Efficient Weighted Graph Matching on GPUs

Michael Mandulak
Department of Computer Science
Rensselaer Polytechnic Institute
Troy, USA
mandum@rpi.edu

Sayan Ghosh
Pacific Northwest National Laboratory
Richland, USA
sayan.ghosh@pnnl.gov

S M Ferdous
Pacific Northwest National Laboratory
Richland, USA
sm.ferdous@pnnl.gov

Mahantesh Halappanvar
Pacific Northwest National Laboratory
Richland, USA
mahantesh.halappanavar@pnnl.gov

George Slota
Department of Computer Science
Rensselaer Polytechnic Institute
Troy, USA
slotag@rpi.edu

***Abstract*—Weighted matching identifies a maximal subset of edges in a graph such that these edges do not share any vertices in common with each other. As a prototypical graph problem, matching has numerous applications in science and engineering, such as linear algebra, multi-level graph algorithms, computer vision and machine learning. There is a critical need for efficient matching algorithms. However, there are challenges in developing efficient, parallel graph matching methods on contemporary GPGPU systems, 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. Although dense-GPU systems are mainstream and offer accelerated on-node interconnection to enhance data access bandwidth, data dependencies and device synchronization costs in multi-GPU enabled massive-graph processing create challenges to sustainable scalability.**

Considering these challenges, we present efficient approximation algorithms for *locally dominant matching*, and we demonstrate scalability via batching and distributing graph data across multiple NVIDIA A100/V100 GPUs of NVIDIA DGX dense-GPU platforms. Our locally dominant (pointer-based) matching method exhibits 2-45$\times$ performance improvements compared to state-of-the-art single-GPU and multithreaded CPU matching implementations on a variety of real-world and synthetic graphs. We show competitive quality comparisons and detailed analysis of GPU-data distribution considerations for practical and efficient weighted graph matching on GPUs.

***Index Terms*—Graph Analytics, Maximal Weighted Matching, GPGPU, CUDA**

I. Introduction

Given a graph $G(V, E, w)$, where $w : E \rightarrow \mathbf{R_{>0}}$ 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) [31], numerical linear algebra [11], [16], computer vision and pattern recognition [4], and a variety of scheduling, resource allocation and facility location problems [1], [8].

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 [27]. 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 paths are amenable to parallelization and therefore perform significantly better on parallel systems [23]. An approximation algorithm is required to generate a solution with a provable bound to the optimal one (detailed in §II).

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 [26], and past research proposed efficient linear algebra based parallel algorithm for finding a perfect matching in a weighted bipartite graph [3], 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 [35].

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™ [19], 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 [37], [38]. Even though graph processing workloads are known for irregular data movement overheads leading to implementation and scalability challenges [40], by processing in batches, graph algorithm are able to regularize the synchronization requirements and thus exploit vendor-optimized GPU collective libraries such as NCCL™ [25] 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.

- We extend the 1/2-approximate locally dominant matching to multi-GPU setting.
- 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 with maintaining the approximation ratio.
- We demonstrate 2–45$\times$ 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 ($\sim$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.

II. Background and Related Work

A. Preliminaries

a) 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. Similarly, $\mathcal{N}(e)$ is also defined. 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.

b) 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 $w(M) \geq \alpha\ w(M^*)$.

B. Locally dominant algorithm

Definition II.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. 1 assuming $M = \emptyset$, the edges {1,0} and {3,4} are locally dominating while {2,3} and {5,4} are not.

We restate the approximation result of a locally dominant algorithm by Preis [36].

Lemma II.1 ([36])**.** 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 [6] and Suitor [30] algorithms described in the literature are two examples of locally dominant frameworks. We next discuss a pointer-based locally dominant algorithm in Algorithm 1, which will provide a base for our multi-GPU algorithms described in the subsequent section. Each iteration of Algorithm 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

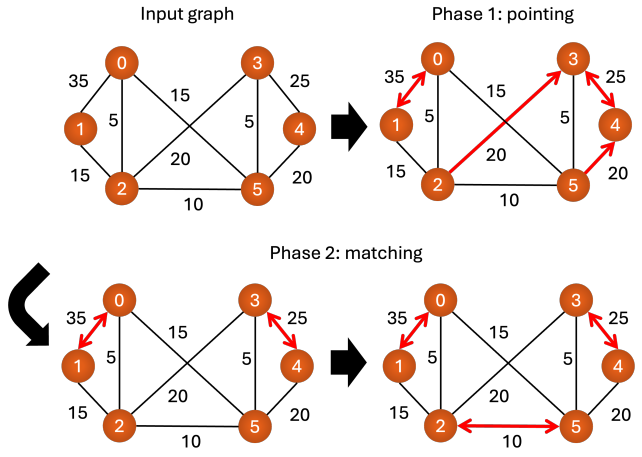


Fig. 1. One iteration of the LD-SEQ algorithm when $M = \emptyset$: *pointing*: for each vertex, choose the heaviest neighbor, and, *matching*: if two vertices point to each other, add the edge to M; remove all edges incident on M, repeat.

(mate) with the highest weight. In the next phase, 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. 1.

Algorithm 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 $mate(u) = v$ ▷ *LD edge*
7: $M = M \cup e$
8: $G = G \setminus \{e \cup \mathcal{N}(e)\}$

Lemma II.2. The matching M generated by Algorithm 1 is maximal and locally dominant.

Proof. We start with $M = \emptyset$, which is locally dominant trivially. 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 II.1, M is also locally dominant. We continue until G is empty, which renders a maximal matching. □

The following corollary immediately follows from Lemma II.2 and Lemma II.1.

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

C. Related Work

Although MWM is solvable in polynomial time [17], [18], 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 [2], [14], [15], [34], [36]. The breakthrough result for designing practical parallel algorithms is the locally dominant algorithm by Preis [36]. 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 [6], [29] and stable matching-based suitor algorithms (henceforth, SR-OMP) [30]. We refer to [35] for a detailed description of many of these algorithms.

Fagginger et al. [20] 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 [6], [33]. Birn et al. [6] uses the pointer-based approach, while Naim et al. [33] 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 paper.

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 [5], [41] Recent works pertaining to scalable graph-related computations on multiple GPUs have been shown to utilize randomized or naturally ordered partitions across multiple devices [9], [24], [39]. For our purposes, we draw from methods that employ batching and sampling strategies [10], [22], [42] to balance edge counts across devices while considering scalability relative to the number of devices allocated for computation.

III. 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 paper. 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.

A. 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.

B. Batching

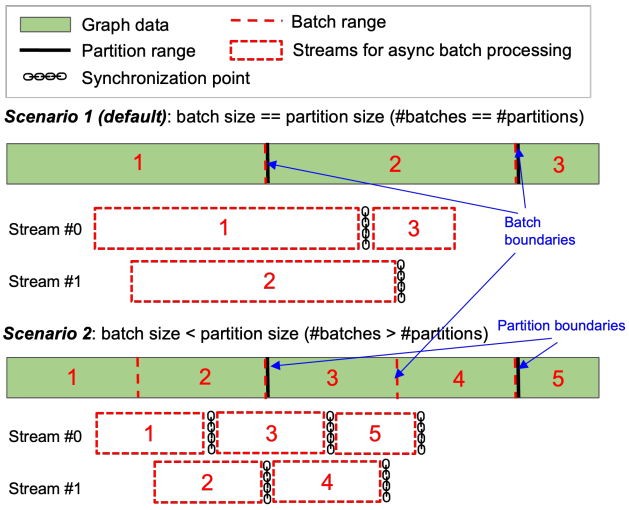


Fig. 2. Scenarios concerning batches and partitions (on a single device) and depicting asynchronous batch processing through CUDA streams.

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 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. 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

Algorithm 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:              strm ← b mod 2
5:              LOADBATCH<strm>(G_id, b)
6:              SETPOINTERS<strm>(G_id^b, 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 ← V_i^b
12:     cudaMemcpyAsyncHtoD(v_batch, strm)
```

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. 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 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 Section IV.

C. 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 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.

D. GPU Implementation

We now discuss the details of our GPU implementation, LD-GPU. We provide a general overview of the algorithm in Fig. 3, which depicts the *pointing* and *matching* phases, as introduced in Algorithm 1. We first provide a high-level description in Algorithm 2 followed by specific kernels in Algorithm 3.

Algorithm 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 vertex u per thread do
12:         if pointers[pointers[u]] = u     ▷ Mutual Check
13:             mate[u] ← pointers[u]
```

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. 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 II.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 [25], 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.

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 available 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 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 §IV. 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 III.1. The matching produces by LD-GPU is $\frac{1}{2}$-approximate for `MWM`.

Proof. It is sufficient to show that the edges committed to matching (Line 12 in Algorithm 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 II.1. We note that the only difference between the sequential LD-SEQ

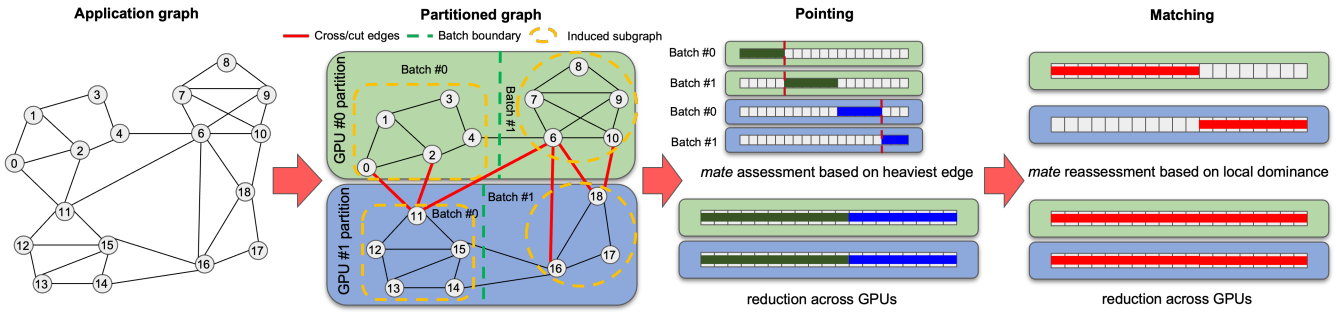


Fig. 3. LD-GPU algorithm illustration considering partitions and batches using multiple GPUs. A graph is first partitioned among devices and logically arranged into ranges of vertices (and adjacent edges) called *batches*. Each batch is processed independently through the pointing phase, followed by a global reduction, the matching phase and another global reduction to synchronize device matching information.

algorithm (Algorithm 1) and LD-GPU (Algorithm 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 3), and (ii) v is in the cross/cut edges (the red edges in Figure 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 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 kernel. 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. □

IV. 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 in this paper, 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/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. Our implementation is publicly available as part of the ExaGraph project repository.[1]

[1]https://github.com/ECP-ExaGraph/sumac

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 [12], except uk-2007-05 and webbase-2001, which are web-crawl graphs from the LAW collection [7]. 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 of three decimal points from $[0, 1]$. Our datasets are listed in Table I, organized into groups: (i) LARGE: graphs with #edges $>$ 1B, and (ii) SMALL: graphs with #edges $\leq$ 1B.

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 §IV-A. Then, we analyze the execution time performance and scalability of LD-GPU against various inputs and systems in §IV-B. In §IV-C, we dissect the GPU utilization of LD-GPU considering variations in graph structure

TABLE I

(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×

and resulting distributions. Finally, in §IV-D, we compare the overall performances of LD-GPU with state-of-the-art OpenMP-based CPU (SR-OMP), single GPU (SR-GPU) and NVIDIA™ RAPIDS™ cuGraph implementations respectively, and introduce a unique Figure-of-Merit (FoM) for assessing the quality and performance of the results.

A. 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) [13]. 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 II, 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 II

LD-GPU AND SR-OMP QUALITY PERCENTAGE DIFFERENCE RELATIVE TO LEMON ON THE SMALL GRAPH INSTANCES.

	Percentage Diff. from LEMON	
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

B. Baseline Performance

Scalability: Fig. 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

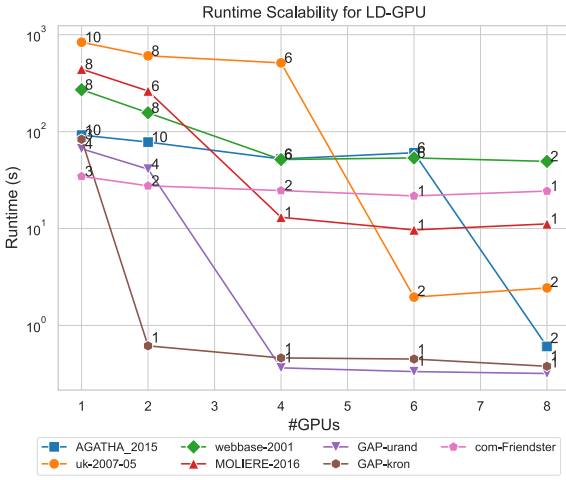


Fig. 4. Strong scaling for LD-GPU on 1–8 GPUs, using a variety of batch counts and choosing the best execution time over 10 runs.

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.

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. 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, as shown in the component-wise timing in Fig. 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. 7 and Fig. 6). We anticipate subsequent batch transfer overheads would ultimately impact the scalability beyond a certain point.

Component-wise timing analysis: In Fig. 5, we examine the individual execution times of the high-level components in Algorithm 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 §III-B). Fig. 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

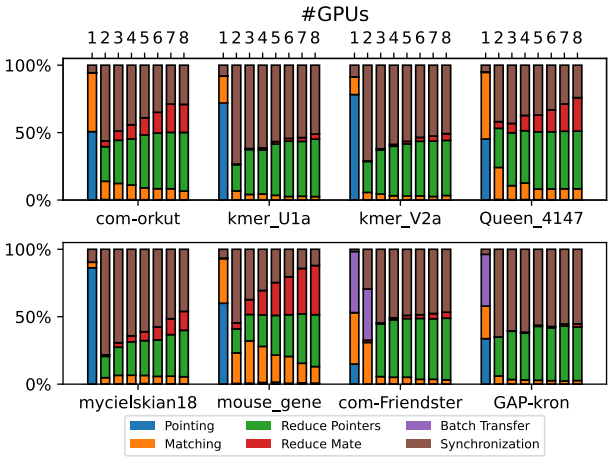


Fig. 5. Component-wise timing (in terms of %-overall in Y-axis) for SMALL/LARGE graphs (X-axis) for variable #batches/GPU on 1–8 GPUs.

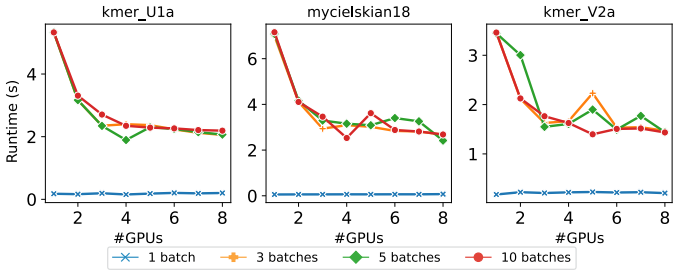


Fig. 6. LD-GPU using 1 (default), 3, 5 and 10 batches on 1–8 GPUs.

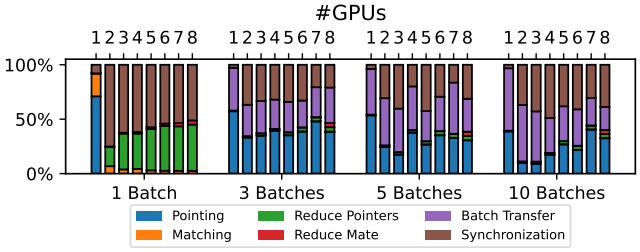


Fig. 7. Component-wise timing (%-overall in Y-axis) for kmer_U1a graph using LD-GPU with 1 (default), 3, 5 and 10 batches (X-axis) on 1–8 GPUs.

on a handful of SMALL graphs where we demonstrated that considering reasonable #batches can increase the scalability relative to the default scenario (see §IV-B). 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.

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 III highlights the performance impact of the GPU generation by comparing contemporary NVIDIA™ "Ampere" 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 III
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×

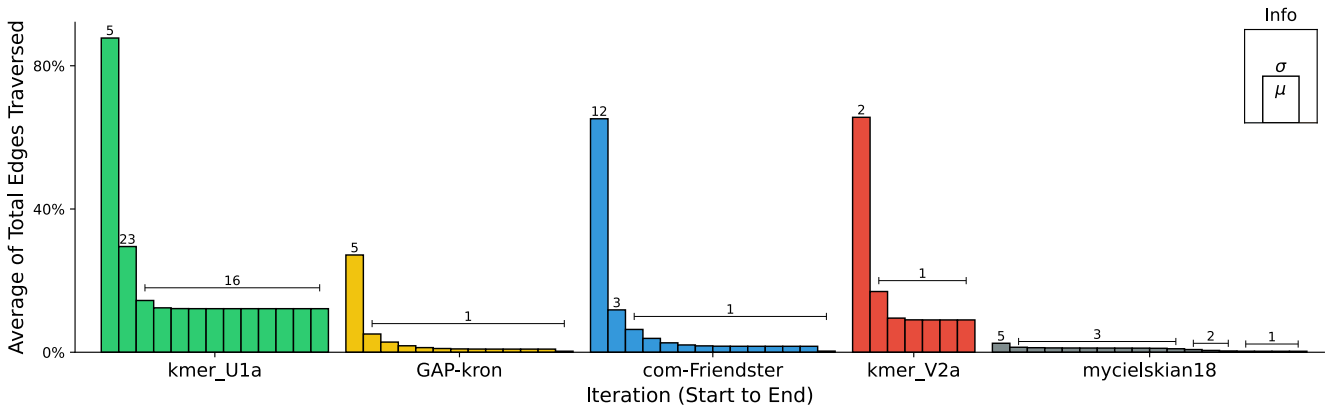


Fig. 8. Mean and standard deviation of % of edges accessed by warps on *pointing* phase iteration of LD-GPU—for 90% of the iterations, less than 20% of the edges are accessed.

We assess the impact of the GPU interconnect, PCIe vs. NVLink (SXM4), in Fig. 9. Foley, et al. [21] 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

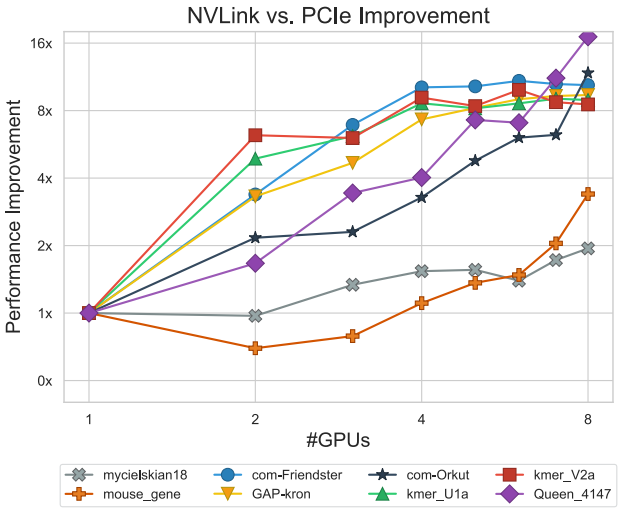


Fig. 9. Execution time speedup of NVLink vs PCIe for data transfer and multi-GPU communication for LD-GPU.

observe the outlier input mouse_gene (smallest graph), which actually demonstrates relatively mild and stable collective communication overhead up to 4 GPUs (see Fig. 5). Hence, we expect non-trivial end-to-end improvements with enhanced GPU interconnects across the vendor generations for larger graphs. Fig. 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.

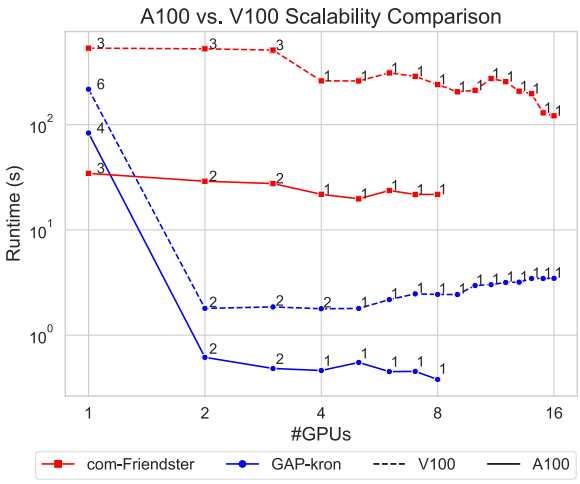


Fig. 10. LD-GPU scalability on the dense-GPU systems with annotated #batches: DGX-2 (16 V100s) vs. the DGX-A100 (8 A100s).

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

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. 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, a majority of the edge traversals are performed in the first iteration of matching (thus, the 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 work on the second iteration. Meanwhile, there are cases such as GAP-kron which exhibit a relatively even distribution of warp-edge work throughout the iterations. This variation is critical to GPU efficiency, as an uneven distribution of edges across warps can signifiy poor device utilization. For our purposes, however, the total amount of work is reduced per iteration as pairings are added to the matching. Thus, we capture the overall variations in the densities of the warp-edge work in Fig. 8 and further study the device occupancy, as discussed next.

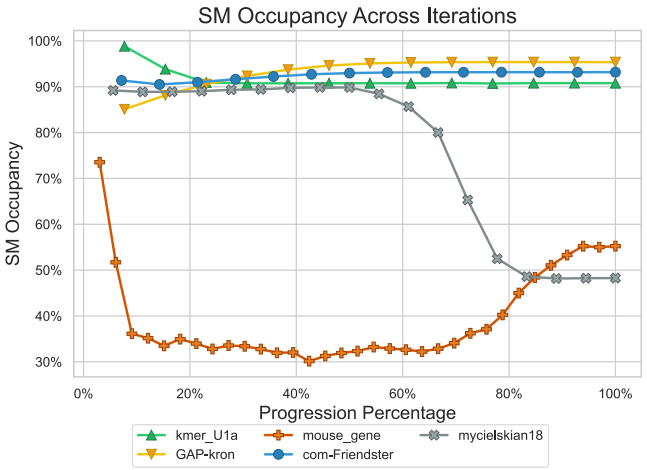


Fig. 11. GPU Streaming Multiprocessor (SM) occupancy (Y-axis, higher is better) as reported through NVIDIA Nsight profiler per iteration of LD-GPU (X-axis shows iteration progression in terms of %).

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. 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.

D. 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 [30], [43] and the GPU Suitor (SR-GPU) algorithm in [32], [33]. 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 [28]. SR-OMP results are collected using 256 CPU threads while SR-GPU results are collected on a single NVIDIA™ A100 GPU.

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 I. 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 I 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 mid-size 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).

Single GPU performance comparison: We aim to 1) leverage multiple GPUs, and 2) solve large-scale matchings; consequently, in most cases, the best execution times obtained use multiple GPUs. SR-GPU adopted load redistribution by varying vertices-per-warp, which can only work in small or regular degree graphs. So, for some graphs, LD-GPU is relatively expensive on a single GPU, as shown in Table IV. However, graphs are multifarious, and fixing vertices-per-warp is not a general recipe for enhancing single GPU performance (rather poses challenges for multi-GPU), as evident from 3/8 cases where LD-GPU is better or competitive, shown in Table IV.

TABLE IV
SINGLE GPU RUNTIME COMPARISON

Graphs	Runtime (s)	
	LD-GPU	SR-GPU
com-Friendster	0.725	**0.661**
Queen_4147	0.027	**0.008**
mycielskian18	**0.019**	0.025
HV15R	**0.045**	0.047
com-Orkut	1.274	**0.036**
kmer_U1a	0.193	**0.048**
kmer_V2a	0.131	**0.058**
mouse_gene	**0.013**	0.016

Comparisons with NVIDIA™ RAPIDS™ cuGraph: Recently, NVIDIA RAPIDS cuGraph has released a weighted approximate matching implementation (following [29], which builds on locally dominant algorithm by Preis [36], see §II-C) for distributed-memory multi-GPU systems. However, current multi-GPU implementation of cuGraph (over modern C++) is experimental, considering a process-per-GPU model, requiring a process to load an entire graph (in a native format such as matrix-market) and then filtering the subgraphs for specific processes, increasing the overall memory usage. For this reason, it is only practical to compare medium-sized graphs with multi-GPU cuGraph (on reasonable #GPUs to optimize the communication overheads). Also, due to the software dependencies mandated by latest cuGraph, we used different versions of the compilers and programming systems compared to the baseline experiments. Specifically, we used GCC/12.2.0, CUDA/12.1, OpenMPI/4.1.4 (CUDA-aware) and miniconda/24.4.0, to build cuGraph using the `conda` package manager[2]. We use the same software versions to build LD-GPU for appropriate comparison. Table V shows the results (for maximal weighted matching, excluding graph loading/processing which can be non-trivial) on 4 A100 GPUs, relative to LD-GPU using a single batch. LD-GPU is an order of magnitude faster than cuGraph; we anticipate this is due to differences in the underlying communication abstractions. Notably, cuGraph uses RAFT Comms (built on top of MPI)[3], while we use NCCL over CUDA streams.

TABLE V
CUGRAPH RUNTIME COMPARISON ON 4 GPUS

Graphs	Runtime (s)	
	LD-GPU	cuGraph
Queen_4147	**0.018**	7.978
mycielskian18	**0.058**	3.055
com-Orkut	**1.218**	32.385
kmer_U1a	**0.152**	2.383
kmer_V2a	**0.202**	2.579

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 VI. 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 VI
MEGA-MATCHING EDGES PER SECOND (HIGHER IS BETTER).

Graphs	FoM (MMEPS)	
	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

V. CONCLUDING REMARKS

In this paper, we devise a parallel algorithm for locally dominant maximal weighted graph matching for multiple GPUs on single node NVIDIA DGX™ platforms. We leverage vendor-optimized collective communication libraries for data transfer

[2]https://docs.rapids.ai/api/cugraph/nightly/installation/getting_cugraph/#conda

[3]https://docs.rapids.ai/api/raft/nightly/cpp_api/mnmg/

between GPUs over NVLink interconnect, bypassing the host CPU. We introduce batching to mitigate limited GPU memory, increasing graph data sizes, and nontrivial graph partitioning problems, the trio behind a myriad of out-of-memory issues. Our batching method defines the working set size on GPUs, providing a mechanism to balance independent work and synchronization. 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.

Towards the development of distributed matching schemes targeting higher quality guarantees or similar improvements, we conclude by highlighting the synchronization overheads prevalent in parallel graph analytics. LD-GPU performs asynchronous processing whenever possible, while adopting a level-synchronous approach through explicit batch processing for correctness. For more complex matching schemes, balancing the parallel efficiency, accuracy and synchronization costs will be relevant in achieving sustainable strong scalability on the next generation of HPC platforms.

ACKNOWLEDGMENTS

This research is in parts supported by the National Science Foundation under Grant No. 2047821, the U.S. DOE ExaGraph project; Data-Model Convergence Initiative (DMC) and the LDRD initiative at the Pacific Northwest National Laboratory (PNNL). PNNL is operated by Battelle Memorial Institute under Contract DE-AC06-76RL01830. We would also like to thank Dr. Tim Carlson at PNNL and the PNNL Research Computing staff for their outstanding support.

REFERENCES

[1] Ravindra K. Ahuja, Thomas L. Magnanti, and James B. Orlin. *Network flows: theory, algorithms, and applications*. Prentice-Hall, Inc., USA, 1993.

[2] David Avis. A survey of heuristics for the weighted matching problem. *Networks*, 13(4):475–493, 1983.

[3] Ariful Azad, Aydin Buluç, Xiaoye S Li, Xinliang Wang, and Johannes Langguth. A distributed-memory algorithm for computing a heavy-weight perfect matching on bipartite graphs. *SIAM Journal on Scientific Computing*, 42(4):C143–C168, 2020.

[4] S. Belongie, J. Malik, and J. Puzicha. Shape matching and object recognition using shape contexts. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 24(4):509–522, 2002.

[5] Massimo Bernaschi, Alessandro Celestini, Pasqua D'Ambra, and Flavio Vella. Multi-GPU aggregation-based AMG preconditioner for iterative linear solvers, 2023.

[6] Marcel Birn, Vitaly Osipov, Peter Sanders, Christian Schulz, and Nodari Sitchinava. Efficient parallel and external matching. In *European Conference on Parallel Processing*, pages 659–670. Springer, 2013.

[7] Paolo Boldi and Sebastiano Vigna. The WebGraph framework I: Compression techniques. In *Proc. of the Thirteenth International World Wide Web Conference (WWW 2004)*, pages 595–601, Manhattan, USA, 2004. ACM Press.

[8] Rainer Burkard, Mauro Dell'Amico, and Silvano Martello. *Assignment Problems*. Society for Industrial and Applied Mathematics, USA, 2009.

[9] Jie Chen, Robert G. Edwards, and Weizhen Mao. Graph contractions for calculating correlation functions in lattice qcd. In *Proceedings of the Platform for Advanced Scientific Computing Conference*, PASC '23, New York, NY, USA, 2023. Association for Computing Machinery.

[10] Han-Yi Chou and Sayan Ghosh. Batched graph community detection on gpus. In *Proceedings of the International Conference on Parallel Architectures and Compilation Techniques*, PACT '22, page 172–184, New York, NY, USA, 2023. Association for Computing Machinery.

[11] Pasqua D'Ambra, Fabio Durastante, S M Ferdous, Salvatore Filippone, Mahantesh Halappanavar, and Alex 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)*, pages 59–67, 2023.

[12] Timothy A. Davis and Yifan Hu. The university of florida sparse matrix collection. *ACM Trans. Math. Softw.*, 38(1), dec 2011.

[13] Balázs Dezső, Alpár Jüttner, and Péter Kovács. LEMON–an open source C++ graph template library. *Electronic notes in theoretical computer science*, 264(5):23–45, 2011.

[14] Doratha E Drake and Stefan Hougardy. A simple approximation algorithm for the weighted matching problem. *Information Processing Letters*, 85(4):211–213, 2003.

[15] Ran Duan and Seth Pettie. Linear-time approximation for maximum weight matching. *Journal of the ACM (JACM)*, 61(1):1–23, 2014.

[16] I. S. Duff and J. Koster. On algorithms for permuting large entries to the diagonal of a sparse matrix. *SIAM Journal on Matrix Analysis and Applications*, 22(4):973–996, 2001.

[17] Jack Edmonds. Maximum matching and a polyhedron with 0, 1-vertices. *Journal of research of the National Bureau of Standards B*, 69(125-130):55–56, 1965.

[18] Jack Edmonds. Paths, trees, and flowers. *Canadian Journal of mathematics*, 17:449–467, 1965.

[19] EETimes. Nvidia's Blackwell Offers FP4, Second-Gen Transformer Engine, 2024.

[20] Bas O Fagginger Auer and Rob H Bisseling. A GPU algorithm for greedy graph matching. *Facing the Multicore-Challenge II: Aspects of New Paradigms and Technologies in Parallel Computing*, pages 108–119, 2012.

[21] Denis Foley and John Danskin. Ultra-performance pascal GPU and NVLink interconnect. *IEEE Micro*, 37(2):7–17, 2017.

[22] Oded Green and David A Bader. cuSTINGER: Supporting dynamic graph algorithms for gpus. In *2016 IEEE High Performance Extreme Computing Conference (HPEC)*, pages 1–6. IEEE, 2016.

[23] Mahantesh Halappanavar, John Feo, Oreste Villa, Antonino Tumeo, and Alex Pothen. Approximate weighted matching on emerging manycore and multithreaded architectures. *The International Journal of High Performance Computing Applications*, 26(4):413–430, 2012.

[24] Ismayil Ismayilov, Javid Baydamirli, Doğan Sağbili, Mohamed Wahib, and Didem 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, page 192–202, New York, NY, USA, 2023. Association for Computing Machinery.

[25] Sylvain Jeaugey. Nccl 2.0. In *GPU Technology Conference (GTC)*, volume 2, page 23, 2017.

[26] Jeremy Kepner and John Gilbert. *Graph algorithms in the language of linear algebra*. SIAM, 2011.

[27] H. W. Kuhn. The hungarian method for the assignment problem. *Naval Research Logistics Quarterly*, 2(1-2):83–97, 1955.

[28] LEMON Contributors. LEMON: Library for efficient modeling and optimization in networks. https://lemon.cs.elte.hu/pub/doc/latest-svn/index.html. Accessed: 2 April, 2024.

[29] Fredrik Manne and Rob H Bisseling. A parallel approximation algorithm for the weighted maximum matching problem. In *International Conference on Parallel Processing and Applied Mathematics*, pages 708–717. Springer, 2007.

[30] Fredrik Manne and Mahantesh Halappanavar. New effective multithreaded matching algorithms. In *2014 IEEE 28th International Parallel and Distributed Processing Symposium*, pages 519–528. IEEE, 2014.

[31] Briance Mascarenhas and Kartikeye Puranam. Analysis of the medical residency matching algorithm to validate and improve equity. *PLOS ONE*, 18(4):1–11, 04 2023.

[32] Md. Naim, Fredrik Manne, Mahantesh Halappanavar, Antonino Tumeo, and Johannes Langguth. GPU suitor. https://hpc.pnl.gov/people/hala/suitor.html. Accessed: 2 April, 2024.

[33] Md. Naim, Fredrik Manne, Mahantesh Halappanavar, Antonino Tumeo, and Johannes Langguth. Optimizing approximate weighted matching on Nvidia Kepler K40. *2015 IEEE 22nd International Conference on High Performance Computing (HiPC)*, pages 105–114, 2015.

[34] Seth Pettie and Peter Sanders. A simpler linear time 2/3-ε approximation for maximum weight matching. *Information Processing Letters*, 91(6):271–276, 2004.

[35] Alex Pothen, SM Ferdous, and Fredrik Manne. Approximation algorithms in combinatorial scientific computing. *Acta Numerica*, 28:541–633, 2019.

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

[37] Amir Hossein Nodehi Sabet, Zhijia Zhao, and Rajiv Gupta. Subway: Minimizing data transfer during out-of-gpu-memory graph processing. In *Proceedings of the Fifteenth European Conference on Computer Systems*, pages 1–16, 2020.

[38] Dipanjan Sengupta, Shuaiwen Leon Song, Kapil Agarwal, and Karsten Schwan. Graphreduce: processing large-scale graphs on accelerator-based systems. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, pages 1–12, 2015.

[39] Francesco Sgherzi, Alberto Parravicini, and Marco 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)*, pages 1259–1263, 2022.

[40] Xuanhua Shi, Zhigao Zheng, Yongluan Zhou, Hai Jin, Ligang He, Bo Liu, and Qiang-Sheng Hua. Graph processing on GPUs: A survey. *ACM Computing Surveys (CSUR)*, 50(6):1–35, 2018.

[41] Kasia Świrydowicz, Eric Darve, Wesley Jones, Jonathan Maack, Shaked Regev, Michael A Saunders, Stephen J Thomas, and Slaven Peleš. Linear solvers for power grid optimization problems: a review of GPU-accelerated linear solvers. *Parallel Computing*, 111:102870, 2022.

[42] Wenyong Zhong, Jianhua Sun, Hao Chen, Jun Xiao, Zhiwen Chen, Chang Cheng, and Xuanhua Shi. Optimizing graph processing on GPUs. *IEEE Transactions on Parallel and Distributed Systems*, 28(4):1149–1162, 2016.

[43] Ümit V. Çatalyürek, Florin Dobrian, Assefaw Gebremedhin, Mahantesh Halappanavar, and Alex Pothen. Distributed-memory parallel algorithms for matching and coloring. In *2011 IEEE International Symposium on Parallel and Distributed Processing Workshops and Phd Forum*, pages 1971–1980, 2011.