 64-core AMD EPYC 7742 CPUs at 2.25GHz, 256MB L3 cache, 8 memory channels, and 1TB DDR4 memory. NVIDIA GPUs either come in the PCIe or proprietary NVLink/NVSwitch based form factors. Proprietary SXM module allows NVIDIA GPUs to directly communicate through NVLink interconnect (DGX-A100 uses SXM4 whereas DGX-2 V100 interconnect uses SXM3). We use CUDA version 10.1.243, OpenMP version 11.2.0 and NCCL version 2.8.3.1 on both platforms. Our comparative CPU runs utilize a server with similar specifications as the A100 system, having two-way 64-core (256 threads) AMD EPYC 7742 CPUs at 2.25GHz, 256MB L3 cache, 8 memory channels, and 2TB DDR4 memory.

The rest of this section is organized as follows. We begin the discussion by comparing the quality of the matching produced by LD-GPU, relative to optimized CPU/GPU implementations, in §4.5.3. Then, we analyze the execution time performance and scalability of LD-GPU against various inputs and systems in §4.5.4. In §4.5.5, we dissect the GPU utilization of LD-GPU considering variations in graph structure and resulting distributions. Finally, we compare the overall performance of LD-GPU with state-of-the-art OpenMP-based CPU (SR-OMP) and single GPU (SR-GPU) implementations.

4.5.3 Matching Quality

We compare the quality of our LD-GPU and the multi-threaded SR-OMP with the sequential optimal `MWM` algorithm included in the Library of Efficient Models and Optimization in Networks (LEMON) [41]. We exclude SR-GPU as we observe the SR-GPU weights are very close to the SR-OMP ones. We are able to only execute LEMON on the SMALL instances since the LARGE graphs resulted out of memory conflicts. In Table 4.2, we show the percentage difference of weights of LD-GPU and SR-OMP algorithms relative to the LEMON. Here, the lower is the better. Across our SMALL inputs, we observe high quality matching output by LD-GPU, with only 6% difference from the optimal, on geometric mean. LD-GPU and SR-OMP achieve a similar quality, which we attribute to both algorithm's greedy approach in approximate maximum weighted matching. These results suggest that although our LD-GPU

algorithm is $\frac{1}{2}$-approximate in the worst case, in practice, we achieve close to optimal quality.

Table 4.2: LD-GPU and SR-OMP quality percentage difference relative to LEMON on the Small graph instances.

	Percentage Diff.	
Graphs	LD-GPU	SR-OMP
Queen_4147	4.8	4.7
mycielskian18	12.5	12.6
HV15R	2.8	2.8
com-Orkut	2.6	2.6
kmer_U1a	8.9	9.0
kmer_V2a	9.9	9.9
mouse_gene	11.2	11.3
Geo. Mean	6.38	6.38

4.5.4 Baseline Performance

4.5.4.1 Scalability

Fig. 4.4 presents strong scaling results on 1–8 A100 GPUs using the large inputs; we picked the best results for every configuration by considering a range of batches (less than 15) on up to 4 devices. Beyond 4 devices, each partition fits into a device, and we can avoid the batch processing related overheads. We observe up to 47× speedup on 8 GPUs relative to a single GPU. This superlinear speedup is due to the sequential nature of batch processing in the *pointing* phase and the associated synchronization and data transfer overheads for the low device counts, which can be optimized away by increasing the number of device partitions. When the batch processing overheads are absent, the scalability plateaus beyond 4 GPUs for most of the large inputs as the scalability from the *matching*/*pointing* phases are offset by the rising costs of collective operations and synchronizations at higher device counts. Details about the relative costs of the high-level components in LD-GPU are discussed next, in §4.5.4.2.

To study the scalability potential of batches, we subject relatively small inputs (to ensure a single partition per device) with higher batches on multiple devices (deliberately introducing nontrivial batch processing overheads). We present the results on the kmer_U1a, mycielskian18 and kmer_V2a graphs in Fig. 4.6. For these instances, the default scenario (single batch/partition) does not exhibit any scalability with increasing the #devices, as the collective reduction/synchronization overheads offset improvements in the *matching* phase,

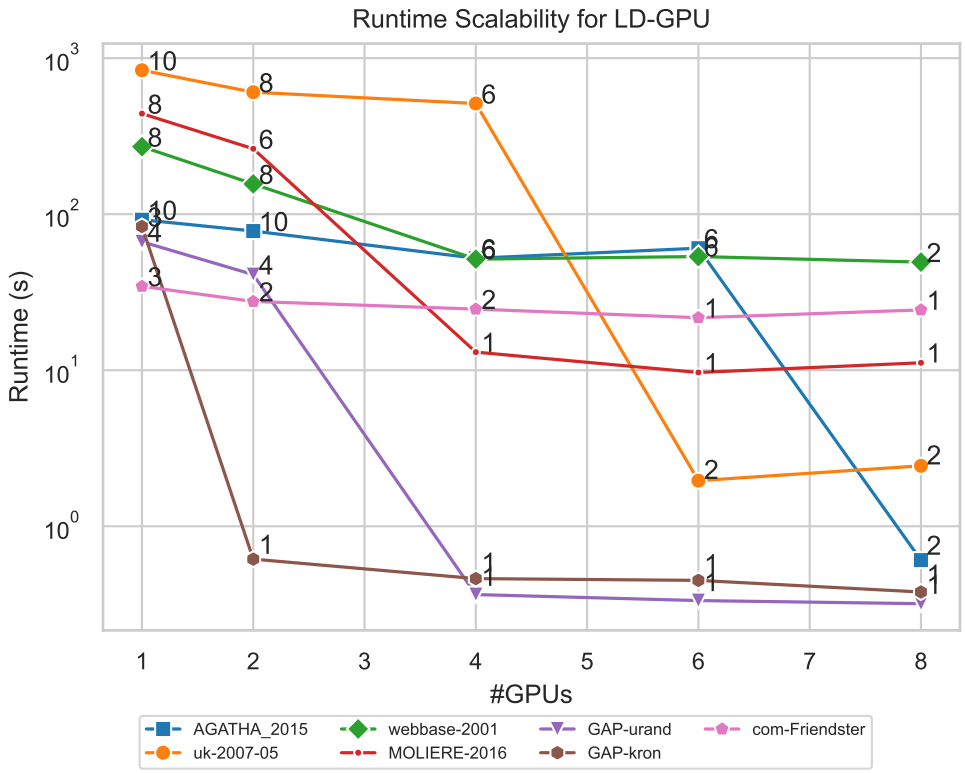

Figure 4.4: Strong scaling for LD-GPU on 1–8 GPUs, using a variety of batch counts and choosing the best execution time over 10 runs.

as shown in the component-wise timing in Fig. 4.7. Increasing the #batches, we observe a more balanced distribution of the independent work (pertaining to the *pointing* phase) and overall data movement/synchronization, despite batch transfer overheads (observe enhanced scalability for 3, 5 and 10 batches in Fig. 4.7 and Fig. 4.6). We anticipate subsequent batch transfer overheads would ultimately impact the scalability beyond a certain point.

4.5.4.2 Component-wise Timing Analysis

In Fig. 4.5, we examine the individual execution times of the high-level components in Algorithm 4.2 for different batches on 1-8 GPUs, considering LARGE and SMALL graphs. We track the individual contributions of the *pointing* and *matching* phases, `allReduce` operation for collecting the global pointers and mate information, batch range related data transfers to device and explicit synchronizations. For the com-Friendster and GAP-kron graphs, we use batches for up to 4 GPUs to accommodate multiple partitions on a device; otherwise, we proceed with the default single batch version (even a single batch uses dual buffers, as explained in §4.4.2). Fig. 4.5 conveys that synchronization and communication costs dominate about 90% of the overall execution time, excluding single GPU runs. In the single

GPU and non-default batching scenarios, the *pointing* phase take about 50% of the overall execution time, as sequential batch processing increases the (local/independent) computation overheads as well. This is similar to our observations on a handful of SMALL graphs where we demonstrated that considering reasonable #batches can increase the scalability relative to the default scenario (see §4.5.4.1). Thus, we see a direct relation in vertex-batch distribution with scalability across the devices, for LD-GPU. Also, due to greater than 50% of the overall time spent in collective communication and synchronization, LD-GPU depends on the efficiency of the underlying communication runtime and GPU interconnection network. Impact on the performance due to GPU platform interconnect is discussed next, in §4.5.4.3.

4.5.4.3 NVIDIA Ampere (A100) vs. Volta (V100) Platforms

To further evaluate the impact of the GPU platform, comparing between generations of device and GPU interconnects, we analyze the performance of LD-GPU considering— (i) devices/interconnects: NVIDIA Ampere (A100) vs. Volta (V100) GPUs, and, (ii) standardized vs. proprietary interconnect: PCIe vs. NVLink (SXM4) on DGX-A100. Table 4.3 highlights the performance impact of the GPU generation by comparing contemporary NVIDIA™ "Am-

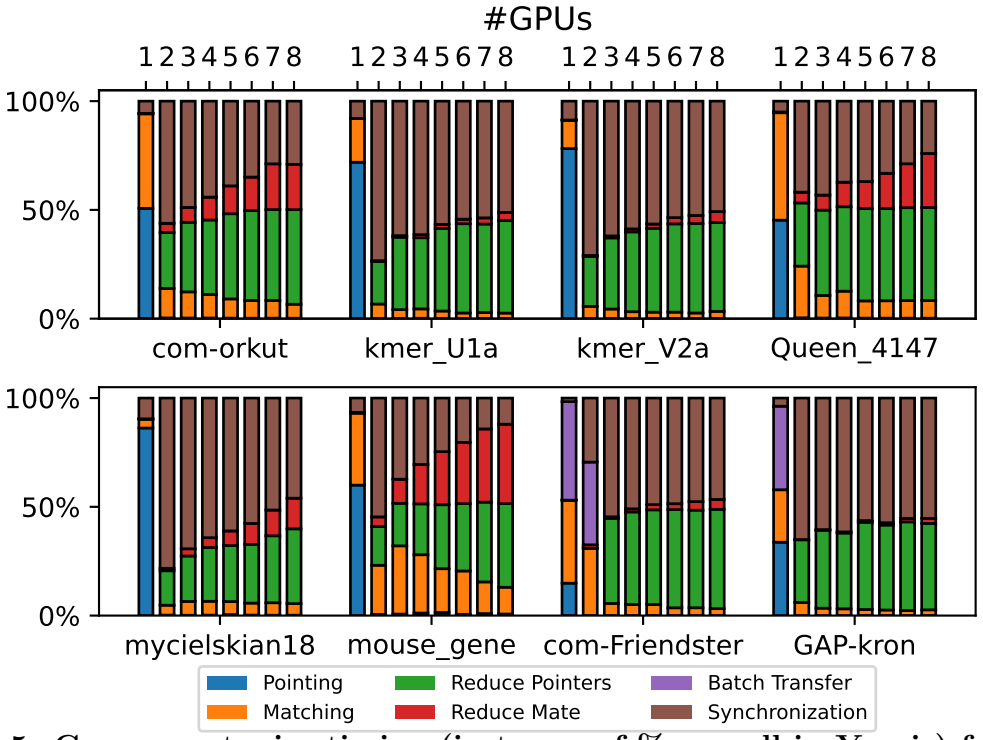

Figure 4.5: Component-wise timing (in terms of %-overall in Y-axis) for Small/Large graphs (X-axis) for variable #batches/GPU on 1–8 GPUs.

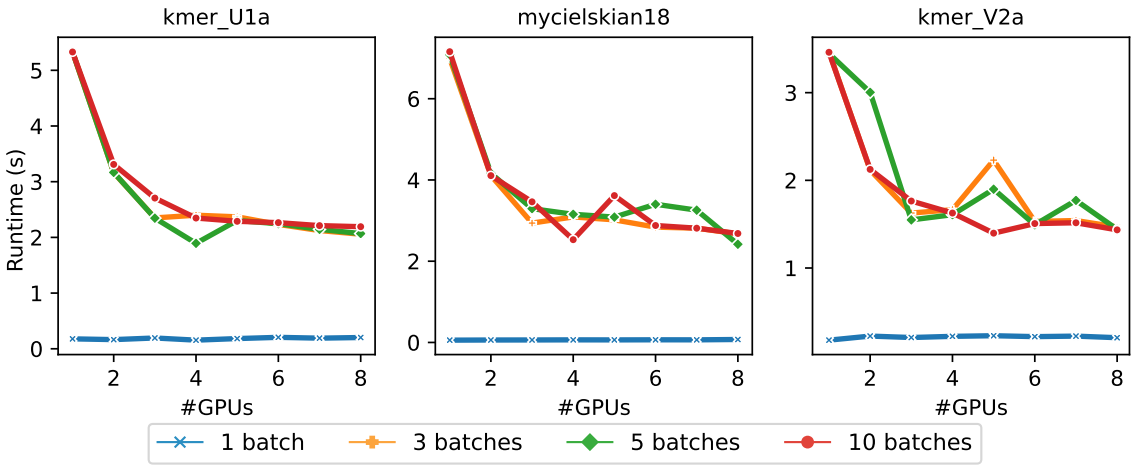

Figure 4.6: LD-GPU using 1 (default), 3, 5 and 10 batches on 1–8 GPUs.

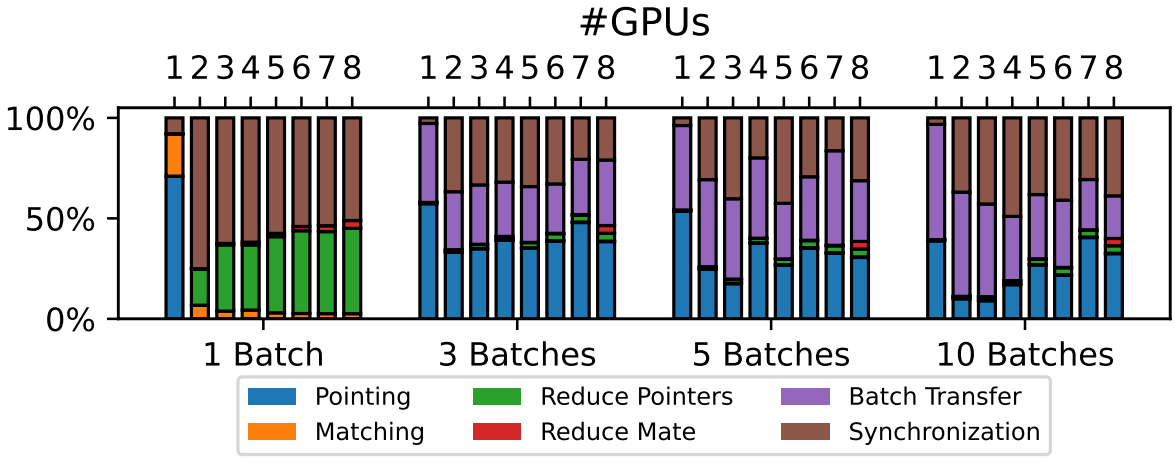

Figure 4.7: Component-wise timing (%-overall in Y-axis) for kmer_U1a graph using LD-GPU with 1 (default), 3, 5 and 10 batches (X-axis) on 1–8 GPUs.

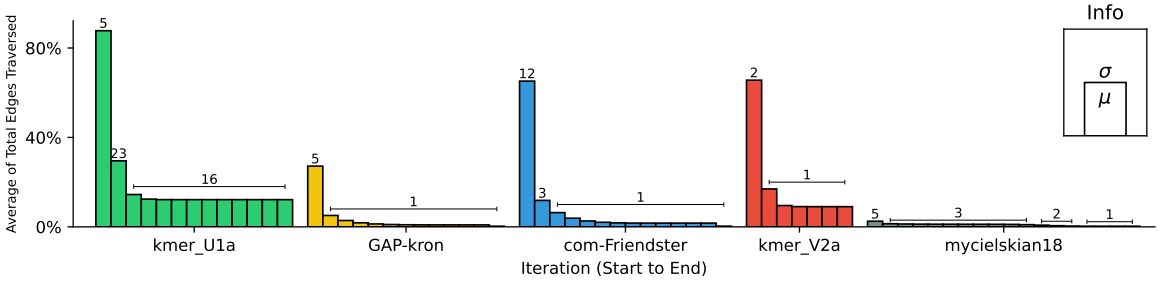

Figure 4.8: Mean and standard deviation of % of edges accessed by warps on *pointing* phase iteration of LD-GPU—for 90% of the iterations, less than 20% of the edges are accessed.

pere" A100™ vs. previous "Volta" V100™, reporting the speedup of LD-GPU on A100™ using SMALL graphs relative to V100™. We use a single device to capture the performance independent of device communication and batch processing. We observe about 2-4× improvements

on contemporary A100 vs. previous-generation V100 GPU.

Table 4.3: LD-GPU speedup on a single NVIDIA A100 vs. V100.

Graphs	A100 Speedup
Queen_4147	1.07×
mycielskian18	2.05×
com-Orkut	2.47×
kmer_U1a	4.56×
kmer_V2a	4.53×
mouse_gene	1.49×
Geo. Mean	2.35×

We assess the impact of the GPU interconnect, PCIe vs. NVLink (SXM4), in Fig. 4.9. Foley, et al. [54] report 5× the bandwidth of PCIe using proprietary NVLink (on previous-generation NVIDIA™ P100™ GPU). We consider SMALL and LARGE inputs, with GAP-kron and com-Friendster using batching for GPU counts less than 4. Given the reliance of LD-GPU on fixed synchronization points around global device-based collective operations, we observe average performance improvements of 3× with NVLink over PCIe interconnect (maximum performance improvement was about 17×). We observe the outlier input mouse_gene (smallest graph), which actually demonstrates relatively mild and stable collective communication overhead up to 4 GPUs (see Fig. 4.5). Hence, we expect non-trivial end-to-end improvements with enhanced GPU interconnects across the vendor generations for larger graphs. Fig. 4.10 compares LD-GPU scalability on NVIDIA™ DGX-A100 (8 A100 GPUs with NVLink SXM4) with previous generation DGX-2 (16 V100 GPUs with NVLink SXM3) for two diverse LARGE inputs over the same #batches. While GAP-kron exhibits a maximum of up to 8× improvement on 8 A100 GPUs as compared to 16 V100 GPUs, com-Friendster demonstrates a maximum improvement of about 10× for the same range. We observe a significant increase in the execution times on V100 GPUs with rising matching iterations, for e.g., GAP-kron exhibits 15 iterations for LD-GPU, whereas com-Friendster runs for around 2,000 iterations.

4.5.5 GPU Utilization

In this section, we demonstrate the challenges in maintaining load balance throughout the progression of the *matching* and *pointing* phases on device. The *pointing* phase determines the heaviest active neighbor edge for a vertex, while the *matching* phase iterates over the remaining unmatched vertices, "removing" edges from matching. Specifically, we ana-

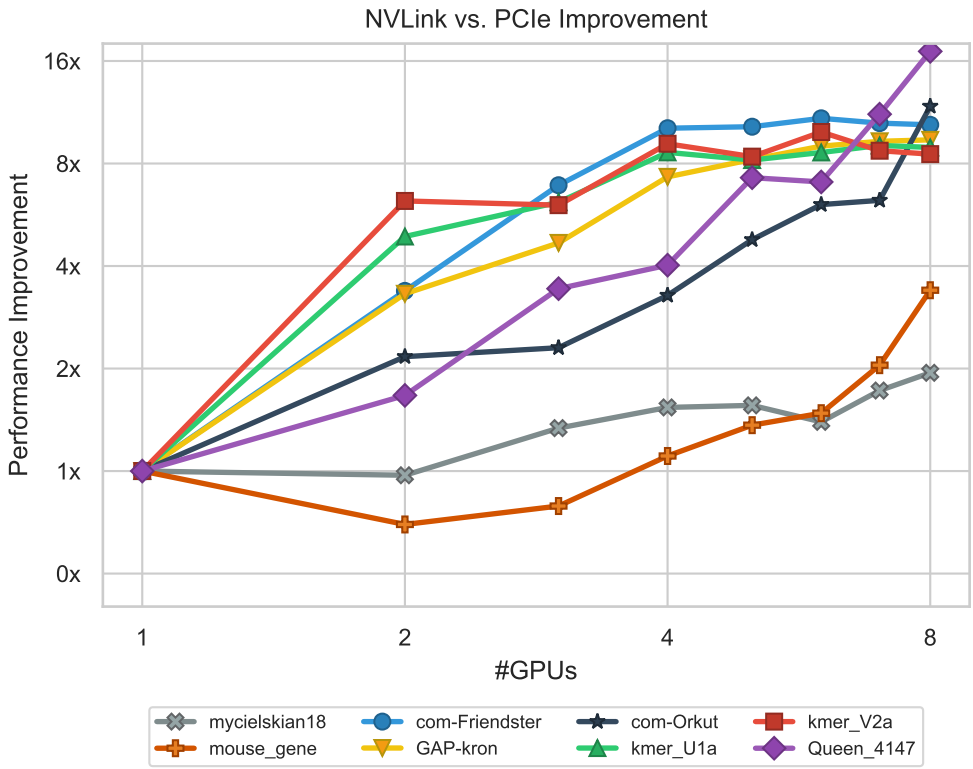

Figure 4.9: Execution time speedup of NVLink vs PCIe for data transfer and multi-GPU communication for LD-GPU.

lyze the amount of edges processed on individual iterations and relate it to the Streaming Multiprocessor (SM) occupancy to assess the work efficiency on device.

4.5.5.1 Warp-Edge Work

The notion of warp-edge work in LD-GPU can be expressed by the volume of consecutive edge traversals during the *pointing* phase to determine the pointer candidate per vertex neighborhood, on a per warp basis. We consider the total number of edges traversed throughout the matching progression across the iterations. Fig. 4.8 depicts SMALL and LARGE inputs, capturing the mean and standard deviation of percentages of the edge traversals across the matching iterations, where each bar represents an iteration of the respective input on LD-GPU.

Despite similar iteration counts across the inputs, we observe approximately 2–5× differences in the variance and peak warp-edge work amounts. On average, majority of the edge traversals are performed in the first iteration of matching (thus, first iteration is the most expensive). Depending on the graph structure and device partitions, we observe cases such as the kmer_U1a having a relatively high variance in the distribution of warp-edge

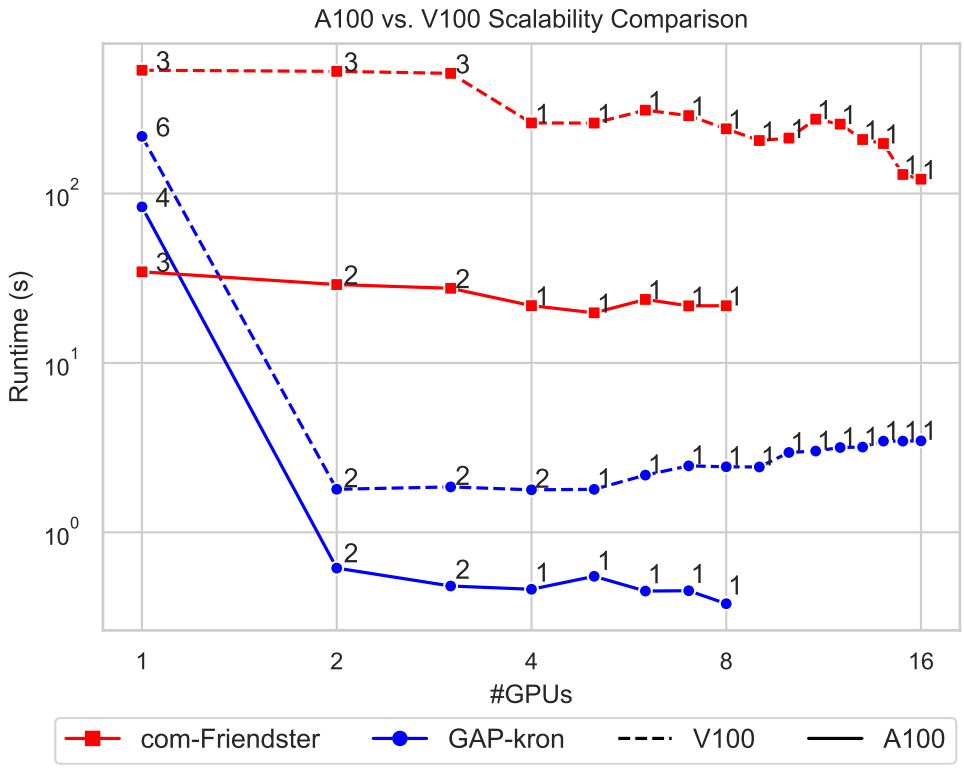

Figure 4.10: LD-GPU scalability on the dense-GPU systems with annotated #batches: DGX-2 (16 V100s) vs. the DGX-A100 (8 A100s).

work on the second iteration. Meanwhile, there are cases such as GAP-kron which exhibit relatively even distribution of warp-edge work throughout the iterations. We capture the overall variations in the densities of the warp-edge work in Fig. 4.8 and to study the device occupancy, as discussed next.

4.5.5.2 Streaming Multiprocessor (SM) Occupancy

Extending the edge-work notion to GPU utilization, we examine the SM occupancy on runs with different graph inputs. We track SM occupancy per groups of the kernel launches, taking an average (over batches) of the *pointing* and the *matching* phases in an iteration. Fig. 4.11 depicts about 90% SM occupancy through 100% of the program iterations for most cases, except the outliers (mycielskian18 and mouse_gene), which exhibit diverging behavior and at the lowest point demonstrates 30%/50% occupancy for the later half (50% mark) of the iterations. Considering the *pointing* phase invokes repetitive neighborhood scan, memory accesses are mostly contiguous, indicative of relatively high SM occupancy, which is a favorable trait for sustainable performance.

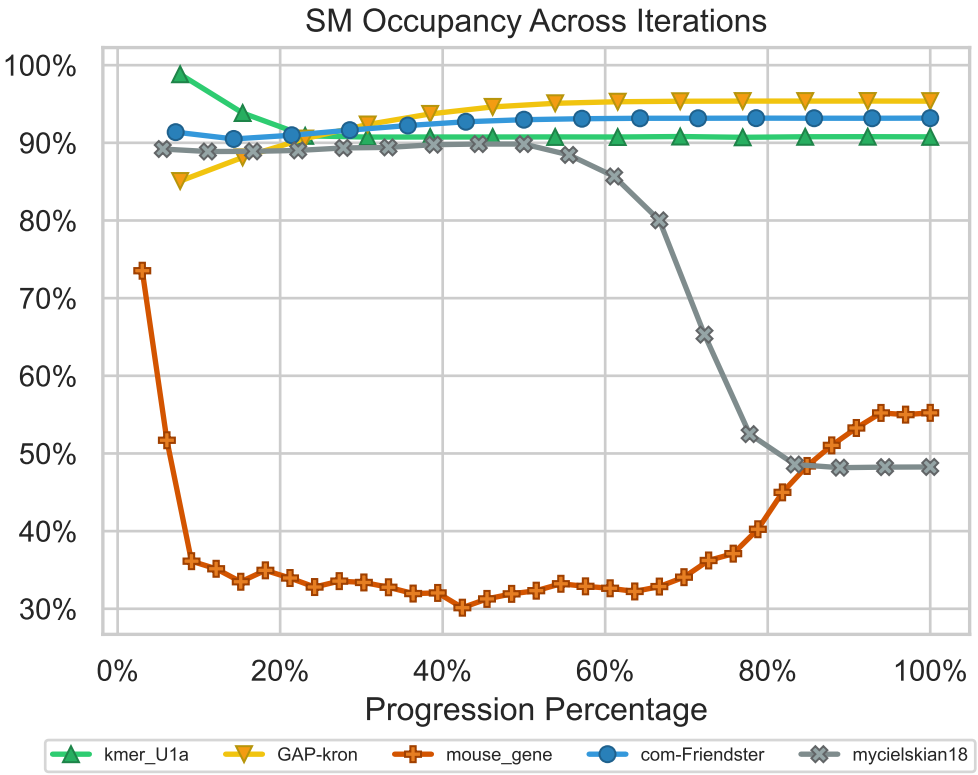

Figure 4.11: GPU Streaming Multiprocessor (SM) occupancy (Y-axis, higher is better) as reported through NVIDIA Nsight profiler per iteration of LD-GPU (X-axis shows iteration progression in terms of %).

4.5.6 Performance Comparisons

To assess the results of our LD-GPU implementation, we consider two state-of-the-art parallel weighted graph matching implementations for comparison: the OpenMP Suitor algorithm (SR-OMP) discussed in [94, 155] and the GPU Suitor (SR-GPU) algorithm in [102, 103]. The Suitor algorithm is an improvement over locally dominant matching algorithm, as the former is able to reduce the number of candidate edges for matching. We further include a sequential baseline in Edmond's Blossom algorithm implementation in the LEMON matching collection in [83]. SR-OMP results are collected using 256 CPU threads while SR-GPU results are collected on a single NVIDIA™ A100 GPU.

4.5.6.1 Execution Time Performance

We compare the results of LD-GPU method to the SR-GPU and SR-OMP implementations for the graphs listed in Table 4.1. For LD-GPU, we consider several device and batch counts to find the best performance. While higher batch counts typically increase execution times for LD-GPU given initial data loading and synchronization overheads, for large and

massive graphs, we can leverage multiple devices for improved partition distribution, outweighing these costs. Table 4.1 lists the best execution times of LD-GPU, SR-GPU, SR-OMP and LEMON, and the speedup of LD-GPU relative to SR-GPU and SR-OMP implementations (it is unfair to compare with LEMON since it is sequential).

Relative to SR-OMP, we observe performance improvements of 2-45×, with a geometric mean of about 7×. In cases such as the mycielskian18 graph, we obtain the highest improvement with a single GPU, while other instances such as the kmer_U1a graph shows a better performance across multiple devices. In practice, we notice that denser graphs perform better when less devices are used, as performance gains using multiple GPUs are often outweighed by the communication overheads between partitions and above-average iteration counts. Across the SMALL instances, we report a geometric mean performance improvements of approximately 5×. Our LARGE instances demonstrate improvements w.r.t SR-OMP of approximately 6× on average. For the largest (in terms of #edges) three graphs in our dataset, we are required to apply batching on our maximum GPU count of eight since one or more partitions could not fit into the available device memory. Among the LARGE inputs, AGATHA-2015, uk-2007-05 and MOLIERE_2016, performed best on relatively larger GPU counts using 2 batches. For uk-2007-05 and webbase-2001, SR-OMP comparison is omitted since SR-OMP requires graphs to be in Matrix Market native data format. GAP-kron and GAP-urand exhibit significantly greater improvements compared to other graphs, most likely due to their synthetic nature and atypical degree distribution. We now discuss the performance of SR-GPU (a single-GPU implementation), for which LD-GPU shows competitive results on a variety of midsize graphs. We omit the comparison results for the majority of the LARGE instances, as we experienced "out of memory" issues with SR-GPU. On 4/7 SMALL instances, SR-GPU is on average 2× faster than LD-GPU, since it optimizes for computation on a single device. In contrary, our goals are to consider larger graphs for efficient multi-device computation, and we observe up to 1.47× speedup relative to SR-GPU using over multiple batches and devices. Overall, SR-GPU shows performance improvements relative to LD-GPU for multiple midsize instances, but is unable to run on our LARGE instances, excluding the com-Friendster graph (SR-GPU uses 32-bit graph representation, while we have adopted 64-bit).

4.5.6.2 Figure of Merit

Comparing parallel maximum weighted matching methods on the basis of execution time only is beset with challenges. Different implementations might adopt various techniques and heuristics to optimize the performance/quality targeting diverse architectures; unless a baseline metric or Figure-of-Merit (FoM) is devised, comparing relative performances under different parameter settings will remain challenging.

For graph matching, a prospective FoM must consider the total #iterations, matching quality, edges in matching and the execution time performance. To that effect, we propose a new FoM: "Mega-Matching Edges per Second" (MMEPS). In essence, we correlate the rate at which edges are committed to the matching, to the enhance the quality over the iterations. We provide instances of comparison on variable size inputs in Table 4.4. For each case for LD-GPU, we collect the best FoM (higher is better) for invocations across devices and compare to the best of the 10 runs of SR-OMP. Under this FoM, LD-GPU demonstrates 2–20× improvements relative to SR-OMP.

Table 4.4: Mega-Matching edges per second (higher is better).

	FoM (MMEPS)	
Graphs	LD-GPU	SR-OMP
AGATHA-2015	8.14	3.77
MOLIERE-2016	1.28	0.31
GAP-urand	41.99	7.37
GAP-kron	29.63	1.21
com-Friendster	37.84	3.12
kmer_U1a	191.35	39.99

4.6 Concluding Remarks

In this chapter, we discuss our parallel algorithm for locally dominant maximal weighted graph matching for multiple GPUs on single node NVIDIA DGX™ platforms. Despite the irregularities in the graph structure and the divergent computation patterns of locally dominant matching (i.e., *pointing* and *matching* phases), we report 2–45× performance improvements of our multi-GPU implementation relative to state-of-the-art OpenMP-based CPU (on 256 threads) for billion-edge graphs. In the next section, we discuss a relevant application of such approximate matching methods to set similarity, a popular topic within web data analyses and data cleaning.

CHAPTER 5
APPROXIMATE MATCHING FOR FUZZY SET SIMILARITY

5.1 Introduction

In this chapter, we discuss a widespread problem in data analysis, search and management operations in set similarity. Specifically, graph-based fuzzy set similarity approaches facilitate the integration of imprecise data, commonly encountered in real-world web processing, by representing elements of the sets as a bipartite graph and calculating the respective similarity scores using a Maximum Weighted Matching (MWM) algorithm. However, the usage of *optimal* MWM in fuzzy set similarity limits performance due to high computational and memory requirements. In this work, we utilize more efficient *approximate* MWM methods aiming to improve the scalability of set similarity search. Thus, we introduce the approximate matching methods to compute fuzzy set similarity and integrate them into the set join workflow. Our evaluations on web data show performance improvements of 2-19× relative to the state-of-the-art with high accuracy (99% recall) while consuming 23% less memory on average.

5.2 Set Similarity

Set similarity computations are a fundamental operation within web data management and analysis, where resultant sets are generated using a user-defined metric such as Jaccard similarity. Two widely utilized operations that incorporate these computations are set similarity *search* and *join*, both of which aim to resolve similarity from a query set or a collection of sets, respectively. They have extensive uses in data science, data mining, and data management for tasks including data cleaning, entity resolution, and web-based data discovery [28, 116, 153]. Most set similarity based join approaches follow a *filter-verify* framework [92] using a similarity threshold δ. Traditional approaches often rely on the notion of exact similarity between elements within sets. However, exact similarity may not be robust in the presence of noise such as misspellings, format variations, and other types of data alterations,

This chapter is under review as: M. Mandulak, S. M. Ferdous, S. Ghosh, M. Halappanvar, and G. Slota, *Approximate matching for fuzzy set similarity*, in Proceedings of the Nineteenth ACM International Conference on Web Search and Data Mining (WSDM'26), Association for Computing Machinery, 2026, pp. 1–4. Portions of this chapter have previously appeared as: M. Mandulak, S. M. Ferdous, S. Ghosh, M. Halappanvar, and G. Slota, *ApproxJoin: Approximate matching for efficient verification in fuzzy set similarity join*, preprint, arXiv:2507.18891, 2025.

common in text-based web data. *Fuzzy set similarity* joins based on *fuzzy token matching* [138], have been proposed to tackle these issues.

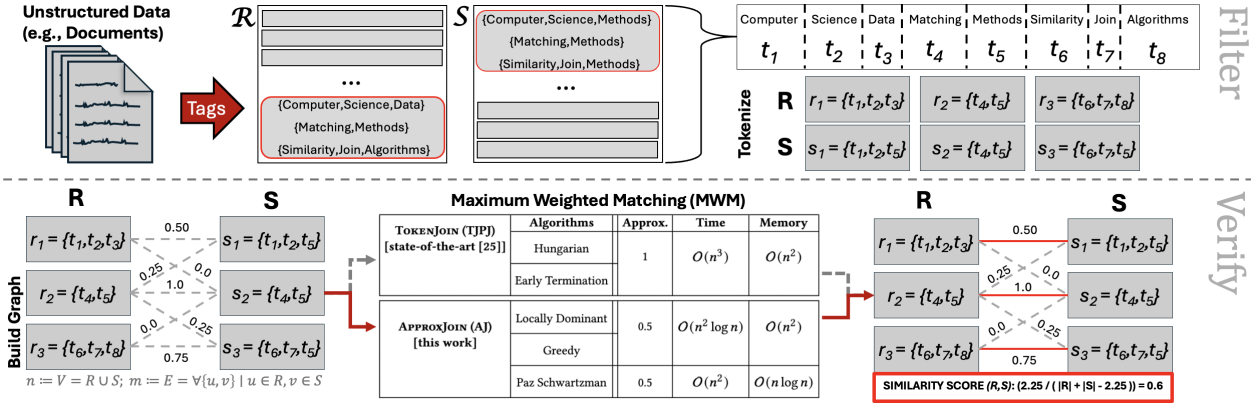

Figure 5.1: Illustration depicting the usage of matching within the fuzzy set similarity join workflow. Text data, such as publication tags, are split into tokens and further into a bipartite graph with similarity-based weights. The maximum weight matching is incorporated into the resultant fuzzy similarity score between the sets R and S.

The most popular fuzzy token matching approaches rely on the *bipartite matching*-based similarity scores [40, 98, 117, 139, 140, 146], where a matching M is a subset of edges in a bipartite graph such that no two edges in M are incident on the same vertex. An example of this workflow is shown in Fig. 5.1, where a maximum weight bipartite matching is computed on the similarity-edge-weighted graph, which is then utilized to calculate the similarity score. However, matching-based similarity is compute and resource intensive, with the matching requiring $O(n^3)$ time and $O(n^2)$ space for two n- element sets. For instance, the best existing fuzzy similarity join method (TokenJoin [146]) requires about 35 hours to compute the complete self-join on the popular KOSARAK dataset.

In this work, we significantly improve the scalability of the join operation by utilizing *approximate* weighted matching in the verification phase. We implement three representative approximate matching algorithms, integrate them within the state-of-the-art TokenJoin framework, and experiment on an extensive range of datasets. Our approximate matching-based join, which we call ApproxJoin, achieves 2-19× performance improvements with high accuracy (0.99 recall) and consumes an average of 23% less memory (detailed in §5.5) than TokenJoin. For example, with the KOSARAK dataset, the runtime decreases from **35 hours** to **5 hours** without compromising accuracy. To the best of our knowledge, this is the first work of its kind to apply approximate matching algorithms to fuzzy set similarity

joins.

5.3 Preliminaries

5.3.1 Fuzzy Set Similarity Join

Our inputs for the fuzzy set similarity join problem are two collections of sets ($\mathcal{R}$ and $\mathcal{S}$), a set similarity function $sim_\phi(R, S)$ (where $R \in \mathcal{R}$ and $S \in \mathcal{S}$), and a user-defined threshold δ. Our goal is to compute all pairs of similar sets, which are defined as: $\mathcal{R} \bowtie_\delta \mathcal{S} = \{(R, S) \in \mathcal{R} \times \mathcal{S} \mid sim_\phi(R, S) \geq \delta\}$ [40, 139, 146]. A set $R \in \mathcal{R}$ contains elements $r \in R$, where each element can be composed of a set of tokens $t \in r$ that are designated as q-grams in string tokenization. In this work, we focus on the *fuzzy set similarity problem* to match set elements under noise, rather than the traditional definition that focuses on exact set overlap to compute $sim_\phi(R, S)$. We provide a visual example in Fig. 5.1.

Given two sets R and S, and a threshold δ, we formulate a bipartite graph $G(V, E, w)$, $V = R \cup S$, between the elements $r \in R$ and $s \in S$ with edges weighted by a chosen similarity measure $\phi(r, s) \in [0, 1]$. We primarily use Jaccard similarity (JAC) here, which is the ratio between set intersection and set union [68]. We find a maximum weight bipartite matching M on G to compute a similarity score between the pair (R, S). The total weight of the matching, $|R \,\widetilde{\cap}_\phi\, S|$, is incorporated into set similarity between R and S as described in [146].

5.3.2 Bipartite Weighted Matching

A bipartite graph $G(V, E, w)$ is defined on a vertex set $V = R \cup S$, where R and S are the two disjoint parts. The edge set E consists of sets $\{u, v\}$, where $u \in R$ and $v \in S$. $w : E \to R_{\geq 0}$ is a non-negative weight function defined on the edges. Throughout the paper, we denote $n \coloneqq |V|$, and $m \coloneqq |E|$. For a vertex v, let $\delta(v)$ be the set of edges incident on v.

A *matching* M in G is a subset of E, where for each edge in M is vertex disjoint, i.e, $e_i \cap e_j = \emptyset$, where $i \neq j$ and $e_i, e_j \in M$. A *perfect matching* is a matching that covers all the vertices. Formally, if M is perfect then $\bigcup\{e \in M\} = V$. Let $w(M)$ be the sum of weights of the edges in the matching, i.e., $w(M) = \sum_{e \in M} w(e)$. The maximum weight bipartite matching (MWM) is to find a matching M_*, whose $w(M_*)$ is the maximum among all possible matchings of G. Note that the graphs generated from similarity joins are complete bipartite graphs, and the resultant maximum weighted matching is always perfect. For a

constant $0 < \alpha < 1$, an *approximate weighted matching* M_a is a matching whose weight is at least α fraction of maximum weight, i.e., $w(M_a) \geq \alpha \cdot w(M_*)$.

5.4 Methodology

In this section, we discuss our primary contribution – the ApproxJoin (AJ) method. We refer the reader to [146], detailing the TokenJoin methodology that we use for candidate generation and filtering (as directly adapted).

5.4.1 Matching Algorithms

The state-of-the-art bipartite matching based fuzzy set join methods, such as [40, 146], employ a primal-dual based method [56] commonly referred to as the Hungarian (HG) algorithm. In this work, we propose incorporating efficient approximate matching into the verification phase of the fuzzy set similarity workflow. We choose three representative approximate maximum weight matching methods: Greedy (GD), Locally Dominant (LD) and Paz and Schwartzman (PS).

Hungarian (HG) Method: The Hungarian (Kuhn-Munkres) method [80] is a primal-dual algorithm for solving MWM problem optimally. We use the HG implementation provided in NetworkX [63] for empirical evaluations. An extension to this method (which we denote as HGE) incorporates early termination conditions for matching. We use the upper bound and lower bound (UBLB) variation as provided by [146].

Greedy (GD) Method: The $\frac{1}{2}$-approximate greedy matching algorithm [5] starts with an empty matching and sorts the edges of the graph in descending order by weight, adding each non-conflicting edge into the matching in order.

Locally Dominant (LD) Method: The $\frac{1}{2}$-approximate Locally Dominant (Pointer Chasing) method from [111], focuses on finding locally dominant edges through mutually pointing neighbors based on highest weights.

Paz and Schwartzman (PS) Method: The state-of-the-art $\frac{1}{2+\varepsilon}$-approximate (for $\varepsilon > 0$) semi-streaming algorithm due to Paz and Schwartzman [108] pushes edges onto a stack by comparing edge weights to those previously processed in the stream. The stack is then post-processed, committing stored edges to the matching while maintaining the matching constraint. We refer to [52, 59] for a more detailed explanation.

5.5 Evaluations

We have extended the open-source Python codebase of a state-of-the-art implementation of fuzzy set similarity join (that uses optimal matching), TokenJoin [146].[6] Our code[7] is available and provided, alongside more detailed descriptions of our theory and experiments. For our experiments, we use the TokenJoin methods for filtering with the Positional and Joint filters (TJPJ) using a similarity threshold of $\delta = 0.7$. All of our experiments are self joins on the respective dataset. In instances where $|\mathcal{R}| \neq 100\%$, a random sample at the given size is generated and used for all methods at that size. Our tests primarily focus on Jaccard similarity (JAC), but note similar trends to those depicted using Normalized Edit similarity (NEDS) [104]. Depicted execution times are averaged over several runs with exclusive allocation on the testbed platform.

Datasets: We experiment on 10 different datasets of web-based text data, whose characteristics are highlighted in Table 5.1. Each token is generated by splitting words into q-grams ($q = 3$) and substituting an integer for each q-gram.

Table 5.1: Experimental datasets and their characteristics.

Datasets	Elements ∈ Set	#Sets	Elements/Set	
			Avg	Max
LIVEJ [100][8]	interests ∈ user	3.1M	36	300
AOL [107][9]	keywords ∈ search	1.8M	3	73
KOSARAK [15][10]	links ∈ user	610K	12	2.5K
ENRON [146][6]	words ∈ email	518K	134	3.2K
DBLP [146][6]	authors ∈ work	500K	13	189
FLICKR [146][6]	words ∈ photo	500K	9	361
GDELT [146][6]	topics ∈ article	500K	19	396
BMS-POS [78][11]	items ∈ sale	320K	6	164
YELP [146][6]	types ∈ business	160K	6	47
MIND [146][6]	words ∈ article	123K	32	357

Platforms: Our baseline results are collected using a single thread on a system with 2TB of DDR4 RAM and dual-socket NUMA AMD EPYC 7742 2.2GHz 64-core processors and 2 threads/core with Ubuntu version 20.04.4 LTS O/S. We use Python version 3.12.9

[6] https://github.com/alexZeakis/TokenJoin/tree/main
[7] https://anonymous.4open.science/r/sasimi-2F82
[8] http://socialnetworks.mpi-sws.org/data-imc2007.html
[9] https://www.cim.mcgill.ca/ dudek/206/Logs/AOL-user-ct-collection/
[10] http://fimi.uantwerpen.be/data/
[11] https://www.kdd.org/kdd-cup/view/kdd-cup-2000

and TJPJ–HG uses NetworkX version 3.4.2.

5.5.1 Baseline Performance

5.5.1.1 Performance Summary

Fig. 5.2 shows the compiled relative performance of all of our test instances, relative to the best method as a difference of execution time. For each test instance, the best algorithm (i.e., the algorithm with the least execution time) is set to zero, and all others are offset from the best one. Approximate methods outperform the optimal methods in every instance. In general, we see the strongest performance results being competitive between AJ–PS and AJ–GD, with a slight performance advantage on average for AJ–PS despite AJ–GD having the best execution times in a majority of cases. We expand on the accuracy benefits of the approaches in the forthcoming section, §5.5.2.

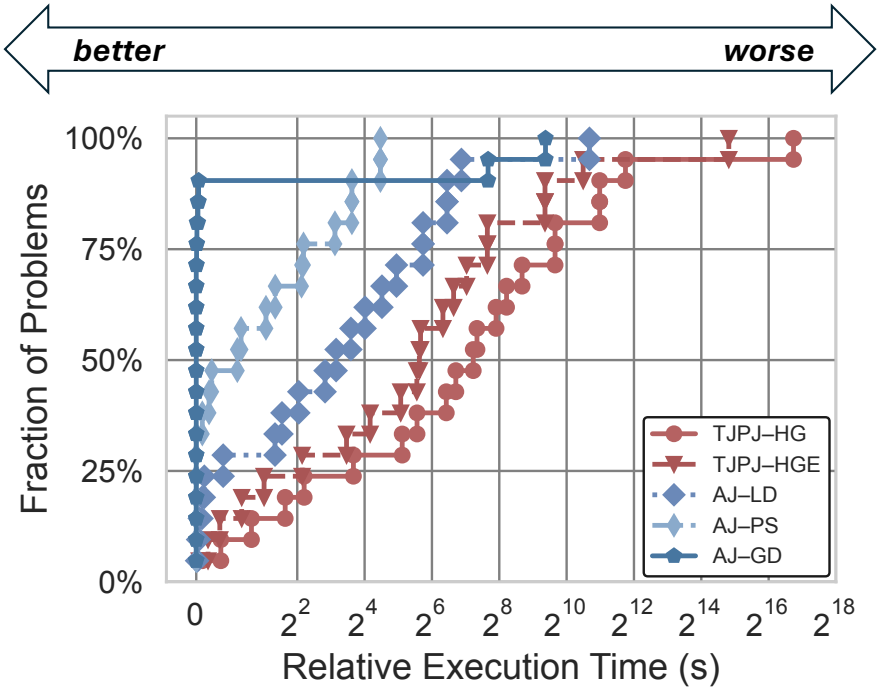

Figure 5.2: A performance profile to summarize the execution time differences between ApproxJoin methods compared to TokenJoin. Approximate methods consistently outperform the optimal.

5.5.1.2 Execution Time

We show the total execution time of the fuzzy set similarity join workflow in Fig. 5.3. In every case, ApproxJoin outperforms TJPJ–HG and TJPJ–EV, with performance improvements being 3.78× compared to TJPJ–HG and 2.18× vs. TJPJ–EV on average. Specifically,

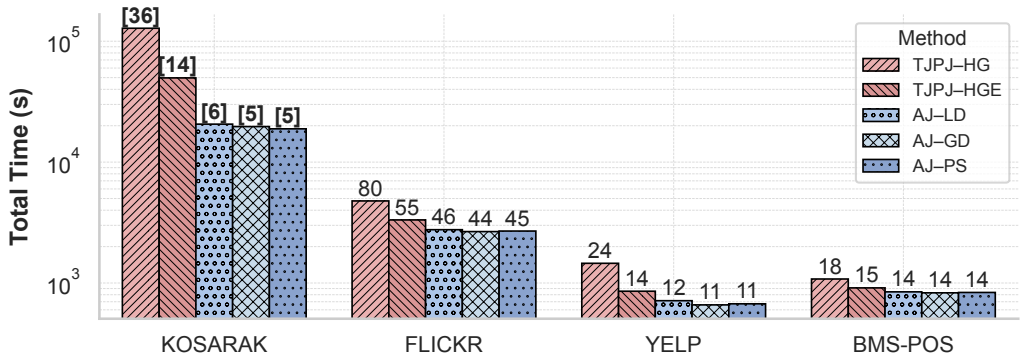

Figure 5.3: JAC total execution time comparison per matching method, $|\mathcal{R}| = 100\%$. The [bold] annotations depict time in hours, while the rest depict time in minutes, rounded up to the nearest whole number.

KOSARAK sees the highest improvements relative to TJPJ–HG: 6.5× with AJ–GD, 6.2× with AJ–LD and 6.7× with AJ–PS, respectively. We attribute this to a high distribution of time spent within the verification phase (94% of the total time). In just verification times, we observe up to 19.1× improvement on KOSARAK using AJ–PS and 4.8× on FLICKR using AJ–GD. In terms of actual execution times, this equates to a difference of more than a day's worth of computing (about 30 hours)! In general, we observe about 2.2× improvement in the (total) execution times across the datasets against TJPJ–HG. On the other hand, compared to TJPJ–EV, there is a 2.5× improvement for KOSARAK and an average 1.4× improvement for the approximation methods considering rest of the datasets. Thus, we see a range of 1.4–6.7× performance improvement with the approximate methods (AJ–LD, AJ–GD and AJ–PS) as compared to optimal TJPJ–HG and TJPJ–EV on total execution time, with 2.0-19.1× improvement to just verification time.

5.5.1.3 Execution Time Scaling

In Fig. 5.4, we perform scaling experiments on a subset of our datasets to assess the performance as the data sizes increase, taking random samples of each dataset at 20% size intervals.

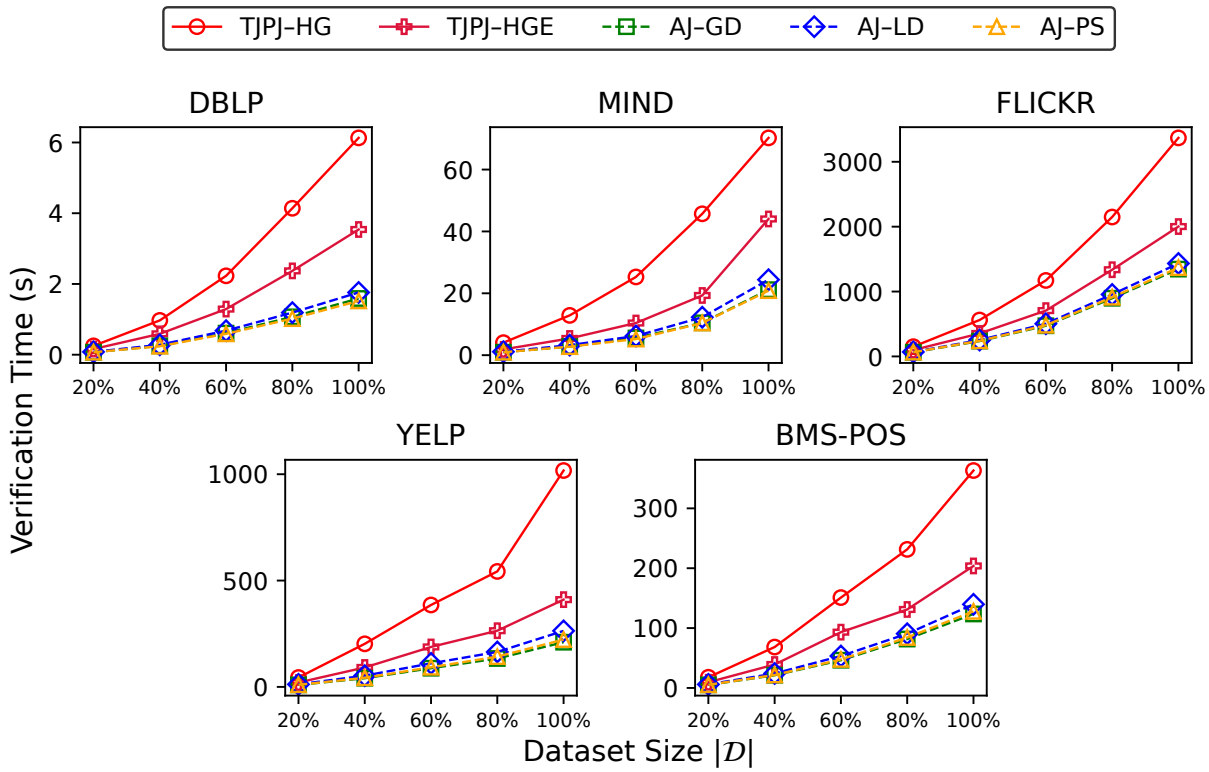

Figure 5.4: JAC verification execution time scaling per set size (lower is better). Average improvement of 3.4× vs. TJPJ–HG and 1.8× vs. TJPJ–EV.

Across every instance, we observe improved scaling trends with reduced execution times for the approximation methods at each size interval. Best performance is achieved at 80%–100% for FLICKR, 4.9× better performance using AJ–GD than TJPJ–HG, and, 2.5× using AJ–PS as compared to TJPJ–EV. On average, the approximation methods shows 3.4× improvement relative to TJPJ–HG and 1.8× improvement against TJPJ–EV. We see AJ–GD outperforming the rest in 16/25 instances, with AJ–PS exhibiting better performance for the remaining 9/25, with an average %-difference of 14% in terms of execution times between methods.

5.5.1.4 Memory Usage Across Program Lifetime

In Fig. 5.5, we compare the lifetime memory consumption (from beginning to end of a particular program run) of the optimal and approximate methods on an instance of $|\mathcal{R}| = 10\%$ for KOSARAK (one of the denser instances). We collect up to 500 distinct samples of instantaneous memory usage for TJPJ–HG, TJPJ–EV and AJ–PS using the `valgrind Massif`

heap profiling tool.[12]

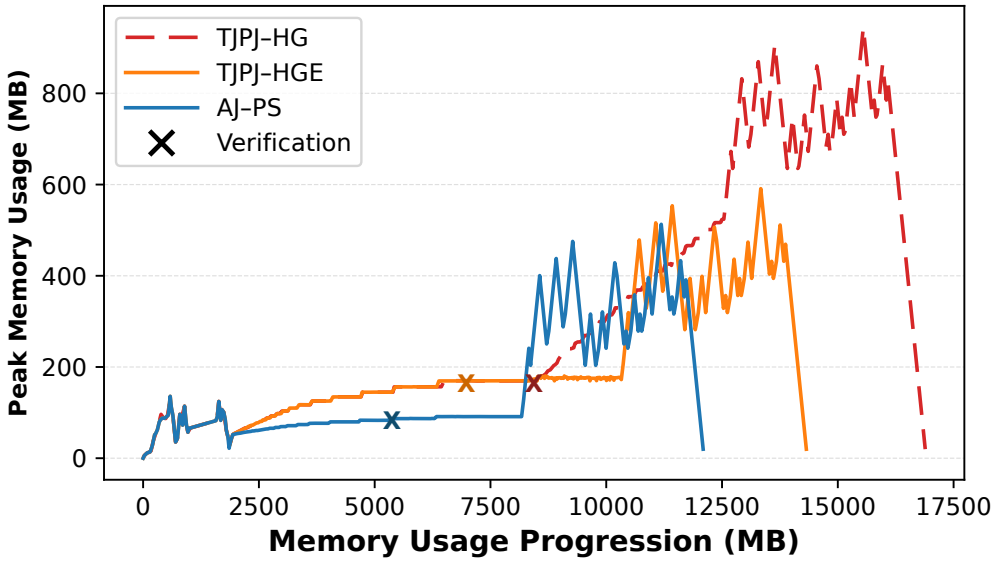

Figure 5.5: Memory usage progression relative to peak memory usage during a run of TJPJ–HG /TJPJ–EV and AJ–PS using KOSARAK $|\mathcal{R}| = 10\%$. Verification center is marked by "×".

Verification happens in the middle portion of the computation (denoted by "x" in Fig. 5.5), where AJ–PS shows under 100MB of peak memory usage while TJPJ–HG /TJPJ–EV can consume up to 40% extra memory in this phase. For KOSARAK (best performing dataset for the approximate methods), TJPJ–HG, TJPJ–EV and AJ–PS depicts peak memory usage of 944MB, 591MB and 512MB, respectively; AJ–PS exhibits 14/45% better memory footprint. We attribute this to the integration of the streaming nature of AJ–PS within verification, allowing us to reduce graph storage.

5.5.2 Approximate Matching Accuracy

Minor variations in the matching weights are expected for the approximate methods; we consider the optimal Hungarian method as our ground truth in assessing the accuracy of ApproxJoin. Since both TJPJ–HG and TJPJ–EV produce the same matching, their qualities are exactly similar. We utilize the resultant set sizes and compare to the default, checking for discrepancies (Δ#Sets) to calculate recall and precision metrics. We define recall and precision relative to our ground truth based on standard true positives (TP), false positives (FP) and false negatives (FN) formulations. Results are collected using the *lowest*

[12]https://valgrind.org/docs/manual/ms-manual.html

values from 3 runs of random samples of $|\mathcal{R}| = 50$K.

5.5.2.1 Matching Weight-based Accuracy

We present the default results in the first half of Table 5.2. We exclude AJ–LD as results were similar to AJ–GD due to the nature of the methods. From these runs, we conclude that the usage of approximate matching yields recall values >0.9 across all instances, with all datasets having at least 0.99 recall besides ENRON. We note that we have excluded the precision as all datasets yielded a precision of 1.0, implying that our datasets are subsets of the dataset yielded by TJPJ–HG. AJ–PS shows the highest recalls among all methods, with geometric mean recall values of **0.9824** and **0.9974** for AJ–GD and AJ–PS, respectively. We attribute the exception of ENRON statistics to its high matching usage (about 87%) alongside a large amount of candidate sets (about 70K) and relatively high dataset density (average 133 elements/set).

Table 5.2: Accuracy assessment subset of our approximate matching based approach, $|\mathcal{D}| = 50$K; Recall/Precision closer or equal to 1.0 is ideal.

Dataset	Recall (Default)				Precision (Upper Bound)			
	Value		Δ#Sets		Value		Δ#Sets	
	GD	PS	GD	PS	GD	PS	GD	PS
LIVEJ	1.0000	1.0000	0	0	0.9998	0.9999	+2	+1
KOSARAK	0.9999	0.9999	-1	-1	0.9999	0.9999	+4	+4
ENRON	0.9293	0.9757	-4515	-593	0.9871	0.9879	+835	+778
FLICKR	0.9994	0.9995	-115	-91	0.9994	0.9995	+114	+109
GDELT	0.9992	0.9996	-937	-481	0.9992	0.9993	+843	+806
MIND	0.9995	0.9999	-38	-8	0.9998	0.9998	+14	+13

5.5.2.2 Upper Bound-based Accuracy

To improve the recall metric using approximate matching, we experiment with trade-offs using upper bounds generated from the approximate matching. Since AJ–GD and AJ–LD are half-approximate solutions, we simply double the matching weights, whereas for the AJ–PS method, we use the sum of ϕ from the primal-dual basis [108]. We show the results in Table 5.2, omitting recall values as every instance yielded a 1.0 recall. This implies that the resultant sets include the default/optimal method's pairings, in addition to new pairings. This inclusion of the extra pairings is represented by the precision metric; the geometric mean

precisions across the approximate matching methods are **0.9978** and **0.9980** for AJ–GD and AJ–PS, respectively.

Based on these assessments, we can consider adapting method's bounds to yield more or less pairings in the resultant dataset. If retaining the pairings generated by an optimal method is essential, then shifting the bounds can induce the necessary pairings without affecting the overall performance (experiments indicate less than 2% performance loss upon expanding the approximation bounds). However, this type of adaptation would not be necessary if the application can withstand minor loss of pairings, i.e., taking a mild hit at the accuracy. Regardless of the choice, we demonstrate significant performance improvements of **1.4**–**6.7**× with high and acceptable accuracy values.

5.6 Concluding Remarks

In this section, we presented ApproxJoin, a fuzzy set similarity join method using approximate maximum weighted matching. We implemented three approximate matching methods within candidate verification: the Greedy (AJ–GD), Locally-Dominant (AJ–LD) and Paz Shwartzman (AJ–PS) methods, and comprehensively compared the execution time performance of our method to the current state-of-the-art, TokenJoin. Across a variety of sparse and dense datasets, ApproxJoin outperforms both the optimal methods of TokenJoin by 2-19× with high accuracy (0.99 recall on average) and reduced memory usage (23% on average). In the next section, we discuss the combination of previous section focuses in an efficient, distributed graph analytics framework including algorithms such as matching.

CHAPTER 6
SCALING DISTRIBUTED GRAPH PROCESSING TO HUNDREDS OF GPUS

6.1 Introduction

In this chapter, we present `HPCGraph-GPU`, highlighting methods for optimized 2D communications for general graph computations on hundreds of GPUs. We extend the usage of 2D distributions to arbitrary and massive-scale graph computations, developing lightweight and sparse communication patterns, active vertex queues, and approaches for more complex reductions and communications. To demonstrate the efficacy of our approach, we implement a handful of the standard benchmark graph algorithms along with more complex routines, including label propagation, maximum weight matching, and pointer jumping. Our efforts approach the theoretical limits of strong and weak scaling for 2D methods, while greatly outperforming the scalability of prior related work. These efforts also offer the first generalized multi-GPU performance results on the largest publicly available dataset, the 128 billion edge 2012 Web Data Commons crawl. On this input, we observe performance from 26-123 billion edges processed per second on 400×V100 GPUs, depending on algorithm complexity.

6.2 Large-Scale Graph Processing

Graph-structured data is pervasive, appearing in a broad range of fields, from math, physics, chemistry, and computer science, among many others. The study and analysis of such graph datasets is correspondingly widespread. However, there are several well-known challenges associated with graph analytics, primarily including the scale and irregularity of graph datasets and the complexity of efficient analytic algorithms to study these large datasets.

Many graph processing engines and techniques have been created in an attempt to address these challenges. Some have offered ease-of-use at the expense of overall performance [118] or are performant but limited in scale to shared-memory [132]. Others have only demonstrated performance for very specialized operations such as breadth-first search

This chapter has previously appeared as: M. Mandulak, and G. Slota, *Scaling distributed graph processing to hundreds of gpus*, in Proceedings of the 54th International Conference on Parallel Processing (ICPP'25), IEEE Computer Society, 2025, pp. 1–10.

(BFS) [3] or relatively simple algorithms like PageRank [73]. A select handful have scaled to large datasets and impressive performance numbers [69, 130, 154]. For very large scale graph analytics or memory-intensive algorithms, distributing the graph structure across multiple nodes or GPUs (we will use the generic term 'rank' when referring to separate memory/compute spaces) is a requirement. However, this introduces several additional challenges, including communication overheads, load balance, and cache inefficiency. These problems are magnified on GPU, where the smaller device memory, lightweight SIMT execution, and higher latency communication (outside of specialized systems such as the DGX [73]) all compound to negatively impact performance [143]. The larger number of GPUs required to process the same dataset, relative to CPU, can also "blow up" the number of messages and communication volume when using common graph distributions.

The original and possibly still most common distribution type for multi-node graph processing assigns a subset of graph vertices and all associated adjacency information for those vertices to a single rank. This is referred to as a "1D distribution" in terms of the adjacency matrix, as each row is fully owned by some rank. This historically has been a reasonable and simple approach, as many graph algorithms tend to perform computations iteratively, updating some vertex (or edge) state information based on the immediate neighborhood. However, this can also limit scalability is two key ways: First, very large degree vertices in graphs with skewed power-law degree distributions are owned by a single rank, resulting in imbalance of computation and communication, especially when strong or weak scaling the number of ranks [79]. Secondly, the overheads for communication in general can become unwieldy. As it is generally necessary for each rank to communicate with every other rank, the number of total messages communicated scales quadratically with the number of ranks.

As such, within the past decade, so-called 1.5D or hybrid distributions [61] have emerged, where selected large degree vertices are shared among multiple ranks, vastly improving load balance for irregular graphs. A more generalized concept is edge-based distributions, where a rank owns some subset of edges for some subset of vertices. Continuing this trend is 2D distributions, where **all** vertices are shared among multiple ranks. In this, a graph's adjacency matrix is broken into rank-owned blocks, where communication occurs in stages among row and column groups of these blocks. Load balance and communication scaling can be vastly improved, and this is the de-facto standard approach for top-performers on

benchmarks such as the Graph500 [3]. 2D-style distributions in particular have a long history in HPC, being employed by dense matrix multiplication methods for decades [65]. Their application to general graph analytics is less widespread [37], mostly being utilized for a few highly-studied benchmark algorithms, such as triangle counting [137] and PageRank [73].

Contributions: We implement a generalized methodology for distributed GPU-based 2D graph processing. We discuss several low-level implementation considerations, including graph data structures, dense/sparse 2D communication patterns, work queues, and workload balance. Our methods exhibit excellent scalability, strong scaling near theoretical limits up to 256 GPUs with graphs small enough to fit in a single device, as well as scaling to 400 GPUs with the largest publicly available graph dataset. We also demonstrate the generalizability of our approach, implementing several complex graph analytics beyond the standard benchmark algorithms.

6.3 Background

Basic Definitions: We consider in simplest terms a graph $G = (V, E)$, where V defines a set of vertices, E defines a set of edges, and $N = |V|, M = |E|$ are the number of vertices and edges, respectively. For computational purposes, this graph is generally represented as an adjacency-based data structure in memory, most often in a compressed sparse row (CSR) or similar format. In such a format, the *adjacencies* (or neighbors) of some vertex v can be directly accessed in some order. The *degree* of v is the number of adjacencies. For the purposes of discussion in this section, we will also consider graph G implicitly represented as an adjacency matrix A. In this square $N \times N$ matrix A, nonzeros at $a_{i,j} \in A$ indicate a single edge or multi-edges between some vertex represented by row identifier i and some other vertex represented by j.

As discussed, this adjacency matrix can be partitioned or distributed across multiple ranks. We will term a rank *owns* a vertex if it is allocated some portion of the vertex's row of the adjacency matrix. For general 2D distributions, multiple ranks own the same vertex. If an owned vertex has an edge to a non-owned vertex, we consider the non-owned vertex as a "ghost". We generally need to maintain information about both owned and ghost vertices to perform most iterative graph computations.

6.3.1 Graph Processing

Generally, graph computations involve iterative updates to state values associated with vertices and/or edges within the given graph input. E.g., BFS will update *parent* or *level* state information as the traversal expands from a root. Commonly, the notion of pulling vs. pushing [11] of updates is considered in this context. Using BFS as an example again, an unvisited vertex can "pull' an update by examining the visitation status of its neighbors and marking itself as visited if it finds a visited neighbor. Conversely, a visited vertex can "push" an update of visitation status to all unvisited neighbors. Similarly, the notion of gather-apply-scatter [61] is used in these contexts, where a vertex gathers neighbors information, applies an update to its state, and scatters this update back to its neighbors.

When considering the distribution of a graph into distributed memory and a typical graph computation, the most important idea for the reader to recognize is that updates to vertex states generally require a vertex's adjacency information as well as the various states of each adjacent vertex. In 1D distributions, all of the 1-hop information is directly accessible for an owned vertex on a single rank. The updated state values of ghosted vertices are what is communicated during iteration computations, ensuring correctness and consistency. This is generally done using a bulk synchronous parallel (BSP) model. As noted above, these ghost updates can either be pushed to the ghost vertex's owner from a non-owning rank or pulled from the owning rank. Either way, this communication is typically done via an all-to-all exchange, requiring $O(p^2)$ messages for p processors.

6.3.2 2D Graph Processing

See Figure 6.1 for an example of a 2×4 2D partitioning of an adjacency matrix into 8 ranks, consisting of row groups and column groups. Note that each row group exclusively owns the same set of vertices and each column group has the same set of ghosts, except for instances along the diagonal where the sets overlap on a single rank. 2D distributions have the added challenge of the fact that the full adjacency and ghost information of a given vertex is not considered to be owned by any single rank. Hence, an additional round of communication is necessary, to ensure consistent state values among all ranks that own that vertex.

If we first consider a pull update and associated communication, it is first necessary to perform some reduction over all vertices owned by the same row group to get consistent

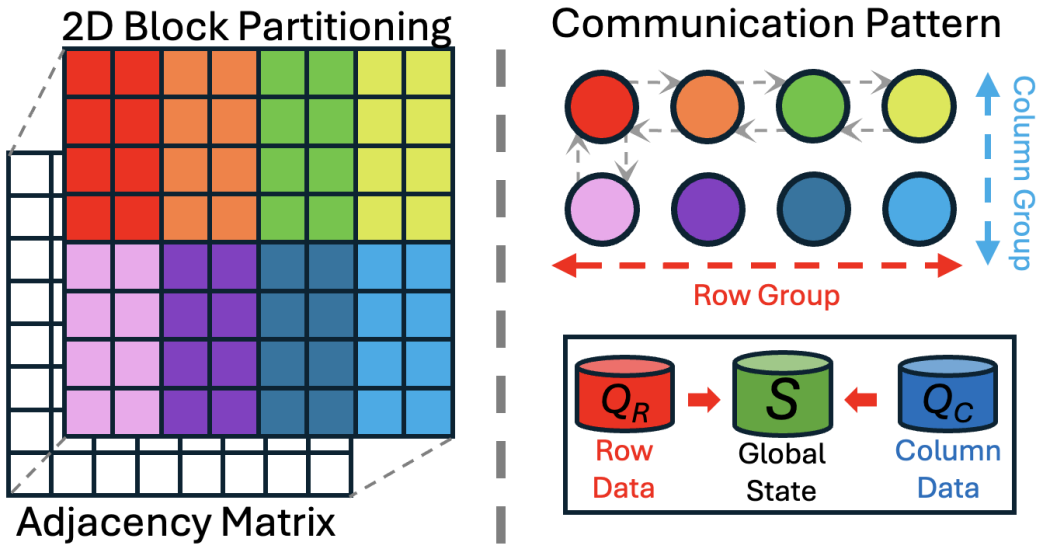

Figure 6.1: 2D block partitioning of an adjacency matrix with 2 row groups and 4 column groups (8 total ranks, designated with color). Communications occur along these row and column groups, with group-based updates (e.g., Q_R and Q_C) being committed to an implicit global state (S).

state values, before broadcasting these values along a column group to ranks that have these vertices as ghosts. Push updates are reversed, where a reduction update is first performed over the column group, before the updates are broadcast or otherwise communicated across a row group. Through these basic methods, all computations possible in a 1D distribution can be equivalently expressed in a 2D distribution.

The benefit of this approach is that each group-wise communication only requires $O(\sqrt{p})$ messages for p total ranks when the row and column group sizes are approximately equal. Since we have $O(\sqrt{p})$ groups in each direction, our total number of messages scales as $O(p)$ instead of $O(p^2)$ as in a 1D all-to-all exchange. However, the total communication volume for 2D distributions can be larger, as a rank can have to communicate up to $O(\frac{N}{\sqrt{p}})$ state updates relative to a maximum of $O(\frac{N}{p}+\frac{M}{p})$ for 1D. Correspondingly, 2D computational loads can also increase as $O(\frac{N}{\sqrt{p}}+\frac{M}{p})$. Because of this scaling behavior, 2D distributions are generally preferred in "weak scaling" scenarios with low communication volumes (e.g., BFS in the Graph500), where a larger N is considered along with a larger p. We will analyze and discuss this as part of our results.

6.3.3 Prior Work

Generalized distributed graph processing has existed for close to two decades now. Early methods utilized I/O intensive operations via MapReduce and similar frameworks [91]. Later, more HPC-based approaches were developed, first optimizing for user-friendly development interfaces [61], but offering low practical scalability [97]. Following those efforts, performance became more of a focus, with several works offering scalability to the largest available public datasets [31, 88, 130]. The most recent work at the very large scale has focused on optimizing partitioning through vertex placement to minimize communication latency and overall cost [57].

While much of the above work has focused on CPUs, GPUs have likewise received considerable attention. Several single GPU and single node multi-GPU frameworks have been developed [69, 71, 141], which address the challenges associated with GPU graph computations mentioned previously. Other work has sought to standardize graph computations via optimized algebraic backends [144], similar to BLAS in the linear algebra domain. While some true distributed multi-node GPU frameworks for general graph computations have been developed [69, 71], none of them have demonstrated fully generalized scalability to the largest current real graph dataset, the 128 billion edge Web Data Commons Crawl.

We know of only one other work besides ours that has utilized 2D distributions for general graph computations on GPU [37, 69]. However, that work utilizes a more generalized communication substrate which allows arbitrary distributions, instead of one specifically optimized for lightweight 2D communications. As demonstrated in our results, this can add overhead that limits performance and scalability in practice.

6.4 Methods: HPCGraph-GPU

To enable general and efficient graph computations in distributed memory, four important considerations are required.

1. **Graph Structure:** A graph structure needs to be compact with minimal overheads to accessing adjacency information or state information.

2. **State Communications:** Vertex (and edge) states need to be communicated in an efficient manner, with volume scaling proportional to the number of state updates.

3. **Vertex Activation:** Many iterative graph computations have a long "tail" of small numbers of updates per iteration, with only a few vertices actively updating. There needs to be a way to track which vertices are relevant to the computation.

4. **Load Balance:** Especially with BSP-style computations, a graph partitioning/distribution should enable computation or communication balance among ranks. Within GPUs, computational load balance should be ensured.

In this section, we detail the distributed structure of our graph data and describe the fundamental communication and computation patterns upon which we build our algorithm implementations. We extrapolate these base communication patterns to complex algorithms, developing more nuanced communication schemes. We call our codebase `HPCGraph-GPU` as an extension of our prior CPU-based `HPCGraph` [130], and we make our code available at the following repository: https://github.com/HPCGraphAnalysis/HPCGraph.

6.4.1 Implementation Details

We implement all of our methods in `C/C++` using `CUDA` for GPU programming and `NCCL` for communications. `NCCL` provided significant performance benefits relative to `CUDA`-aware `MPI`. Graph construction is done on CPU before being transferred to GPU using OpenMP and MPI. We use no other library dependencies, enabling broad portability and ease of compilation.

6.4.2 Graph Representation

For our 2D graph structure, we maintain data relative to our notion of row and column (ghost) ownership within our communication setup. We give an overview of the primary variables used for our representation in Table 6.1, maintained on each rank. Each rank individually tracks the total number of vertices in its row group N_R and its column group N_C, and the starting offsets in terms of global vertex IDs for its row N_{Offset_R} and column N_{Offset_C} vertices. Additionally, it has information about its row group and column group IDs (ID_R, ID_C) and a rank's group-rank within the row group and column group ($Rank_R$, $Rank_C$). Unlike in some matrix methods, we consider a rank as owning only a single block in the distribution.

Locally, vertices are remapped into local vertex IDs, mapping global identifiers from $[0 \dots N)$ to local identifiers in $[0 \dots N_T)$. This is a standard practice in distributed graph

Table 6.1: Primary variables used in our 2D graph structure.

Variable	Description
N	Global number of vertices
M	Global number of edges
Adj	Adjacency list for local CSR
Off	Offsets for local CSR
R	Number of ranks in each row group
C	Number of ranks in each column group
ID_R	Row group ID
ID_C	Column group ID
$Rank_R$	Row group rank
$Rank_C$	Column group rank
N_R	Local number of row group vertices
N_C	Local number of column group vertices
N_T	Total number of unique row+column group vertices
N_{Offset_R}	Starting global vertex ID for row group
N_{Offset_C}	Starting global vertex ID for column group
$Type$	How local vertex IDs are structured
C_{offset_R}	Starting local vertex ID for row group
C_{offset_C}	Starting local vertex ID for column group

structures, done to accelerate local computations and simplify memory accesses. These local IDs are correspondingly used within our local CSR graph, defined by the adjacency array Adj and offsets array Off. As is standard, the adjacencies for a vertex v are listed between $\Big[Adj[Off[v]] \dots Adj[Off[v+1]]\Big)$ and the local degree of vertex v is calculated as $Off[v+1] - Off[v]$. Note that in a 2D data distribution, the local degree is not necessary the actual degree of v, though the actual degree will be equal to the summed local degrees across all ranks in the row group that owns v.

Table 6.2: Definitions for mapping global to local vertex IDs.

$Type$	Definition	Row and column vertex mapping
0	No overlap in row/column GIDs	Row LIDs $= [0 \dots N_R)$ Col LIDs $= [N_R \dots N_R + N_C)$
1	$N_{Offset_R} \leq N_{Offset_C}$	$diff = N_{Offset_C} - N_{Offset_R}$ Row LIDs $= [0 \dots N_R)$ Col LIDs $= [diff \dots diff + N_C)$
2	$N_{Offset_R} > N_{Offset_c}$	$diff = N_{Offset_R} - N_{Offset_C}$ Row LIDs $= [diff \dots diff + N_R)$ Col LIDs $= [0 \dots N_C)$

The way that the vertex IDs are mapped from global→local is dependent on what

we term as the rank's *Type*. The possible *Type* of the mapping is internally denoted as an integer in $[0, 1, 2]$, and the *Type* is defined based on the relative global IDs of row and column vertices. We give their definitions in Table 6.2. This explicit mapping is done for two reasons. Primarily, our communication methods, as described later, communicate via global IDs for consistency. Thus converting global→local IDs is a common operation. A structured definition can do this conversion via simple arithmetic, avoiding a hash table lookup that would otherwise be needed. In addition, this method also simplifies dense communications, as local IDs for row and column vertices are compacted in order, a communication of an array of vertex state values only requires the offset (C_{offset_R} or C_{offset_C}) within the local mapping and the number of vertices in the group, regardless of any row/column vertex overlap.

6.4.3 Communication Patterns

In this section, we detail the various types of communication patterns seen in algorithms built upon 2D communications. We classify these communications as being dense, sparse, or complex, depending on the method and application.

We will consider a generic vertex state algorithm as the basis for our descriptions, shown in Algorithm 6.1. In this generic algorithm, all vertex states are first initialized in some fashion. E.g., PageRanks are initialized to $\frac{1}{N}$ or BFS parent/level information is initialized to `MAX_INT`. For some fixed number of iterations until convergence, these vertex states are updated by looping through all (or some via a queue) vertices needed for the computation. States are generally propagated or updated via edge relations, as indicated in `update`$(S[v], S[u])$.

Algorithm 6.1 Basic State Update Algorithm

```
1: procedure StateAlgorithm(Graph G(V, E))
2:     S ← InitState(G)                          ▷ Initialize state of vertices
3:     for some number of iterations do
4:         UpdateState(G, S)
5: procedure UpdateState(G, S)
6:     for v ∈ V do
7:         for (v, u) ∈ E do
8:                                  ▷ State updates of u (push) and/or v (pull)
9:             update(S[v], S[u])
```

6.4.3.1 Dense Communications

We define dense communications as when *all* vertex state values are communicated along row and column groups, regardless of whether or not they were updated. For simple state update/reduction patterns, this requires no additional processing or handling of data queues or messages. In Algorithm 6.2, we show the typical patterns for a dense push and dense pull communication.

Algorithm 6.2 Basic Dense Communication Pattern

```
1: procedure StateAlgorithm(Local Graph G(V, E))
2:     S ← InitState(G)
3:     for some number of iterations do
4:         UpdateState(G, S)
5:         // Assuming that N_R = N_C
6:         if PUSH
7:             AllReduce(S[C_offset_C], N_C, COL_GROUP_COMM)
8:             Broadcast(S[C_offset_R], N_R, ROW_GROUP_COMM)
9:         else if PULL
10:            AllReduce(S[C_offset_R], N_R, ROW_GROUP_COMM)
11:            Broadcast(S[C_offset_C], N_C, COL_GROUP_COMM)
```

For a push operation, as visualized in Fig. 6.2, we simply perform an AllReduce across each column group on the vertex state S array starting at offset C_{Offset_R} through N_C values in the array. For PageRank, we would be performing a `SUM` reduction over these values. When $R = C$, we follow this with a row group Broadcast starting at $Offset_R$ through N_R state values with the root equal to the row group ID. When $R \neq C$, we used multiple grouped broadcasts via aggregated Group Calls in NCCL. We have found this approach most performant in practice compared to explicit Send/Recv or other gather-based operations. The roots will be all ranks that have an overlap between their row and column vertices, broadcasting up to a maximum of N_C values, each offset from C_{Offset_R} by multiples of N_C in the state values array.

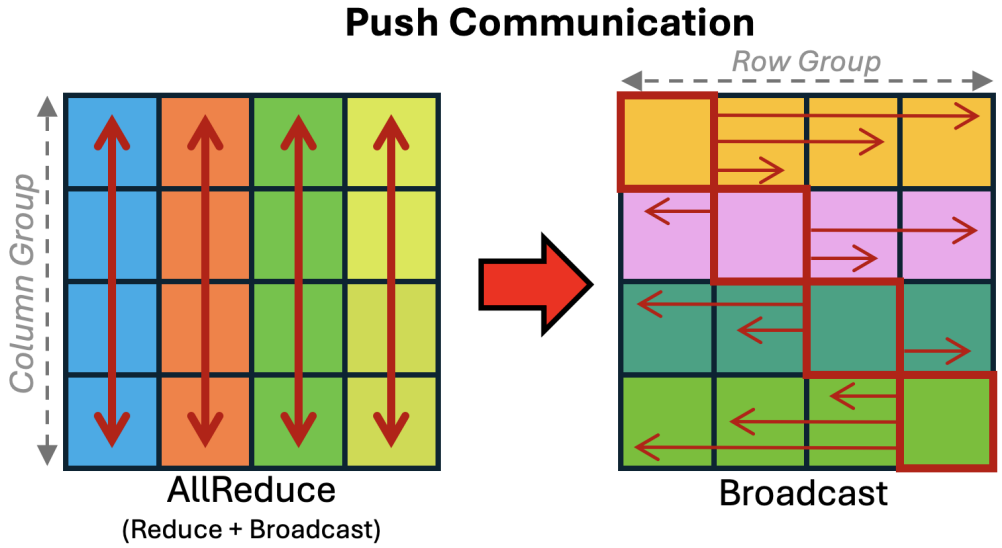

Figure 6.2: "Push" communication pattern (opposite "pull"), described as an AllReduce of the column group followed by a Broadcast of the row group, with 16 total ranks.

A pull operation first reduces over the row group and then broadcasts over the column group. We perform dense pull operations more-or-less the exact same as above, first performing an AllReduce over the row group and then a Broadcast or a group of Broadcasts over the column group. Note that some graph computations such as Label Propagation Community Detection might require a more complex reduction. In the case of Label Propagation, a vertex updates its state to the statistical 'mode' of labels in its neighborhood. In such instances, an approach similar to what we do for sparse communications (discussed next) would be required.

Because of the large communication volume, we reserve the usage of dense communications for instances when the amount being communicated is known to be large. For an algorithm such as PageRank, where vertices tend to update their PageRank values by a small amount on every iteration, dense communications are solely what is used. For many other graph algorithm (e.g., Label Propagation), the number of updates drastically decreases over multiple iterations, so dense communications become wasteful. Hence, we therefore switch to sparse communications after a certain cutoff, defined by the proportion of state updates relative to the vertices in the graph. For our experiments, we make this switch when under $\frac{N}{\texttt{max}(R,C)}$ of vertices in the graph have been updated, to ensure that communication volume is always being saved.

Algorithm 6.3 Basic Sparse Communication Pattern

```
1: procedure StateAlgorithm(Local Graph G(V, E))
2:     S ← InitState(G)
3:     for some number of iterations do
4:         if PUSH
5:             Q ← UpdateState(G, S, Q, q_in)
6:             sbuf ← BuildQueue(G, S, Q, q_in)
7:             rbuf ← AllGatherv(sbuf, COL_GROUP_COMM)
8:             Q ← ReduceQueue(G, S, rbuf)
9:             sbuf ← BuildQueue(G, S, Q, q_in)
10:            rbuf ← Broadcast(sbuf, ROW_GROUP_COMM)
11:            Q ← ReduceQueue(G, S, rbuf)
12:        else if PULL
13:            ...                    ▷ Same procedure with Row/Column comms swapped
```

6.4.3.2 Sparse Communications

Sparse communications trade additional computational overhead in building queues with reduced communication volume by only sending updated values. As we have noted, performant graph codes must utilize sparse communications in some fashion. The general communication pattern for push and pull updates are similar to those above, so we focus specifically on our methodology for efficient queue construction and reductions. See Algorithm 6.3 for an overview.

We first consider the case of push communications. During the `UpdateState()` kernel, we keep track of all unique column vertices with updates pushed to them, and place them in a queue Q. To do this in a thread-safe way, we use an `atomicExch()` on a boolean array q_{in} indexed based on local vertex IDs. A **true** at a given index signifies the vertex is already in the queue. Since a column vertex might have its state updated multiple times, we delay explicitly building the communication queue until launching the subsequent kernel `BuildQueue()`, given in Algorithm 6.4.

Algorithm 6.4 Build Communication Queue

```
1: procedure BuildQueue(G, S, Q, q_in)
2:     sbuf = ∅, n_send = 0
3:     t_idx = unique offset in Q for each thread
4:     v = Q[t_idx]
5:     idx = atomicAdd(n_send, 2)
6:     sbuf[idx] = v                  ▷ Place v and its state into the send queue
7:     sbuf[idx + 1] = S[v]
8:     q_in[v] = false
```

Here, we iterate through all vertices in the queue and build a communication buffer containing {vertex GID, state value} pairs, using the finalized state values for the current iteration. An AllGatherv-style communication (implemented as an NCCL AllGather followed by a group of broadcasts) is then performed to distribute all per rank updated values through the column group. A reduction kernel is then called, given in Algorithm 6.5, which utilizes similar logic to `UpdateState()` in order to update values. This logic can be relatively simple and accomplished via atomics, or it can be more complex logic, as discussed above. The reduction kernel also builds another queue to be used for updates among the row group.

Algorithm 6.5 Reduce Communication Queue

```
1:  procedure ReduceQueue(G, S, rbuf, q_in)
2:      Q = ∅, n_q = 0
3:      t_idx ← unique offset in rbuf for each thread, modulo 2
4:      v = rbuf[t_idx]
5:      val = rbuf[t_idx + 1]
6:      old = S[v]
7:      new = AtomicOp(S[v], val)              ▷ 'Op' can be MIN,ADD, etc
8:                                             ▷ Or can be a more complex routine
9:      if v ∈ Row Group and new ≠ old
10:         if atomicExch(q_in[v], true) == false   ▷ If v is not in queue
11:             idx = atomicAdd(n_q, 1)
12:             Q[idx] = v
```

To perform row communications, we first call a kernel for building a row group communication buffer. This kernel takes in the group of unique row-owned vertices that have been updated in Algorithm 6.5 and places their GID and most recent state value into a communication buffer. This buffer is then communicated via a broadcast or NCCL group of broadcasts (when $R \neq C$) among the row group. Finally, the row group performs a reduction by parsing the received communications and reducing the values as in Algorithm 6.5. The kernel utilized for queue building is essentially the same as with Algorithm 6.4. For the reduction with no further communications, we omit the final if-statement section in Algorithm 6.5.

As with dense communications, pull communications essentially mirror the above. We first build a queue of updates and perform a sparse reduction among the row group. These updates are then broadcast among the column groups for further reduction.

6.4.3.3 Complex Communications

Many graph algorithms do not directly follow the typical state update patterns given above. In fact, we have found that many non-trivial analytic algorithms we have implemented have some additional considerations. We highlight several noteworthy examples below:

Complex Reductions: As we have noted, certain algorithms require non-trivial reductions to compute updates. For such instances, one can solely use the sparse communications with custom reduction operations in `ReduceQueue()`. We implement a maximum weight matching algorithm that uses such a communication pattern.

2.5D Processing: For other algorithms like Label Propagation, the 'complex reduction' is very expensive. Determining the statistical mode of labels for a large neighborhood requires either compute and memory-intensive sorting or compute-intensive hash table construction. We implement this algorithm and opt for the latter using a space-efficient GPU hash-table adapted from prior work [125, 126]. For our approach, each rank in a row group first reduces neighborhood labels into a set of hash tables for all N_R vertices based on their locally-owned edges. Next, each row rank is set to hierarchically own $\frac{N_R}{R}$ unique *local* vertices by block partitioning the vertices owned by the row group. The initial hash tables are exchanged to the owner of the local vertex, which performs the final reduction. These final values are then broadcast back out to the row group, before being subsequently broadcast to the column group in the standard fashion. We refer to this as *2.5D Processing*, with similar examples of "beyond-2D" methods appearing in the literature for algebraic computations [131]. This method can also be used in general for dense communications requiring complex reductions, with the tradeoff of building and communicating a group-wise set of *local* buffers in place of a possibly larger all-gather buffer.

Packet Swapping: In some applications, such as pointer jumping and least common ancestor traversal [16, 17], updates are not solely propagated along edges. For such applications, we use the notion of an information 'packet' that is communicated across row and column groups. We denote this communication pattern as "packet swapping". Generally, the packets contain owner, state, and send direction (e.g., an originating pointer and to which rank to jump), as well as other application-specific data. Communication of these packets can be similarly exchanged pairwise between any ranks via a single set of row and column group communications, as we do with the more structured communications patterns above.

6.4.4 Computation Patterns

6.4.4.1 Vertex Activation

We maintain active vertices by building a row-group-consistent queue on each iteration. For push updates, this approach requires little modification to our discussed kernels. In our `UpdateState()` kernel, we note if a vertex we are placing into Q is row-owned. If so, we also place that vertex into a separate 'active vertices' queue. Likewise, when we are performing the final row-wise reduction, we re-include the final if-statement logic from Algorithm 6.5, this time placing any updated but non-yet-queued vertices into the next active queue.

A vertex queue for pull updates is a bit more complex and expensive. The most important difference is that the active vertices on a subsequent pull iteration are **not** the vertices that updated their state on the current iteration – the active vertices must be set as the *neighbors* of these updated vertices. When we construct and reduce the state update queue along our row-group communication, we track all row vertices that have been updated. We then call a kernel which examines the adjacencies of these vertices and uniquely places them into an active vertex queue. This queue is then shared in a push-style sparse communication across the column groups and then the row groups.

6.4.4.2 Load Balance

One issue with the above approach is that vertices are not ordered within the queue, so GPU load balancing techniques such as sorting vertices by degree within a static CSR are ineffective. Other methods in the literature [99] utilize hierarchical processing methods via implicit or explicit queues, where a vertex's adjacencies are expanded via different methods based on degree. E.g., a low degree's adjacencies are all expanded by a single thread while a large degree vertex is processed by an entire thread block.

Manhattan Collapse: For the relevant adjacency expansion in the cases where we use a queue, we instead utilize the "Local Manhattan Collapse" loop collapse [129], detailed in Algorithm 6.6. This method collapses the nested [For $v \in V$] and [For $(v, u) \in E$] loops by performing a prefix sums over a portion of the active vertex queue assigned to a given GPU thread block. We assign one vertex per thread for each block for simplicity. The block then computes the prefix sums on vertex degrees to get total work and work offsets for each vertex. We then iterate over the total work bounds, assigning unique edges to each thread using the degree offsets and a binary search. These edges are used to perform an algorithm-dependent

computation or reduction. This model of computation is agnostic to our application, and it allows us to effectively assign balanced edge work per thread in any arbitrary queue-based graph algorithms.

Algorithm 6.6 Local Manhattan Collapse per Rank

```
1:  procedure UpdateState(G, S, Q, q_in)
2:      t_id = thread ID
3:      b_id = block ID
4:      v = Q[t_id + b_id × BlockSize]                    ▷ Thread-Queue Assign
5:      work = shared memory array of (BlockSize + 1)
6:      work[t_id + 1] = O[v + 1] − O[v]                  ▷ Degree of v
7:      block_scan(work)                                  ▷ Compute prefix sums
8:      for i = t_id to work[BlockSize] do
9:          j = binary_search(i, work)
10:         u = A[O[v + (i − work[j])]]
11:         update S[v]                                   ▷ Algorithm Dependent
12:         if S[v] was updated
13:             if atomicExch(q_in[v], true) == false
14:                 q ← v                                 ▷ Place v in communication queue
15:         i = i + BlockSize
```

While this can add overhead relative to hierarchical methods, algorithm implementation becomes much simpler and cleaner, computational load balance is almost fully optimized, and the overhead is small and easy to optimize, being near-negligible when the work per edge is high for algorithms such as Label Propagation. For certain graphs, where the maximum degree is close to N, a different approach would likely required, even with a 2D distribution. However, we know of no such graphs that exist at the large scale.

Vertex Distribution: We primarily use a 'striped' distribution for vertex to row group assignment when processing real data. Here, vertex with original GID 0 is assigned to the first row group, vertex 1 is assigned to the second group, with vertex n being assigned to the first row group after vertex $n-1$ is assigned to the n^{th} row group, continuing through all N vertices. We found that such a distribution offers comparable load balance to a random distribution without having varying group sizes, and it maintains some degree of memory locality of the original graph (whose vertices are often ordered using BFS or DFS traversals). Explicit 2D graph partitioning for smaller-scale graphs and matrices has been considered for decades [20, 24], though existing software lacks practical scalability to the massive inputs we consider. Future work might investigate communication-optimizing methods based on

hardware network topology and similar methods [57].

6.5 Implemented Algorithms

We implement a handful of algorithms to test our methods, listed in Table 6.3. We consider BFS, PageRank and connected components as our primary 'benchmark' algorithms, with the rest highlighting the discussed complex communication patterns. The most important details for our experimental section are given in the table. We run PR and LP for fixed iteration counts, which is often standard for benchmarking purposes.

Table 6.3: Algorithms, abbreviations, and experimental notes.

Algorithm Name	Abbr.	Notes
Breadth-first Search	BFS	Standard hybrid method.
PageRank	PR	We run for 20 iterations.
Connected Components	CC	We run until convergence.
Approx. Max Weight Matching	MWM	We run until convergence.
Label Propagation	LP	We run for 20 iterations.
Pointer Jumping	PJ	We run until convergence.

Breadth-First Search: We implement a standard push/pull-optimized BFS using sparse communications and static parameters from the original Beamer et. al paper [8].

PageRank: We implement the standard PageRank algorithm as a pull-based vertex state program with dense communications, instead of an optimized linear algebraic routine. We will compare against an optimized routine in our results.

Connected Components: We implement the color propagation-based distributed algorithm for (weakly) connected components decompositions. We use this to study the effects of the described optimizations. We opt for this algorithm for connected components in place of a pointer-jumping based routine, as its simplicity and typical 'graph algorithmic' pattern enables us to generalize results to a broad class of algorithms. We implement both push and pull variants with both dense and sparse communications and with and without active vertex queues for comparison.

Label Propagation: We implement our previously-described '2.5D' variation of label propagation community detection as a pull update with sparse communications and vertex queues.

Approximate Maximum Weight Matching: We implement a distributed version of the Locally-Dominant approximate maximum weight matching (MWM) algorithm [111]. We

define a MWM as follows: given our graph with respective edge weights w, a *matching* is a subset of edges, $M \subseteq E$, where every vertex of G has at most one endpoint in M. A MWM is a matching of maximum total weight among all possible matching. This method requires each vertex to scan neighbors and set a pointer along the highest weight, non-matching edge. Once set, mutually-pointing pairs are committed to the matching. Within the reduction pattern, we maintain a queue of unmatched vertices for pointers each iteration. All vertices in the queue consider reduced edge information among row/column groups with pointer updates marked and reduced. Mutual checks occur as a global state reduction for a finalized matching.

Pointer Jumping: We implement a pointer jumping algorithm that functions as a root-finding mechanism of a subtrees within a forest on our graph, similar to an approach used for connectivity. First, we instantiate a pointer for each vertex along an owned edge within a graph, creating a forest. We then iterate by communicating pointers up each subtree. Pointers are treated as packets of information that include the relevant jump, original global source, and destination IDs. These are communicated along either the row/column group, depending on the ownership of the source vertex, and correspondingly updated.

Table 6.4: Graph input datasets. The RMAT inputs are scale-XX with standard Graph500 parameters (*edgefactor* $= 16$, $A = 0.57$, $B = 0.19$, $C = 0.19$). The RAND inputs have the same size and order as the RMAT graphs but are generated with an Erdös-Renyí $G(n, m)$ process.

Name	Abbr.	Vertices	Edges
twitter-2010	TW	41M	1.4B
com-friendster	FR	65M	1.8B
web-ClueWeb09	CW	1.7B	7.9B
gsh-2015	GSH	988M	33B
WDC12	WDC	3.5B	128B
RMATXX	RMAT	2^{24}–2^{32}	2^{28}–2^{36}
RANDXX	RAND	2^{24}–2^{32}	2^{28}–2^{36}

6.6 Results

We run two primary sets of experiments. First, we consider scaling experiments using standard benchmark algorithms (PR, CC, BFS) and a selection of standard benchmark graphs, given in Table 6.4. We consider graphs as undirected for consistency across algorithms, effectively symmetrizing the adjacency matrix. Our second set of experiments

considers our complex algorithms. We put particular focus on the Web Data Commons 2012 web crawl, which as of May 2025 is still the largest publicly available real graph.

Our primary test system is AiMOS at RPI, which contains nodes with 2× IBM Power-9 CPUs and 512 GB DRAM, 6× NVIDIA 32 GB V100 GPUs, and an EDR Infiniband network. On node, a group of 3 GPUs per each CPU is interconnected with NVLink. Communications on node across GPU groups and communications across the network required movement though the CPU, which was likely our largest bottleneck for scaling. We ran our scaling experiments on up to 400 GPUs with the WDC12 graph, but limited ourselves to 256 GPUs for other inputs due to high turnaround times for larger allocations. Runs larger than 400 GPUs were not possible due to system constraints. We compiled with CUDA V11.8.89, NCCL 2.10.3, and OpenMPI 3.1.5. For smaller-scale comparisons to code we could not run on AiMOS, we additionally used a workstation with 4×A100 GPUs, referred to as `zepy`.

6.6.1 Strong Scaling

Given in Fig. 6.3 are strong scaling experiments using BFS, PR and CC. The top plot gives total execution times for 1-256 ranks (when possible) on our smaller inputs, reported as the maximum time over all ranks. We observe scaling on all inputs up to 256 GPUs. Given that TW and FR both fully fit within the memory of a single V100 GPU, we consider these results to be extremely notable. The middle plots of Fig. 6.3 plot just the communication times. We note that both communication and computation times both continue to show reductions as we scale, with the exception of the smallest inputs with PageRank. However, we note communication times begin to dominate for all graphs and all applications at the largest scale. The jump between 4 and 16 ranks in some tests is due to all 4 rank runs being on a single node and not using the network.

The bottom plot gives our speedups from 16 ranks, the smallest number of ranks for which we were able to obtain results for all graph and algorithm combinations. As discussed, a downside of a 2D distribution is that communication volume and work per rank can scale by $O(\frac{N}{\sqrt{p}})$. Thus, when strong scaling a bandwidth-intensive application, one might expect speedups to be theoretically bounded at the extreme end as $p \to \infty$ by a factor of $\sqrt{p}$, which we also plot for comparison. We observe most speedup values from $16 \to 256$ GPUs being in the near-optimal range of 3-4× $\approx \sqrt{\frac{256}{16}}$. However, as noted, the primary benefit of a 2D distribution is with processing larger inputs, represented by our weak scaling experiments.

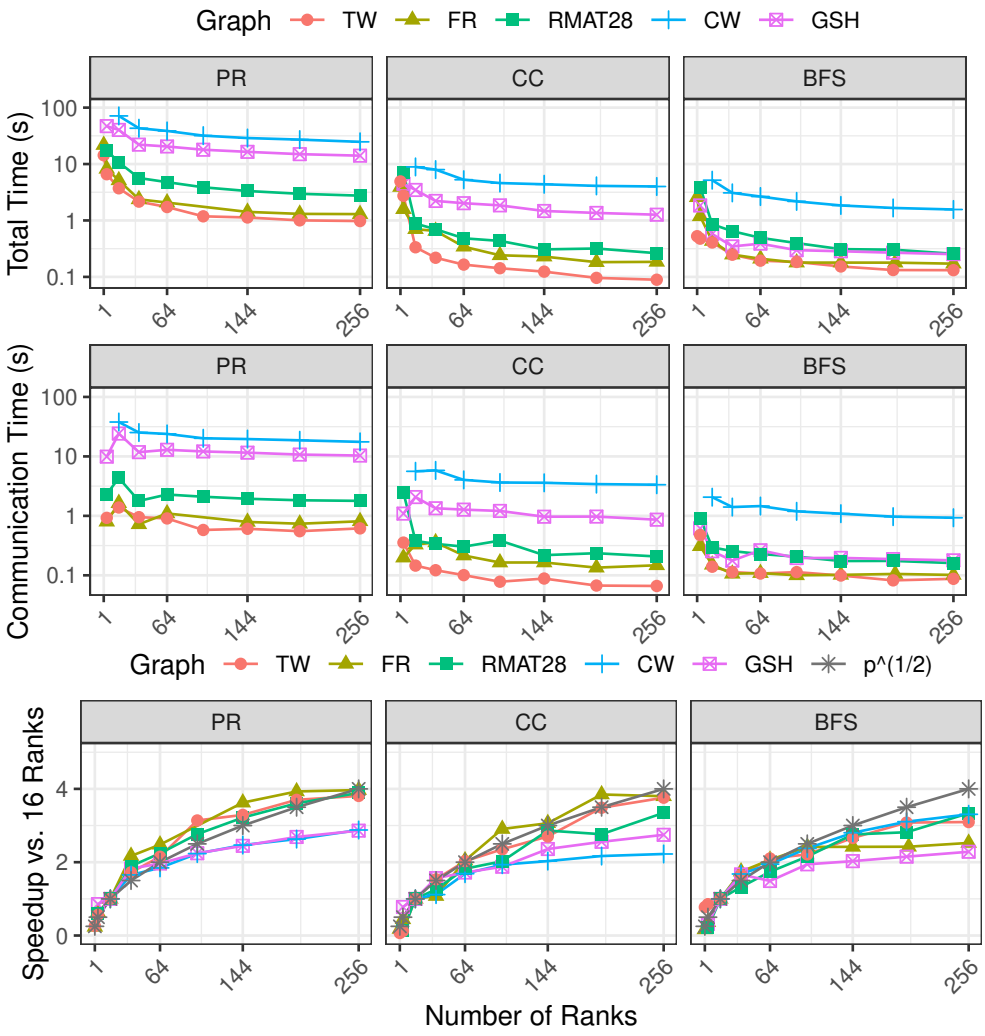

Figure 6.3: Strong scaling with total times (top), communication times (middle), and speedups (bottom) from 1 to 256 ranks on our benchmark tests.

6.6.2 Weak Scaling

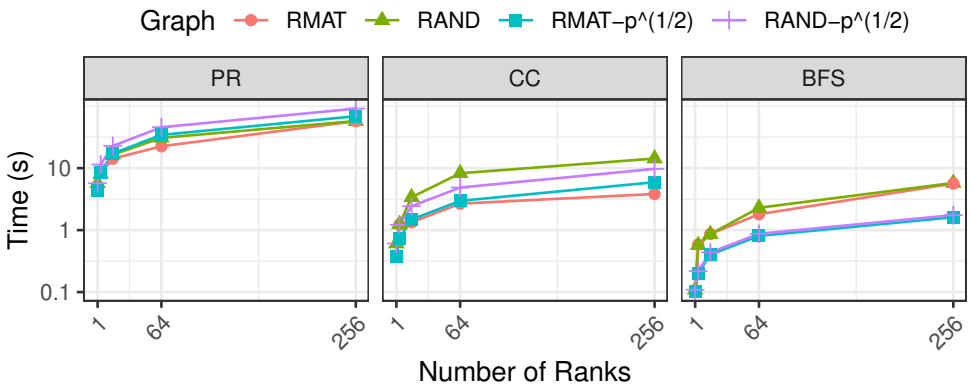

Figure 6.4: Weak scaling on RMAT and Random Graphs.

In Fig. 6.5, we plot results from our weak scaling tests on RMAT and Erdös-Renyí random graphs along with times from 1 rank scaled by a $\sqrt{p}$ factor. We use graphs generated to have 2^{24} vertices and 2^{28} edges per rank. We observe that the timings follow the $\sqrt{p}$ factor discussed before, with all timings from all experiments just under doubling for every $4\times$ increase in rank count, indicating that our methods are approaching the theoretical limits of efficiency. The exception is BFS, where the single GPU runs are correspondingly faster due to the relatively higher communication cost for the algorithm. Overall, these results are incredibly promising, and we hope to scale out even farther, pending access to larger test systems.

6.6.3 WDC Results

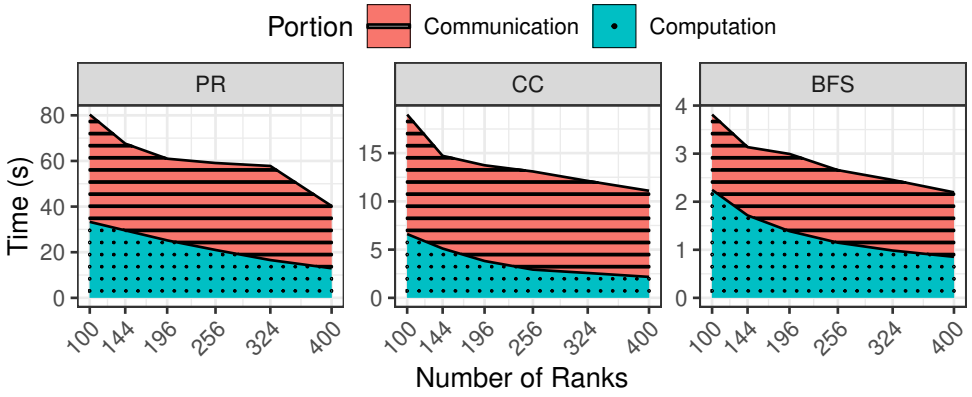

Figure 6.5: Computation and communication on WDC from 100 to 400 ranks.

Fig. 6.3 plots timing results from the same benchmark algorithms running on WDC from 100 to 400 ranks. Here, we consider the total time as the proportion of time spent in communication and computation phases. The maximum time over all ranks for each is reported. Again, we note that overall times scale nicely from 100 to 400 ranks, achieving speedups of about $2\times$ for all algorithms, matching the expected $O(\sqrt{p})$ factor again. We observe that computation and communication also scales for all algorithms, though the speedup is less for communication, as expected.

6.6.4 Sparse Communications and Vertex Queues

Next, we use the CC algorithm to demonstrate the effects of our implemented communication and workload strategies. See Fig. 6.6, where we compare pull updates with dense communications and no vertex queue (Base), fully using sparse communications (+SP),

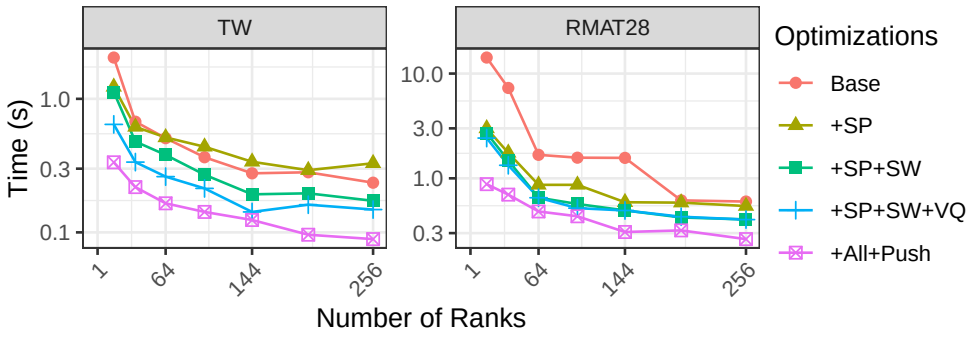

Figure 6.6: Effect of optimizations on Color Propagation CC performance.

switching from dense to sparse communications (+SP+SW), adding in a vertex queue (+SP+SW+VQ), and then utilizing push updates with all strategies (+All+Push). We note the differences are significant, equating to an order of magnitude. While we can only show a couple inputs for space, we note that we observe these relative speedups consistently across the other inputs and that these speedups are also consistent with the other implemented algorithms that use them in part or in full (BFS, LP, MWM).

6.6.5 Non-square Distributions

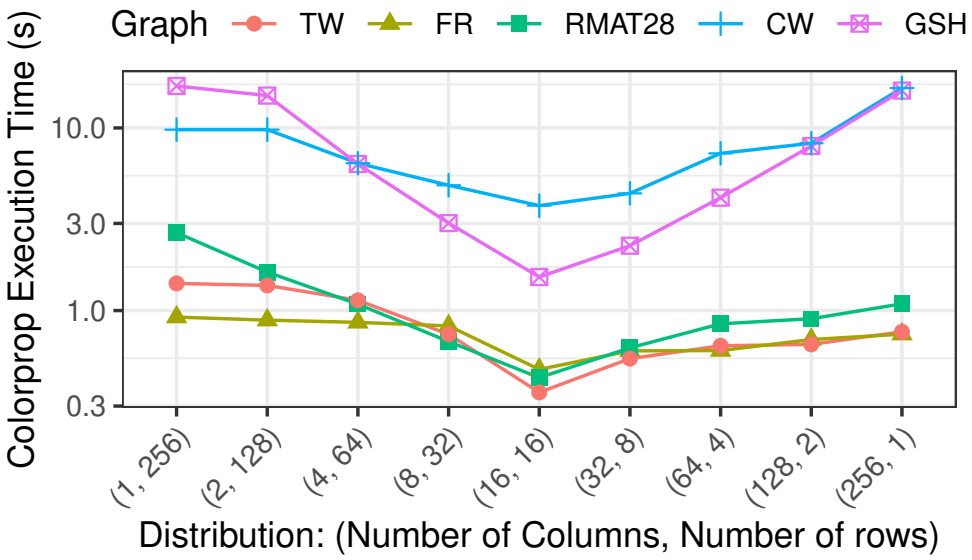

Figure 6.7: Non-square results with CC by varying C, R with 256 total ranks.

It is established that a 'square' 2D distribution (where $R = C$) minimizes communication. While the bulk of our tests use square distributions, we also study the effects of non-square distributions on performance. We plot results in Fig. 6.7 of running CC on 256 ranks while varying R and C among all possible combinations. We first note the obvious

in that a 16×16 distribution is optimal. However, performance does not significantly degrade right in the vicinity, especially for larger inputs, and particularly when decreasing the number columns. We observe about a 1.4× slowdown from (32,8) to (16,16). Note that CC is a push implementation, so the more expensive reduction communication is along the column group. These results suggest that performance scaling is still possible, even without a nice square number of ranks, and that one should bias towards minimizing the reduction direction of the implemented algorithm.

6.6.6 Complex Algorithms

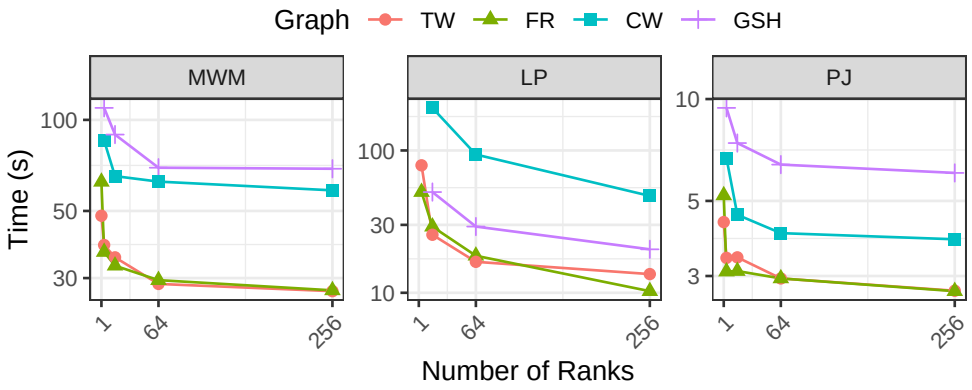

Figure 6.8: MWM (left), LP (middle), and PJ (right) strong scaling from 1 to 256 ranks on the real inputs.

We further collect some base strong scaling results using the implemented complex algorithms of MWM, LP, and PJ. We display these results in Fig. 6.8. Overall, we observe strong scaling to 256 ranks for almost all methods and inputs. The overall execution times for MWM and PJ tend to plateau more significantly compared to our other algorithms, due to problem complexity and necessary communications to synchronize states. LP execution times exhibit better scaling trends under the 2.5D approach, having proportionally less communication cost and more computation. Nevertheless, the performance improvements are consistent across inputs, demonstrating the applicability of 2D distributions for algorithms with complex communication requirements.

6.6.7 Comparisons to Prior Art

We consider two direct comparisons to existing distributed GPU graph frameworks. Our first comparison is to Gluon-GPU [37, 69], part of the Galois library. While this code offers the fastest generalized distributed GPU performance we have found in the literature,

we note the code has not been maintained for close to 6 years and has several outdated dependencies, requiring significant effort to run on our test systems. We ran comparisons on TW, FR, and RMAT28 using our three benchmark algorithms (BFS, CC, PR) on AiMOS. We were unable to successfully run GSH or CW due to memory allocation failures. We use their 2D CVC (cartesian vertex cut) partitioning with the push variants for BFS and CC and the pull variant for PR along with no thread binding and asynchronous communications, as suggested by the authors for best performance. We give our comparison in Figure 6.9.

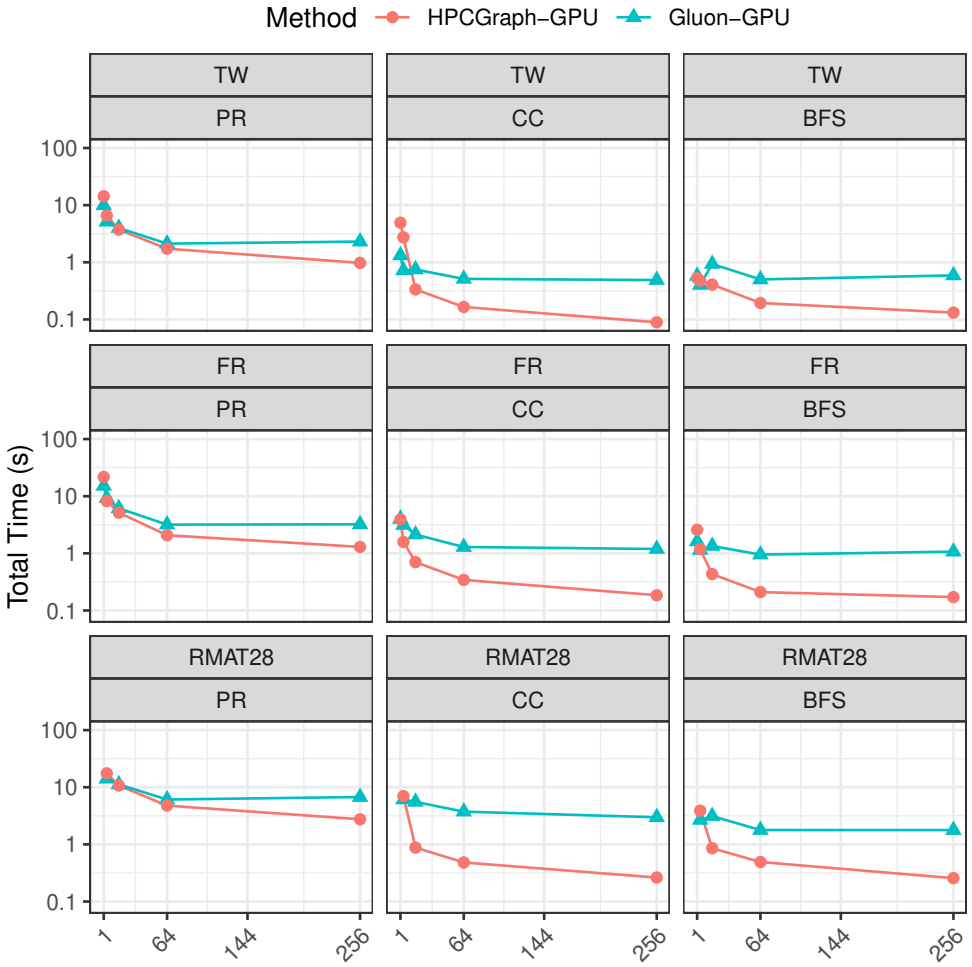

Figure 6.9: Our method and Gluon-GPU running from 1 to 256 ranks on TW (top), FR (middle), and RMAT28 (bottom) while processing PR (left), CC (middle), and BFS (right).

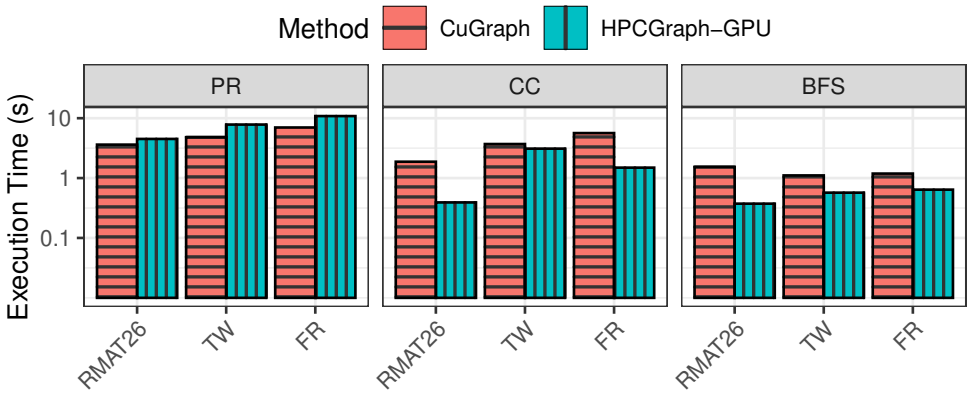

Figure 6.10: CuGraph comparison on PR, CC, and BFS.

We note that performance of our method approximately matches Gluon-GPU on single rank and single node runs (ranks of 1 and 4), but Gluon-GPU suffers significant relative performance degradation when communicating across the network. Gluon-GPU does not scale at all past 64 ranks on the majority of tests. We note that 'Gluon', the communication layer, was built for general-purpose communications, and the 2D CVC distribution is built on top of it. This adds overhead relative to our optimized 2D communication methods, and it is likely the reason for the observed scaling results.

We also compare to the latest CuGraph release (v25.04), which utilizes a 2D distribution methodology for PageRank [73]. However, as we were unable to get it successfully running on AiMOS after extensive efforts, we used 4×A100s on `zepy` for this comparison. We plot a comparison for our benchmark algorithms in Fig. 6.10. We use RMAT26 as RMAT28 (and the other larger graphs) did not run on CuGraph on this system. We ran 20 iterations of PR for both and note an average slowdown of 1.47× for our code, likely due the fact that CuGraph uses optimized linear algebra routines, instead of a general purpose graph computational model, and computation times dominate at the small single-node scale. We otherwise observe average speedups of 3.25× on CC and 2.64× on BFS for our code.

6.7 Concluding Remarks

In this section, we introduced a 2D distributed GPU-based graph processing methodology for large scale inputs. We detailed its implementation and showed its efficacy on a wide range of benchmark and complex graph analytic algorithms. We leveraged a variety of dense and sparse communication patterns, vertex queues, and workload balance techniques, and we showed scaling near the theoretical limits of 2D methods on up to 400 GPUs, using graphs

with up to billions of vertices and hundreds of billions of edges. This work significantly outperforms existing methods across a series of comparative tests.

CHAPTER 7
CONCLUDING REMARKS

7.1 Thesis Contributions

This thesis sought to push the boundaries of performance for a variety of graph analysis methods, ranging from vertex ordering and general analytics to weighted matching and its relevant applications. The main contributions of the work contained within this thesis are summarized as follows.

In the realm of vertex ordering, our parallel degree-based refinement algorithm improved upon the performance of existing heuristic methods by up to 15× on popular analysis methods such as PageRank, Louvain and graph connectivity. For general graph statistics on multiple GPUs using C++26 asynchronous chaining workflows, our method improved upon the baseline implementation by up to 55× with a 2× improvement upon alternative streaming-based implementations.

For approximate maximum weighted matching, our multi-GPU locally dominant implementation improved upon CPU multithreaded and single GPU comparison methods by 2-45× while showing results for billion edge graphs. The direct application of these approximate maximum weighted matching methods to set similarity outperformed the state-of-the-art set similarity join method by 2-19× while maintaining high accuracy (0.99 recall) and reducing memory usage by approximately 23%.

Extending general analytics to a 2D framework, our work showed near theoretical scaling for popular baseline graph analytics such as PageRank, BFS and connected components. Our methods also outperformed alternative distributed multi-GPU graph processing frameworks at scale while showing flexibility to complex algorithms such as weighted matching and pointer jumping. This work further scales up to 400 GPUs and runs the largest available real graph of Web Data Commons 2012.

Portions of this chapter previously appeared as: M. Mandulak, R. Hu, and G. Slota, *Explicit ordering refinement for accelerating irregular graph analysis*, in 2022 IEEE High Performance Extreme Computing Conference (HPEC), IEEE Computer Society, 2022, pp. 1–8.

7.2 Future Works

Alongside the push of performance bounds within a variety of graph analysis domains, this thesis serves as a primary point of countless future works in the field. We discuss a few significant avenues stemming from included works.

7.2.1 Vertex Ordering

Developed as introductory work towards an optimization-based approach to vertex ordering for efficient graph analysis, a number of directions for progression exist. The degree-based refinement method can be further evaluated through testing on graphs on a similar scale to that of ClueWeb09 or graph sets with diverse class distribution. Similarly, a wider variety of graph analytic algorithms could see improvements from explicit refinement methods, especially under parallel models differing from that of shared-memory. We plan on evaluating the degree-based refinement method using GPU-implemented analytic algorithms and performing a comparison to current results.

Furthermore, we plan to develop alternate explicit refinement methods with similarities drawn from common graph partitioning methods for improved parallel speedup. This naturally leads to the consideration of explicit subgraph optimization in vertex ordering. Such a method would consider optimal edge cuts in partitioning for explicit ordering refinement and the preservation of global ordering schemes within subgraphs. Spectral partitioning methods and multi-level methods similar to that of graph coarsening could also show improvement in application within explicit refinement methods.

Methods within the field of optimization can also be applied to a graph-focused model under an explicit mapping of vertex labels. Similar attempts were made using linear and non-linear programming models relative to each metric, but were ultimately runtime infeasible in our specific application. Further work towards this would allow for the efficient application of solver models to subgraphs specifically partitioned for ordering optimization, providing a relative optimal ordering.

7.2.2 `C++26` Asynchronous Chaining

Using the proposed *Senders* framework from the lens of algorithm composability allows for the implementation of countless graph analytics using asynchronous chaining. This allows for the baseline performance improvements of multi-GPU systems without the complexities

of device code development. In the instances of complex algorithms that do not necessarily fit within standardized library operations (reduce, scan, etc.), the framework allows for the integration of device functionalities between asynchronous chain calls. Thus, any algorithm can fit within this generalized framework given the proper design considerations. Similarly, performance improvements will only expand as these technologies further develop, allowing such a framework to be powerful in usability and flexible in application. The extension to alternative hardware usages apart from our tests on NVIDIA hardware also serves as a point of comparison.

7.2.3 Approximate Weighted Matching

This thesis primarily targeted parallel implementations and applications of half-approximate methods, which can be furthered through methods with approximation guarantees greater than half-approximate. These algorithms are notably complex and reliant on augmenting path-based methods, requiring further synchronization and dependency overlaps within iteration. Despite their complexity, there are a number of applications that yield higher-quality results from higher-quality matching, such as cardinality-based problems and data cleaning methods apart from set similarity. The intersection between these parallel implementations and applications also serves as a point of future work, as approximate methods above half-approximate are largely unexplored in parallel settings.

7.2.4 2D Graph Processing

Building upon the *HPCGraph-GPU* framework proposed, there are a number of extensions to both the base and complex algorithm sets shown in the relevant work. Further optimizations exist within the communication patterns of complex methods to yield improved scalability, which naturally extends to new inclusions within the set, such as community detection methods, partitioning methods and others. The mitigation of graph overheads in the initial distribution as well can occur through direct to GPU graph computations, further reducing the initial communication overheads.

7.2.5 Developing Graph Standards

Considering developing works in the field of scalable graph analytics, there exist notable proposals of `C++` standardized graph libraries [112] as a collective framework for efficient

graph algorithms. Considering the wide range of per-instance graph libraries targeting specific algorithms or hardware setups [35, 37, 43, 51, 135], a consolidated standard for graph algorithms towards future works is of utmost benefit to the field. While this may be difficult in instances where optimizations rely on specialized data structures or hardware, building from a generalized standard sets a baseline for performance while increasing the usability of graph algorithms across domains. Regarding the work presented in this thesis, we specifically include frameworks and solutions that can be easily integrated into the `C++` standard, fitting the adaptation of generic graph structures using data views. This especially works well with the proposed `C++26` *senders* framework, as data abstractions through device-aware views are already compliant with developing `C++` standard operations, with need only to fit within pre-defined graph constructs. Thus, a major point of future work in this space is the ease of integration of proposed graph algorithms within the outlined `C++` standard, paving the way for future developments in standardized parallelism for `C++`-based graph algorithms.

REFERENCES

[1] R. K. Ahuja, T. L. Magnanti, and J. B. Orlin, *Network Flows: Theory, Algorithms, and Applications*, Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1993.

[2] A. Apostolico and G. Drovandi, *Graph compression by bfs*, Algorithms, 2 (2009), pp. 1031–1044.

[3] J. Arai, M. Nakao, Y. Inoue, K. Teranishi, K. Ueno, K. Yamamura, M. Sato, and K. Fujisawa, *Doubling graph traversal efficiency to 198 terateps on the supercomputer fugaku*, in 2024 International Conference for High Performance Computing, Networking, Storage and Analysis (SC '24), IEEE Computer Society, 2024, pp. 1616–1629.

[4] J. Arai, H. Shiokawa, T. Yamamuro, M. Onizuka, and S. Iwamura, *Rabbit order: Just-in-time parallel reordering for fast graph analysis*, in Proceedings of the 2016 IEEE International Parallel and Distributed Processing Symposium (IPDPS), IEEE Computer Society, 2016, pp. 22–31.

[5] D. Avis, *A survey of heuristics for the weighted matching problem*, Networks, 13 (1983), pp. 475–493.

[6] A. Azad, A. Buluç, X. S. Li, X. Wang, and J. Langguth, *A distributed-memory algorithm for computing a heavy-weight perfect matching on bipartite graphs*, J. Sci. Comput., 42 (2020), pp. C143–C168.

[7] L. Backstrom, D. Huttenlocher, J. Kleinberg, and X. Lan, *Group formation in large social networks: Membership, growth, and evolution*, in Proceedings of the 12th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining (KDD '06), Association for Computing Machinery, 2006, p. 44–54.

[8] S. Beamer, K. Asanovic, and D. Patterson, *Direction-optimizing breadth-first search*, in Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis (SC '12), ACM and IEEE Computer Society, 2012, pp. 1–10.

[9] S. Belongie, J. Malik, and J. Puzicha, *Shape matching and object recognition using shape contexts*, IEEE Trans. Pattern Anal. Mach. Intell., 24 (2002), pp. 509–522.

[10] M. Bernaschi, A. Celestini, F. Vella, and P. D'Ambra, *A multi-gpu aggregation-based amg preconditioner for iterative linear solvers*, IEEE Trans. Parallel Distrib. Syst., 34 (2023), p. 2365–2376.

[11] M. Besta, M. Podstawski, L. Groner, E. Solomonik, and T. Hoefler, *To push or to pull: On reducing communication and synchronization in graph computations*, in Proceedings of the 26th International Symposium on High-Performance Parallel and Distributed Computing, ACM, 2017, pp. 93–104.

[12] M. Birn, V. Osipov, P. Sanders, C. Schulz, and N. Sitchinava, *Efficient parallel and external matching*, in European Conference on Parallel Processing, Springer, 2013, pp. 659–670.

[13] R. E. Blake, *Partitioning graph matching with constraints*, Pattern Recognit., 27 (1994), pp. 439–446.

[14] V. D. Blondel, J.-L. Guillaume, R. Lambiotte, and E. Lefebvre, *Fast unfolding of communities in large networks*, J. Stat. Mech.: Theory Exp., 2008 (2008), P10008.

[15] F. Bodon, *A fast apriori implementation*, in Workshop on Frequent Itemset Mining Implementations, IEEE Computer Society, 2003, pp. 56–65.

[16] I. Bogle and G. M. Slota, *Achieving speedups for distributed graph biconnectivity*, in 2022 IEEE High Performance Extreme Computing Conference (HPEC), IEEE, 2022, pp. 1–7.

[17] ——, *Distributed algorithms for the graph biconnectivity and least common ancestor problems*, in IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW), IEEE, 2022, pp. 1139–1142.

[18] P. Boldi, M. Rosa, M. Santini, and S. Vigna, *Layered label propagation: a multiresolution coordinate-free ordering for compressing social networks*, in Proceedings of the 20th International Conference on World Wide Web (WWW '11), ACM, 2011, p. 587–596.

[19] P. Boldi and S. Vigna, *The WebGraph framework I: Compression techniques*, in Proceedings of the Thirteenth International World Wide Web Conference (WWW 2004), ACM Press, 2004, pp. 595–601.

[20] E. G. Boman, K. D. Devine, and S. Rajamanickam, *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 (SC '13), Association for Computing Machinery, 2013, pp. 1–12.

[21] R. Burkard, M. Dell'Amico, and S. Martello, *Assignment Problems*, Society for Industrial and Applied Mathematics, Philadelphia, PA, USA, 2009.

[22] J. Callan, M. Hoy, C. Yoo, and L. Zhao, *ClueWeb09 Data Set*. http://www.lemurproject.org/clueweb09/index.php, 2009. (19 September 2022).

[23] H. Cao, Y. Wang, H. Wang, H. Lin, Z. Ma, W. Yin, and W. Chen, *Scaling graph traversal to 281 trillion edges with 40 million cores*, in Proceedings of the 27th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, ACM, 2022, pp. 234–245.

[24] Ü. V. Çatalyürek and C. Aykanat, *A fine-grain hypergraph model for 2d decomposition of sparse matrices.*, in 2001 IEEE International Parallel and Distributed Processing Symposium (IPDPS), vol. 1, IEEE Computer Society, 2001, pp. 118–125.

[25] Ü. V. Çatalyürek, C. Aykanat, and E. Kayaaslan, *Hypergraph partitioning-based fill-reducing ordering for symmetric matrices*, J. Sci. Comput., 33 (2011), pp. 1996–2023.

[26] S. Chakrabarti, B. Dom, S. Kumar, P. Raghavan, S. Rajagopalan, A. Tomkins, D. Gibson, and K. JM, *Mining the web's link structure*, Computer, 32 (1999), pp. 60 – 67.

[27] M. Charikar, M. T. Hajiaghayi, H. Karloff, and S. Rao, *L22 spreading metrics for vertex ordering problems*, in Proceedings of the Seventeenth Annual ACM-SIAM Symposium on Discrete Algorithm (SODA '06), Society for Industrial and Applied Mathematics, 2006, p. 1018–1027.

[28] S. Chaudhuri, V. Ganti, and R. Kaushik, *A primitive operator for similarity joins in data cleaning*, in 22nd International Conference on Data Engineering (ICDE'06), IEEE Computer Society, 2006, pp. 1–5.

[29] J. Chen, R. G. Edwards, and W. Mao, *Graph contractions for calculating correlation functions in lattice qcd*, in Proceedings of the Platform for Advanced Scientific Computing Conference (PASC '23), Association for Computing Machinery, 2023, 4935, pp. 1–10.

[30] F. Chierichetti, R. Kumar, S. Lattanzi, M. Mitzenmacher, A. Panconesi, and P. Raghavan, *On compressing social networks*, in Proceedings of the 15th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining (KDD '09), Association for Computing Machinery, 2009, p. 219–228.

[31] A. Ching, S. Edunov, M. Kabiljo, D. Logothetis, and S. Muthukrishnan, *One trillion edges: graph processing at facebook-scale*, Proc. VLDB Endow., 8 (2015), p. 1804–1815.

[32] H.-Y. Chou and S. Ghosh, *Batched graph community detection on gpus*, in Proceedings of the International Conference on Parallel Architectures and Compilation Techniques (PACT '22), Association for Computing Machinery, 2023, p. 172–184.

[33] C. W. Coley, W. Jin, L. Rogers, T. F. Jamison, T. S. Jaakkola, W. H. Green, R. Barzilay, and K. F. Jensen, *A graph-convolutional neural network model for the prediction of chemical reactivity*, Chem. Sci., 10 (2019), pp. 370–377.

[34] *cuDF: GPU DataFrame Library for Python.* RAPIDS AI and NVIDIA Corporation, https://github.com/rapidsai/cudf, 2025. (7 July 2025).

[35] *cuGraph: GPU Accelerated Graph Analytics.* NVIDIA Corporation and RAPIDS AI, https://github.com/rapidsai/cugraph, 2025. (7 July 2025).

[36] P. D'Ambra, F. Durastante, S. M. Ferdous, S. Filippone, M. Halappanavar, and A. Pothen, *Amg preconditioners based on parallel hybrid coarsening and multi-objective graph matching*, in 2023 31st Euromicro International Conference on Parallel, Distributed and Network-Based Processing (PDP), Euromicro, 2023, pp. 59–67.

[37] R. Dathathri, G. Gill, L. Hoang, V. Jatala, K. Pingali, V. K. Nandivada, H.-V. Dang, and M. Snir, *Gluon-async: A bulk-asynchronous system for distributed and heterogeneous graph analytics*, in 2019 28th International Conference on Parallel Architectures and Compilation Techniques (PACT), IEEE, 2019, pp. 15–28.

[38] E. Davidson and M. Levine, *Gene regulatory networks*, Proc. Natl. Acad. Sci. U.S.A., 102 (2005), 4935.

[39] T. A. Davis and Y. Hu, *The university of florida sparse matrix collection*, ACM Trans. Math. Softw., 38 (2011), pp. 1–25.

[40] D. Deng, A. Kim, S. Madden, and M. Stonebraker, *Silkmoth: an efficient method for finding related sets with maximum matching constraints*, Proc. VLDB Endow., 10 (2017), p. 1082–1093.

[41] B. Dezső, A. Jüttner, and P. Kovács, *Lemon–an open source c++ graph template library*, Electron. Notes Theor. Comput. Sci., 264 (2011), pp. 23–45.

[42] L. Dhulipala, I. Kabiljo, B. Karrer, G. Ottaviano, S. Pupyrev, and A. Shalita, *Compressing graphs and indexes with recursive graph bisection*, in Proceedings of the 22nd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining (KDD '16), Association for Computing Machinery, 2016, p. 1535–1544.

[43] L. Dhulipala, J. Shi, T. Tseng, G. E. Blelloch, and J. Shun, *The graph based benchmark suite (gbbs)*, in Proceedings of the 3rd Joint International Workshop on Graph Data Management Experiences & Systems and Network Data Analytics (GRADES-NDA'20), Association for Computing Machinery, 2020, pp. 1–8.

[44] D. E. Drake and S. Hougardy, *A simple approximation algorithm for the weighted matching problem*, Inf. Process. Lett., 85 (2003), pp. 211–213.

[45] R. Duan and S. Pettie, *Linear-time approximation for maximum weight matching*, JACM, 61 (2014), pp. 1–23.

[46] I. S. Duff and J. Koster, *On algorithms for permuting large entries to the diagonal of a sparse matrix*, SIAM J. Matrix Anal. Appl., 22 (2001), pp. 973–996.

[47] A. D'Alconzo, I. Drago, A. Morichetta, M. Mellia, and P. Casas, *A survey on big data for network traffic monitoring and analysis*, IEEE TNSM, 16 (2019), pp. 800–813.

[48] J. Edmonds, *Maximum matching and a polyhedron with 0, 1-vertices*, J. Res. Natl. Inst. Stand. Technol., 69 (1965), pp. 55–56.

[49] ——, *Paths, trees, and flowers*, Can. J. Math., 17 (1965), pp. 449–467.

[50] B. O. Fagginger Auer and R. H. Bisseling, *A GPU Algorithm for Greedy Graph Matching*, in Facing the Multicore - Challenge II: Aspects of New Paradigms and Technologies in Parallel Computing, R. Keller, D. Kramer, and J.-P. Weiss, Eds., Springer, Berlin, Germany, 2012, pp. 108–119.

[51] W. Fan, T. He, L. Lai, X. Li, Y. Li, Z. Li, Z. Qian, C. Tian, L. Wang, J. Xu, et al., *Graphscope: a unified engine for big graph processing*, Proc. VLDB Endow., 14 (2021), pp. 2879–2892.

[52] S. M. Ferdous, A. Pothen, and M. Halappanavar, *Streaming Matching and Edge Cover in Practice*, in 22nd International Symposium on Experimental Algorithms (SEA 2024), Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2024, pp. 12:1–12:22.

[53] F. Fier, N. Augsten, P. Bouros, U. Leser, and J.-C. Freytag, *Set similarity joins on mapreduce: an experimental survey*, Proc. VLDB Endow., 11 (2018), pp. 1110–1122.

[54] D. Foley and J. Danskin, *Ultra-performance pascal gpu and nvlink interconnect*, IEEE Micro, 37 (2017), pp. 7–17.

[55] M. Frasca, K. Madduri, and P. Raghavan, *NUMA-aware graph mining techniques for performance and energy efficiency*, in Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis (SC '12), 2012, ACM and IEEE Computer Society, pp. 1–11.

[56] Z. Galil, *Efficient algorithms for finding maximum matching in graphs*, ACM Comput. Surv., 18 (2002), pp. 23–38.

[57] X. Gan, G. Wu, S. Qiu, F. Xiong, J. Si, J. Fang, D. Dong, C. Gong, T. Li, and Z. Wang, *Graphcube: Interconnection hierarchy-aware graph processing*, in Proceedings of the 29th ACM SIGPLAN Annual Symposium on Principles and Practice of Parallel Programming, ACM, 2024, pp. 160–174.

[58] M. R. Garey, D. S. Johnson, and L. J. Stockmeyer, *Some simplified np-complete graph problems*, Theor. Comput. Sci., 1 (1976), pp. 237–267.

[59] M. Ghaffari and D. Wajc, *Simplified and space-optimal semi-streaming* $(2+\epsilon)$*-approximate matching*, in Proceedings of the 2nd Symposium on Simplicity in Algorithms (SOSA), Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2019, pp. 13:1–13:8.

[60] Y. Go, M. Jamshed, Y. Moon, C. Hwang, and K. Park, *Apunet: revitalizing gpu as packet processing accelerator*, in Proceedings of the 14th USENIX Conference on Networked Systems Design and Implementation (NSDI'17), USENIX Association, 2017, p. 83–96.

[61] J. E. Gonzalez, Y. Low, H. Gu, D. Bickson, and C. Guestrin, *Power-Graph: Distributed Graph-Parallel Computation on Natural Graphs*, in 10th USENIX Symposium on Operating Systems Design and Implementation (OSDI 12), USENIX Association, 2012, pp. 17–30.

[62] O. Green and D. A. Bader, *cuSTINGER: Supporting dynamic graph algorithms for gpus*, in 2016 IEEE High Performance Extreme Computing Conference (HPEC), IEEE, 2016, pp. 1–6.

[63] A. A. Hagberg, D. A. Schult, and P. J. Swart, *Exploring network structure, dynamics, and function using NetworkX*, in Proceedings of the 7th Python in Science Conference, NumFOCUS, 2008, pp. 11 – 15.

[64] M. Halappanavar, J. Feo, O. Villa, A. Tumeo, and A. Pothen, *Approximate weighted matching on emerging manycore and multithreaded architectures*, Int. J. High Perform. Comput. Appl., 26 (2012), pp. 413–430.

[65] B. Hendrickson, R. Leland, and S. Plimpton, *An efficient parallel algorithm for matrix-vector multiplication*, Int. J. High Speed Comput., 7 (1995), pp. 73–88.

[66] J. Hoberock, E. Niebler, L. Gooch, and B. A. L. Maurer, *P2300R10: std::execution*. https://www.open-std.org/jtc1/sc22/wg21/docs/papers/2024/p2300r10.html. (7 July 2025).

[67] I. Ismayilov, J. Baydamirli, D. Sağbili, M. Wahib, and D. Unat, *Multi-gpu communication schemes for iterative solvers: When cpus are not in charge*, in Proceedings of the 37th International Conference on Supercomputing (ICS '23), 2023, ACM, p. 192–202.

[68] P. Jaccard, *Étude comparative de la distribution florale dans une portion des alpes et des jura*, Bull. Soc. Vaud. Sci. Nat., 37 (1901), pp. 547–579.

[69] V. Jatala, R. Dathathri, G. Gill, L. Hoang, V. K. Nandivada, and K. Pingali, *A study of graph analytics for massive datasets on distributed gpus*, in International Parallel and Distributed Processing Symposium (IPDPS), IEEE Computer Society, 2020, pp. 84–94.

[70] S. Jeaugey, *Nccl 2.0*, in GPU Technology Conference (GTC), vol. 2, NVIDIA, 2017, pp. 1–23.

[71] Z. Jia, Y. Kwon, G. Shipman, P. McCormick, M. Erez, and A. Aiken, *A distributed multi-gpu system for fast graph processing*, Proc. VLDB Endow., 11 (2017), pp. 297–310.

[72] S. R. Jino Ramson and D. J. Moni, *Applications of wireless sensor networks — a survey*, in 2017 International Conference on Innovations in Electrical, Electronics, Instrumentation and Media Technology (ICEEIMT), IEEE Computer Society, 2017, pp. 325–329.

[73] S. Kang, J. Nke, and B. Rees, *Analyzing multi-trillion edge graphs on large gpu clusters: A case study with pagerank*, in 2022 IEEE High Performance Extreme Computing Conference (HPEC), IEEE, 2022, pp. 1–7.

[74] G. Karypis and V. Kumar, *A parallel algorithm for multilevel graph partitioning and sparse matrix ordering*, J. Parallel Distrib. Comput., 48 (1998), pp. 71–95.

[75] I. Kawaminami, A. Estrada, Y. Elsakkary, H. Jananthan, A. Buluc, T. Davis, D. Grant, M. Jones, C. Meiners, A. Morris, S. Pisharody, and J. Kepner, *Large scale enrichment and statistical cyber characterization of network traffic*, in 2022 IEEE High Performance Extreme Computing Conference (HPEC), IEEE, 2022, p. 1–7.

[76] J. Kepner and J. Gilbert, *Graph Algorithms in the Language of Linear Algebra*, Society for Industrial and Applied Mathematics, Philadelphia, PA, USA, 2011.

[77] J. Kepner, M. Jones, P. Dykstra, C. Byun, T. Davis, H. Jananthan, W. Arcand, D. Bestor, W. Bergeron, V. Gadepally, M. Houle, M. Hubbell, A. Klein, L. Milechin, G. Morales, J. Mullen, R. Patel, A. Pentland, S. Pisharody, A. Prout, A. Reuther, A. Rosa, S. Samsi, T. Trigg, C. Yee, and P. Michaleas, *Focusing and calibration of large scale network sensors using graphblas anonymized hypersparse matrices*, in 2023 IEEE High Performance Extreme Computing Conference (HPEC), IEEE, 2023, p. 1–9.

[78] R. Kohavi, C. E. Brodley, B. Frasca, L. Mason, and Z. Zheng, *Kdd-cup 2000 organizers' report: peeling the onion*, SIGKDD Explor. Newsl., 2 (2000), p. 86–93.

[79] M. Koohi Esfahani, P. Boldi, H. Vandierendonck, P. Kilpatrick, and S. Vigna, *On overcoming hpc challenges of trillion-scale real-world graph datasets*, in 2023 IEEE International Conference on Big Data (BigData), IEEE, 2023, pp. 215–220.

[80] H. W. Kuhn, *The hungarian method for the assignment problem*, Nav. Res. Logist., 2 (1955), pp. 83–97.

[81] H. Kwak, C. Lee, H. Park, and S. Moon, *What is twitter, a social network or a news media?*, in Proceedings of the 19th International Conference on World Wide Web (WWW '10), Association for Computing Machinery, 2010, p. 591–600.

[82] D. Lasalle and G. Karypis, *Multi-threaded graph partitioning*, in Proceedings of the 2013 IEEE 27th International Symposium on Parallel and Distributed Processing (IPDPS '13), IEEE Computer Society, 2013, p. 225–236.

[83] *LEMON: Library for Efficient Modeling and Optimization in Networks*. LEMON Contributors, https://lemon.cs.elte.hu/pub/doc/latest-svn/index.html. (2 April 2024).

[84] J. Leskovec, K. J. Lang, A. Dasgupta, and M. W. Mahoney, *Community structure in large networks: Natural cluster sizes and the absence of large well-defined clusters*, Internet Math., 6 (2008), pp. 123 – 129.

[85] *libcu++: The NVIDIA C++ Standard Library*. NVIDIA Corporation, https://github.com/NVIDIA/libcudacxx, 2024. (7 July 2025).

[86] D. Liben-Nowell and J. Kleinberg, *The link prediction problem for social networks*, in Proceedings of the Twelfth International Conference on Information and Knowledge Management (CIKM '03), Association for Computing Machinery, 2003, p. 556–559.

[87] Y. Lim, U. Kang, and C. Faloutsos, *Slashburn: Graph compression and mining beyond caveman communities*, IEEE Trans. Knowl. Data Eng., 26 (2014), pp. 3077–3089.

[88] H. Lin, X. Zhu, B. Yu, X. Tang, W. Xue, W. Chen, L. Zhang, T. Hoefler, X. Ma, X. Liu, et al., *Shentu: processing multi-trillion edge graphs on millions of cores in seconds*, in International Conference for High Performance Computing, Networking, Storage and Analysis (SC18), IEEE, 2018, pp. 706–716.

[89] A. Lumsdaine, D. Gregor, B. Hendrickson, and J. Berry, *Challenges in parallel graph processing.*, Parallel Process. Lett., 17 (2007), pp. 5–20.

[90] S. Maass, C. Min, S. Kashyap, W. Kang, M. Kumar, and T. Kim, *Mosaic: Processing a trillion-edge graph on a single machine*, in Proceedings of the Twelfth European Conference on Computer Systems, ACM, 2017, pp. 527–543.

[91] G. Malewicz, M. H. Austern, A. J. Bik, J. C. Dehnert, I. Horn, N. Leiser, and G. Czajkowski, *Pregel: a system for large-scale graph processing*, in Proceedings of the 2010 ACM SIGMOD International Conference on Management of Data, ACM, 2010, pp. 135–146.

[92] W. Mann, N. Augsten, and P. Bouros, *An empirical evaluation of set similarity join techniques*, Proc. VLDB Endow., 9 (2016), p. 636–647.

[93] F. Manne and R. H. Bisseling, *A parallel approximation algorithm for the weighted maximum matching problem*, in International Conference on Parallel Processing and Applied Mathematics, Springer, 2007, pp. 708–717.

[94] F. Manne and M. Halappanavar, *New effective multithreaded matching algorithms*, in 2014 IEEE 28th International Parallel and Distributed Processing Symposium, IEEE Computer Society, 2014, pp. 519–528.

[95] B. Mascarenhas and K. Puranam, *Analysis of the medical residency matching algorithm to validate and improve equity*, PLOS ONE, 18 (2023), pp. 1–11.

[96] R. R. McCune, T. Weninger, and G. Madey, *Thinking like a vertex: A survey of vertex-centric frameworks for large-scale distributed graph processing*, ACM Comput. Surv., 48 (2015), pp. 1–39.

[97] F. McSherry, M. Isard, and D. G. Murray, *Scalability! but at what {COST}?*, in 15th Workshop on Hot Topics in Operating Systems (HotOS XV), USENIX Association, 2015, pp. 1–5.

[98] S. Melnik, H. Garcia-Molina, and E. Rahm, *Similarity flooding: A versatile graph matching algorithm and its application to schema matching*, in Proceedings 18th International Conference on Data Engineering, IEEE, 2002, pp. 117–128.

[99] D. Merrill, M. Garland, and A. Grimshaw, *Scalable gpu graph traversal*, ACM SIGPLAN Not., 47 (2012), pp. 117–128.

[100] A. Mislove, M. Marcon, K. P. Gummadi, P. Druschel, and B. Bhattacharjee, *Measurement and analysis of online social networks*, in Proceedings of the 5th ACM/Usenix Internet Measurement Conference (IMC'07), ACM and USENIX, October 2007, pp. 29–42.

[101] G. Muzio, L. O'Bray, and K. Borgwardt, *Biological network analysis with deep learning*, Brief. Bioinform., 22 (2021), pp. 1515–1530.

[102] M. Naim, F. Manne, M. Halappanavar, A. Tumeo, and J. Langguth, *GPU Suitor*. https://hpc.pnl.gov/people/hala/suitor.html. (2 April 2024).

[103] ——, *Optimizing approximate weighted matching on nvidia kepler k40*, in Proceedings of the 2015 IEEE 22nd International Conference on High Performance Computing (HiPC), 2015, pp. 105–114.

[104] G. Navarro, *A guided tour to approximate string matching*, ACM Comput. Surv., 33 (2001), p. 31–88.

[105] *nvexec: Sender/Receiver Framework for GPU-Centric Asynchronous Programming*. NVIDIA Corporation, https://github.com/NVIDIA/nvexec, 2024. (7 July 2025).

[106] *NVIDIA's Blackwell Offers FP4, Second-Gen Transformer Engine*. https://www.eetimes.com/nvidias-blackwell-gpu-offers-fp4-transformer-engine-sharp/, 2024. (15 November 2025).

[107] G. Pass, A. Chowdhury, and C. Torgeson, *A picture of search*, in Proceedings of the 1st International Conference on Scalable Information Systems, ACM, 2006, p. 1–10.

[108] A. Paz and G. Schwartzman, *A (2+ϵ)-approximation for maximum weight matching in the semi-streaming model*, in Proceedings of the Twenty-Eighth Annual ACM-SIAM Symposium on Discrete Algorithms (SODA), SIAM, 2017, pp. 2153–2161.

[109] S. Pettie and P. Sanders, *A simpler linear time 2/3-ε approximation for maximum weight matching*, Inf. Process. Lett., 91 (2004), pp. 271–276.

[110] A. Pothen, S. Ferdous, and F. Manne, *Approximation algorithms in combinatorial scientific computing*, Acta Numer., 28 (2019), pp. 541–633.

[111] R. Preis, *Linear time 1/2-approximation algorithm for maximum weighted matching in general graphs*, in Annual Symposium on Theoretical Aspects of Computer Science, Springer, 1999, pp. 259–269.

[112] P. Ratzloff, A. Lumsdaine, K. Deweese, M. Osama, S. McMillan, J. Firoz, M. Wong, J. Maurer, R. Dosselmann, M. Galati, G. Davidson, and O. Rosten, *Graph library: Comparison*, Tech. Rep. P3337r0, International Organization for Standardization / International Electrotechnical Commission (ISO/IEC), Geneva, Switzerland, 2025.

[113] A. H. N. Sabet, Z. Zhao, and R. Gupta, *Subway: Minimizing data transfer during out-of-gpu-memory graph processing*, in Proceedings of the Fifteenth European Conference on Computer Systems, ACM, 2020, pp. 1–16.

[114] I. Safro and B. Temkin, *Multiscale approach for the network compression-friendly ordering*, J. Discrete Algorithms, 9 (2011), pp. 190–202.

[115] N. Sakharnykh, *Everything you need to know about unified memory*, in GPU Technology Conference (GTC), vol. 64, NVIDIA, 2018, pp. 1–5.

[116] S. Sarawagi and A. Kirpal, *Efficient set joins on similarity predicates*, in Proceedings of the 2004 ACM SIGMOD International Conference on Management of Data (SIGMOD '04), Association for Computing Machinery, 2004, p. 743–754.

[117] A. D. Sarma, L. Fang, N. Gupta, A. Y. Halevy, H. Lee, F. Wu, R. Xin, and C. Yu, *Finding related tables.*, in Proceedings of the 2012 ACM SIGMOD International Conference on Management of Data (SIGMOD '12'), vol. 10, 2012, pp. 2213836–2213962.

[118] N. Satish, N. Sundaram, M. M. A. Patwary, J. Seo, J. Park, M. A. Hassaan, S. Sengupta, Z. Yin, and P. Dubey, *Navigating the maze of graph analytics frameworks using massive graph datasets*, in Proceedings of the 2014 ACM SIGMOD International Conference on Management of Data (SIGMOD '14), Association for Computing Machinery, 2014, p. 979–990.

[119] N. Schlömer, *pygalmesh: Python Interface for CGAL's Meshing Tools*. https://github.com/nschloe/pygalmesh, 2019. (6 April 2022).

[120] M. Schmidt, *Packet switching in multi-queue switches*, in Encyclopedia of Algorithms, M.-Y. Kao, ed., Springer US, Boston, MA, 2008, pp. 618–620.

[121] D. Schultes, *The Shortest Path Problem*. http://www.diag.uniroma1.it/challenge9/data/tiger/, 2005. (6 April 2022).

[122] D. Sengupta, S. L. Song, K. Agarwal, and K. Schwan, *Graphreduce: processing large-scale graphs on accelerator-based systems*, in Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis (SC'15), ACM and IEEE Computer Society, 2015, pp. 1–12.

[123] F. Sgherzi, A. Parravicini, and M. D. Santambrogio, *A mixed precision, multi-gpu design for large-scale top-k sparse eigenproblems*, in 2022 IEEE International Symposium on Circuits and Systems (ISCAS), IEEE Circuits and Systems Society, 2022, pp. 1259–1263.

[124] X. Shi, Z. Zheng, Y. Zhou, H. Jin, L. He, B. Liu, and Q.-S. Hua, *Graph processing on gpus: A survey*, ACM Comput. Surv., 50 (2018), pp. 1–35.

[125] G. M. Slota, J. W. Berry, S. D. Hammond, S. L. Olivier, C. A. Phillips, and S. Rajamanickam, *Scalable generation of graphs for benchmarking hpc community-detection algorithms*, in Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis (SC'19), ACM and IEEE Computer Society, 2019, pp. 1–14.

[126] G. M. Slota and C. Brissette, *Constant-memory graph coarsening*, in 2024 IEEE High Performance Extreme Computing Conference (HPEC), IEEE, 2024, pp. 1–7.

[127] G. M. Slota, S. Rajamanickam, K. Devine, and K. Madduri, *Partitioning trillion-edge graphs in minutes*, in 2017 IEEE International Parallel and Distributed Processing Symposium (IPDPS), IEEE, 2017, pp. 646–655.

[128] G. M. Slota, S. Rajamanickam, and K. Madduri, *Bfs and coloring-based parallel algorithms for strongly connected components and related problems*, in 2014 IEEE 28th International Parallel and Distributed Processing Symposium, IEEE Computer Society, 2014, pp. 550–559.

[129] G. M. Slota, S. Rajamanickam, and K. Madduri, *High-performance graph analytics on manycore processors*, in 2015 IEEE International Parallel and Distributed Processing Symposium, IEEE, 2015, pp. 17–27.

[130] ——, *A case study of complex graph analysis in distributed memory: Implementation and optimization*, in 2016 IEEE International Parallel and Distributed Processing Symposium (IPDPS), IEEE, 2016, pp. 293–302.

[131] E. Solomonik and J. Demmel, *Communication-optimal parallel 2.5 d matrix multiplication and lu factorization algorithms*, in European Conference on Parallel Processing, Springer, 2011, pp. 90–109.

[132] C. L. Staudt, A. Sazonovs, and H. Meyerhenke, *Networkit: A tool suite for large-scale complex network analysis*, Netw. Sci., 4 (2016), pp. 508–530.

[133] K. Świrydowicz, E. Darve, W. Jones, J. Maack, S. Regev, M. A. Saunders, S. J. Thomas, and S. Peleš, *Linear solvers for power grid optimization problems: a review of gpu-accelerated linear solvers*, Parallel Comput., 111 (2022), 102870.

[134] I. K. Tezaur, M. Perego, A. G. Salinger, R. S. Tuminaro, and S. F. Price, *Albany/felix: a parallel, scalable and robust, finite element, first-order stokes approximation ice sheet solver built for advanced analysis*, Geosci. Model Dev., 8 (2015), pp. 1197–1220.

[135] *The Graph 500 Benchmark*. https://graph500.org/. (20 December 2024).

[136] *The Graph Challenge*. https://graphchallenge.mit.edu/. (20 December 2024).

[137] A. S. Tom and G. Karypis, *A 2d parallel triangle counting algorithm for distributed-memory architectures*, in Proceedings of the 48th International Conference on Parallel Processing (ICPP '19), Association for Computing Machinery, 2019, pp. 1–10.

[138] J. Wang, G. Li, and J. Fe, *Fast-join: An efficient method for fuzzy token matching based string similarity join*, in 2011 IEEE 27th International Conference on Data Engineering, IEEE Computer Society, 2011, pp. 458–469.

[139] J. Wang, G. Li, and J. Feng, *Extending string similarity join to tolerant fuzzy token matching*, ACM Trans. Database Syst., 39 (2014), pp. 7:1–7:45.

[140] J. Wang, C. Lin, and C. Zaniolo, *MF-Join: Efficient fuzzy string similarity join with multi-level filtering*, in 2019 IEEE 35th International Conference on Data Engineering (ICDE), IEEE Computer Society, 2019, pp. 386–397.

[141] 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, ACM, 2016, pp. 1–12.

[142] W. Wu and P. Demar, *A gpu-accelerated network traffic monitoring and analysis system*, in 2013 IEEE Conference on Computer Communications Workshops (INFOCOM WKSHPS), IEEE, 2013, pp. 77–78.

[143] Q. Xu, H. Jeon, and M. Annavaram, *Graph processing on gpus: Where are the bottlenecks?*, in 2014 IEEE International Symposium on Workload Characterization (IISWC), IEEE Computer Society, 2014, pp. 140–149.

[144] C. Yang, A. Buluç, and J. D. Owens, *Graphblast: A high-performance linear algebra-based graph framework on the gpu*, ACM Trans. Math. Softw., 48 (2022), pp. 1–51.

[145] J. Yang and J. Leskovec, *Defining and evaluating network communities based on ground-truth*, in Proceedings of the ACM SIGKDD Workshop on Mining Data Semantics (MDS '12), ACM, 2012, pp. 1–8.

[146] A. Zeakis, D. Skoutas, D. Sacharidis, O. Papapetrou, and M. Koubarakis, *Tokenjoin: Efficient filtering for set similarity join with maximum weighted bipartite matching*, Proc. VLDB Endow., 16 (2022), p. 790–802.

[147] W. Zhang and J. P. Lazaro, *A survey on network security traffic analysis and anomaly detection techniques*, Int. J. Emerg. Technol. Adv. Appl., 1 (2024), p. 8–16.

[148] K. Zhao, Y. Rong, J. X. Yu, W. Huang, J. Huang, and H. Zhang, *Graph ordering: Towards the optimal by learning*, in International Conference on Web Information Systems Engineering, Springer, 2021, pp. 423–437.

[149] K. Zhao, Y. Zhou, H. Pan, Z. Wang, S. Zhong, and C. Tian, *Sans: Streaming anonymized network sensing*, in 2024 IEEE High Performance Extreme Computing Conference (HPEC), IEEE, 2024, pp. 1–7.

[150] W. Zhong, J. Sun, H. Chen, J. Xiao, Z. Chen, C. Cheng, and X. Shi, *Optimizing graph processing on gpus*, IEEE Trans. Parallel Distrib. Syst., 28 (2016), pp. 1149–1162.

[151] D. Zhou, O. Bousquet, T. Lal, J. Weston, and B. Olkopf, *Learning with local and global consistency*, in Advances in Neural Information Processing Systems 16, Neural Information Processing Systems Foundation, 2004, pp. 321–328.

[152] W. Zhou, Y. Zhou, J. Li, and M. H. Memon, *Lsrec: Large-scale social recommendation with online update*, Expert Syst. Appl., 162 (2020), 113739.

[153] E. Zhu, D. Deng, F. Nargesian, and R. J. Miller, *Josie: Overlap set similarity search for finding joinable tables in data lakes*, in Proceedings of the 2019 International Conference on Management of Data (SIGMOD '19), Association for Computing Machinery, 2019, p. 847–864.

[154] X. Zhu, W. Chen, W. Zheng, and X. Ma, *Gemini: a computation-centric distributed graph processing system*, in Proceedings of the 12th USENIX Conference on Operating Systems Design and Implementation (OSDI'16), USENIX Association, 2016, p. 301–316.

[155] V. Çatalyürek, F. Dobrian, A. Gebremedhin, M. Halappanavar, and A. Pothen, *Distributed-memory parallel algorithms for matching and coloring*, in 2011 IEEE International Symposium on Parallel and Distributed Processing Workshops and Phd Forum, IEEE Computer Society, 2011, pp. 1971–1980.