# GnnOne: A Unified System Optimizations for GNN Kernels

Yidong Gong
William & Mary
ygong07@wm.edu

Pradeep Kumar
William & Mary
pkumar@wm.edu

## ABSTRACT

Graph Neural Networks (GNN) involve two basic sparse kernels, SDDMM and SpMM, on which all GNN models could be built. Prior works have explored piecemeal solutions by using different storage formats and computation paradigms, resulting in excess memory consumption, and have not yet realized their full potential. This paper, called GnnOne, studies these two basic sparse kernels in GPU and shows that they can be built on the same system design principle of data load being the limiting factor irrespective of their computing paradigms. Hence GnnOne presents a unified two-stage data-load design that provides greater performance through novel techniques of data-load balancing, data-load optimizations, and data-reuse. Such a unified design also enables the usage of a single sparse storage format to increase productivity, memory saving, and reduce maintenance. Evaluations show that the proposed system achieves an average speedup of 6.25× and 6.02× for SpMM and SDDMM over many prior works for different feature lengths. For GNN training, GnnOne achieves 2.01× average speedup over dgNN, 2.28× average speedup over DGL on 3 different GNN models.

## CCS CONCEPTS

• **General and reference** → **Performance**; • **Computing methodologies** → **Neural networks**.

## KEYWORDS

Graph Neural Networks, GPU, Performance

**ACM Reference Format:**
Yidong Gong and Pradeep Kumar. 2024. GnnOne: A Unified System Optimizations for GNN Kernels. In *The 33rd International Symposium on High-Performance Parallel and Distributed Computing (HPDC '24), June 3–7, 2024, Pisa, Italy*. ACM, New York, NY, USA, 13 pages. https://doi.org/10.1145/3625549.3658655

## 1 INTRODUCTION

A sparse matrix or graph is a widely used data model in Graph Neural Networks [4, 15, 23, 32, 48–50] (GNN) to boost the performance of deep learning (DL) training on sparse data among different applications, such as social media, biology, chemistry, recommendation systems, and knowledge base completions [8, 14, 24, 28, 29, 33, 45, 51]. Prior works [35] have established SDDMM (sampled dense dense matrix multiplication) and SpMM (sparse matrix dense matrix multiplication) as the *basic building blocks* (sparse kernels) to construct any GNN model as discussed in-depth in §2. For example, when GNN calls SpMM in the forward computation, the back-propagation calls SpMM and SDDMM.

A graph consists of vertex set $V$ and edge set $E$, where $E$ denotes the non-zero element of the sparse matrix (more information is in §2). SDDMM and SpMM produce output at the *edge-level* and *vertex-level* respectively. Hence they follow different computation paradigms: SDDMM naturally aligns with edge-centric computation, while SpMM is vertex-centric. These distinct paradigmatic differences between these two kernels have so far inspired piecemeal solutions resulting in very contrasting SpMM and SDDMM designs, presenting a challenge in efficiently combining them as part of the same workflow of GNN training.

To illustrate, consider the existing high-level approaches as listed next. 1) Optimize SpMM kernel only [19, 20, 37], sometimes even proposing a custom storage format for the same, without giving attention to their applicability to SDDMM. The custom storage format also creates further hurdles in their integration into popular DL or GNN frameworks, such as DGL [35] or Pytorch-Geometric [9] (PyG), which do not support custom formats. 2) Downgrade SDDMM into a vertex-centric variant [5, 39, 47] so that a single storage format, such as compressed sparse row (CSR), could be used in the whole GNN workflow. However, the downgrade impedes the application of potential optimizations to SDDMM thereby achieving only non-optimal SDDMM performance. 3) Individually align SpMM and SDDMM, such as DGL [35] uses CSR and coordinate list (COO) formats respectively. Similarly, Sputnik [11] uses a custom format for SpMM but uses CSR for SDDMM. Such approaches not only consume multiple storage formats which leads to excessive memory consumption but also make optimization on two different problems which let us believe that SpMM and SDDMM are fundamentally different.

Clearly, none of the above approaches are ideal in an end-to-end GNN training setup as they either compromise on SDDMM optimization or increase the system complexity in the integration, maintenance, and optimization. Also, SDDMM being a new sparse kernel, was not studied previously in scientific computation or in classical graph analytics work, its support is only recently been introduced in vendor libraries such as Cusparse [1] by Nvidia. However, it is still excessively slow as our evaluation (§5) shows.

All these observations lead us to believe that SDDMM has neither been comprehensively studied individually nor along with SpMM as part of GNN training. This points to a critical research question: *are differences in computation paradigms of SDDMM and SpMM fundamental enough to design piecemeal optimizations or can we design common optimizations that can allow high-performance in both individual kernels without sacrificing the performance of either*?

In response, we propose **GnnOne**, a system dedicated to a comprehensive study of sparse kernels, and identifying fundamental differences in their computation paradigm. GnnOne shows that


*HPDC '24, June 3–7, 2024, Pisa, Italy*

ACM ISBN 979-8-4007-0413-0/24/06
https://doi.org/10.1145/3625549.3658655

those differences, despite being fundamental, are not responsible for current piecemeal solutions. GnnOne turns to a *data-load centric* design based on our novel observations to achieve a unified design. To be more specific, a) the fundamental trade-off is only in the reduction stage of the computation (§3) and *not in their data-load stages*; and b) sparse kernels should be *dominated by the data-load performance and not by actual computation* due to irregularity introduced by the sparsity irrespective of their computation paradigm. Hence, the sparse kernels– performing similar data-load and being dominated by it– are unified at the data-load design by GnnOne. For their common optimizations, GnnOne then revisits some of the known problems in sparse kernels from data-load perspective, achieving workload balancing through *data-load balance*, and further optimization by minimizing data-load volume through *data reuse*.

GnnOne achieves unified design and optimization through a novel **two-stage data-load** technique. *Stage 1* loads NZEs and corresponding edge-level features using a fully balanced edge-parallel strategy and caches them explicitly for their reuse in Stage 2. *Stage 2* loads vertex-level features of cached NZEs using another balanced data-load strategy, improves data-load performance, and introduces their data-reuse using a novel **symbiotic thread scheduler**. The scheduler is developed from the insight gathered from our in-depth study of sparse kernels to achieve their symbiotic co-existence in the GNN training.

The separation of data-load into two stages allows their independent optimization in many novel ways. a) Stage 1 uses GPU shared memory for caching non-zero elements and edge-level features whose optimum size can be determined *independent of the neighborhood size* in the dataset. The caching is new in SDDMM while its independence is new in SpMM compared to current systems. b) Our new insights point out that reduction fundamentally impacts the data-load performance through the memory barrier usage and excessive usage of shared memory or registers (§3.2). The symbiotic scheduler through *thread-grouping*, and a novel *Consecutive scheduling policy* optimizes the data-load performance in Stage 2, introduces huge data-reuse, and enables a more thread-local reduction in both sparse kernels.

The unified design and the symbiotic co-existence of sparse kernels enable their implementation based on a single sparse storage format as long as reduction can access the row ID of every non-zero element efficiently. This observation gives the standard COO format a head start. However, our optimization can be applied to other formats and explain the trade-off involved in selecting the COO format compared to CSR or custom storage formats.

Performance evaluations show that for SDDMM, our COO-based GnnOne achieves an average 6.54×, 4.17×, and 6.38× speedup compared to CSR-based Featgraph, COO-based DGL, and CSR-based dgSparse library [3] which is used in dgNN[47] respectively among different feature lengths, and one to two orders of average speedup compared to CuSparse, and Sputnik. For SpMM, our COO-based GnnOne achieve 9.80×, 3.20×, 4.86×, and 1.73× speedup compared to CSR-based Ge-SpMM [19], CSR-based Cusparse, and the workload balanced solutions(custom format-based) by GNNAdvisor [37], and Huang et. al [20] respectively for various feature lengths.

We also show that GnnOne, without any kernel fusion, achieves 2.01× speedup over dgNN [47], a highly fused GNN system, for GAT [32]. Compared to DGL, GnnOne gains 1.89×, 1.27×, and 3.68× average speedup for GCN [23], GIN [40], and GAT [32].

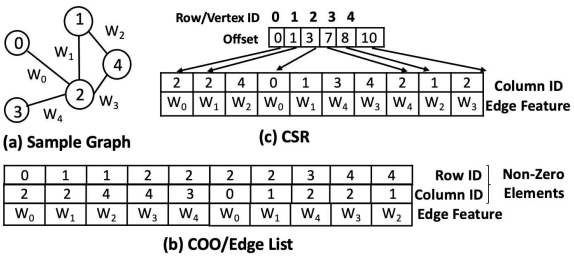


**Fig. 1: Illustration of sparse matrix storage formats, showing edge-level features. The edge-level features are not static like the corresponding sparse matrix topology. Vertex-level features have been omitted for brevity.**

Reliance on standard sparse format (COO) allows GnnOne to be integrated with popular GNN frameworks such as DGL and PyG that support COO format. Moreover, it can also be part of many scientific computing workflows as vendor libraries such as Cusparse support SpMM using the COO format. It can also influence its SDDMM performance which has been far below expectations and is yet to support the standard COO format. GnnOne not only have far-reaching impacts on GNN system optimizations, maintenance, and productivity due to the unified property that we discuss but also on sparse models, another user of such sparse kernels.

The remainder of the paper is organized as follows. The background is presented in §2, motivation and overview in §3, detailed design and discussion in §4, and evaluations in §5. Other related works and discussions are presented in §6, and we conclude in §7.

## 2 BACKGROUND

**GPU.** GPU has become a very important accelerator for deep learning training. A GPU is made up of many Streaming Multiprocessors (SM). A program in GPU is called *kernel* and is executed by a group of *thread blocks*, also called *CTA* (cooperative thread array). Each CTA is configured with a specific number of threads such as 64, 128, 256, 512, etc. At any time, a number of CTA may be active on the same GPU SM, and there count per SM is called *GPU Occupancy*. A higher occupancy is preferred as it can hide the long latency instructions, such as data-load instruction.

CTA threads are grouped in warps– 32 threads– that execute in single instruction multiple threads (SIMT) fashion. *Coalesced memory* can only be achieved if each thread of the warp issue data load on consecutive locations. Coalescing results in faster data movement from the global memory to its register for computation.

Each CTA also has a fixed amount of programmable hardware cache (e.g., 64KB), which may optionally be used as shared memory in CTA, where a program can modify or access its contents directly. However, writing data to GPU's shared memory requires a memory barrier before it can be read for further processing.

**Graph Storage Format.** Sparse data that we focus on in this work could be modeled using sparse matrices or graphs. A graph has a set of vertices $|V|$, and connections between vertices are represented as an edge set $|E|$. A sample sparse matrix/graph is shown in Fig. 1(a),

where there are only a few non-zero columns in any row. A row corresponds to a vertex, while the count of non-zero elements (**NZE**) is equal to the edge-count ($|E|$) in the corresponding graph.

**Terminology.** In this paper, we mostly use sparse matrix terminology but do use graph-related terminologies for types of tensors and parallelism to avoid name collision. For tensors (features), we label them as *vertex-level* or *edge-level*. For example, $W$ of size $|E| \times |1|$ is an edge-level tensor in Fig. 1(a) where each NZEs has a scalar feature. Similarly, a dense matrix of size $|V| \times |F|$ would be called a vertex-level tensor, where each vertex has a vector feature of size $|F|$. Similarly, for parallelism, we label them as vertex-parallel or edge-parallel. The *row-length* is equivalent to vertex degree.

Fig. 1 shows different storage formats of the sparse matrix that only store non-zero elements(**NZE**) of the sparse matrix. Compressed Sparse Row (**CSR**) uses three arrays: The first array contains the column ID of every NZE of the matrix. The *offset array* contains offsets to the NZE array for each row. The third array is the edge-level tensor. Coordinate List (**COO**) format consists of two arrays, the first array contains the tuple of row and column IDs, while the second array is the edge-level tensor. Cusparse defines the COO to be stored in the CSR way [2].

**Custom Storage Format.** Any format that does not conform to the above-discussed form, we call them custom format. For example, any additional metadata on top of the CSR format makes it a custom format. Though we will be comparing against custom formats for performance, our goal is to rely on standard formats.

**Basic Sparse Kernels.** Sparse kernels involve sparse and dense matrices. *SDDMM* or sampled Dense Dense Matrix multiplication ($W \leftarrow \mathrm{A} \odot (\mathrm{XY}^T)$) introduces sparsity to the product of two dense matrices (or vertex-level tensors) X and $\mathrm{Y}^T$, each of size $|V| \times |F|$, using sparse adjacency matrix $A$. Thus the result $W$ is an edge-level tensor of same size as A ($|E| \times 1$). Hence, the dot product happens across (vertex) feature dimensions. *SpMM* or Sparse matrix and dense matrix multiplication ($Y \leftarrow AX$) is the matrix multiplication operation between a sparse matrix (or edge-level tensor) A of size $|E| \times 1$ and a dense matrix (or vertex-level tensor) X of size $|V| \times |F|$ resulting in a dense matrix (or a new vertex-level tensor) Y of size $|V| \times |F|$.

Some GNN models, such as GCN, may not need SDDMM in its forward and backward computation, while other GNN models still need both SDDMM and SpMM in their computation, so optimizing them either individually or together is still vital.

**Vertex-Parallel and Edge-Parallel.** When computation units are divided equally among rows (or vertices), it is called vertex-parallel method. E.g., a warp assigned to each row for performing SpMM. However, each warp performs a varying amount of work due to varying row lengths, leading to severe workload imbalance. On the other hand, the *edge-parallel method* allocates an equal amount of NZEs to each warp. Such solutions are known as *non-zero split* in the SpMV domain, and imparts good workload balancing amid the need for inter-thread communication for reduction implementation.

**Feature-Parallel Method.** From *feature-parallel* [42] perspective, warp threads are deployed along the vertex-feature dimension to achieve coalesced data load from matrices X and $Y^T$ in GPUs, i.e., dense matrices are read row-wise.

# 3 MOTIVATION AND OVERVIEW

## 3.1 Optimizing SDDMM and SpMM Together

SDDMM and SpMM kernels and their variants serve as the basic building blocks for all the sparsity needs of GNN [35]. Optimizing them can contribute to all kinds of GNN models. Several GNN models, such as GAT [32], GaAN [49] need to include both SDDMM and SpMM together for training and inference. However, SDDMM have different computation patterns than SpMM, SDDMM computation pattern is derived from feature dimension reduction, i.e. reduction is column-based; while SpMM computation is from neighborhood dimension reduction, i.e. reduction is row-based.

Because of the above differences, numerous efforts have been made to optimize the SDDMM and SpMM independently tailored to the specific computation pattern of each kernel, and this has often resulted in conflicting designs and/or sub-optimal performance of sparse kernels. For example, DGL designed kernels are very different from each other, e.g. it uses COO for (edge-parallel) SDDMM achieving a good workload balancing, however, it uses CSR for SpMM leading to excessive memory consumption for graph storage in GNN training, while we present soon that workload balancing alone does not bring DGL SDDMM desired results. However, newer GNN systems [39, 47] have ignored workload-balancing achieved by DGL SDDMM by sticking to CSR format for both the kernels by implementing a vertex-centric variant. Other GNN works, such as Ge-SpMM [19] and GNNAdvisor [37] has only offered SpMM to support only limited GNN models, and hence it is not clear if the same design can provide an optimal solution for SDDMM in order to support wider GNN models. A more in-depth study of prior works is presented in §6.

In this work, we focus on sparse kernel design that does not compromise the performance of any sparse kernels, specifically on the design of SDDMM and its co-existence with SpMM as part of GNN workflow. We comment that the community has not understood this co-existence fully. We establish clear facts to prove that SDDMM and SpMM can be built from the same fundamentals so that their design will not conflict with each other while achieving the desired performance goal. To corroborate this, we now present two general observations: the first is about similarities and differences between SDDMM and SpMM, and the second is about the general trend for sparse kernel computation on modern hardware. These two present us with novel opportunities for the co-existence of both the sparse kernels and to achieve common optimizations.

**Observation #1: Only Reduction is Fundamentally Different.** Many steps are *similar* for these two sparse kernels, or only minor differences exist. E.g., loading NZEs and the vertex-features of their column ID are the same. The minor difference is that SpMM additionally loads edge-features of NZE, while SDDMM additionally fetches vertex-feature of the row ID of each NZE. Both of them produce an equal count of individual dot products.

The *fundamental difference* exists only for reduction: SDDMM performs reductions of the dot products along the vertex-feature dimension to generate output at the edge level, while SpMM does it along the neighborhood dimension of the sparse matrix to generate output at the vertex level.

**Observation #2: Data Load ≫ Actual Compute.** Sparse kernel computations, like many other computations, involve three steps:

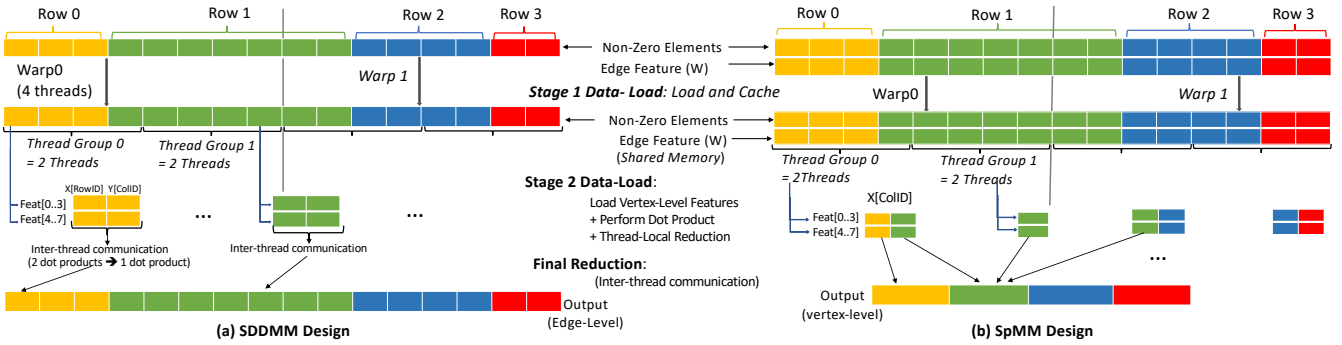


**Fig. 2: Illustration of the proposed unified design for SDDMM and SpMM computation: each warp (say, 4 threads) is shown to handle 8 NZE, and 8 is feature-length for brevity. The 8 NZEs are divided among two thread groups using the proposed Symbiotic scheduling policy, where they work on 4 consecutive NZEs. Each thread fetches 4 consecutive vertex-level features from the dense matrix.**

bringing data (NZEs and their features) from the *device memory* to the registers of the computation unit, and then doing *actual computation* (dot product and reduction), followed by updating the result to the output. Modern GPU processors are very fast and can work in SIMT fashion resulting in faster computation. However, memory speed (device memory to-and-fro register) has not kept pace with processor performance, and is popularly know as "memory wall" in computer architecture field. Further, the irregularity introduced due to sparsity makes data-load the dominant cost in GPU.

Clearly, both sparse kernels show a common data-load pattern that should dominate the overall performance. We do prove this claim empirically later in Fig. 11. Hence, this motivates *system design of GnnOne to focus on abstracting out the common data-load pattern using an unified data-load design, followed by balancing and optimizing it while also introducing data reuse.*

## 3.2 GnnOne: Challenges and Overview

**Unified Data-Load.** GnnOne abstracts out the common data-load pattern and designs a *unified data-load stage* whose optimization is the main research goal to benefit both the sparse kernels. Viewing the sparse kernels from the prism of data-load performance, the known problem of workload-imbalance becomes a data-load imbalance problem, while optimizing the performance of sparse kernels requires data-load optimization including introducing data-reuse. Thus the final goal is the study of various aspect of unified data-load optimization.

**Advantages.** This unified data-load design is very relevant as a number of optimizations that have been proposed for SpMM (or even SpMV) can become relevant for SDDMM, while the techniques that we propose for SDDMM becomes relevant for SpMM. Data-load unification also have other wide ranging advantages. It may also enable the deployment of a single storage format in both sparse kernels leading to significant memory saving in GNN, such as in DGL which relies on two different formats; and selecting a standard graph storage format (COO) ensures its compatibility with standard libraries, such as Cusparse. The unified design also leads to less development and maintenance cost as each data-load optimization technique is likely going to be applicable to both kernels, thereby leading to code modularization and code reuse. Finally, vendor libraries such as CuSparse can quickly reuse their SpMM codebase and extend it to SDDMM rather than spending additional time on design thinking, as their current SDDMM is extremely slow, which we believe is based on a different design than its SpMM.

**Challenge: Data-load balancing alone is not sufficient.** Achieving a balanced data-load alone does not imply the best data-load or kernel performance. Hence merely borrowing the idea from prior works focused on balanced data-load alone does not guarantee an optimal data-load performance.

To illustrate for *SDDMM*, DGL does use edge-parallel design, which achieves data-load balance. However, it is even slower than the CSR-based SDDMM of dgSparse library [3], which is used by GNN systems such as dgNN [47]. E.g., when the feature-length is 32, DGL is slower by an average of 2.01× than dgSparse on datasets listed in Table 1. More results are presented in §5. Similarly, for *SpMM*, we can consider the well-known nonzero-split (edge-parallel) techniques pioneered by SpMV [6, 27] to achieve data-load balance. Unfortunately, Yang et al [42], which extended this design to SpMM, concludes that a nonzero-split SpMM is slower than their vanilla vertex-parallel SpMM for the majority of their datasets.

It is no surprise that such results have established the non-optimal vertex-parallel variants as the state-of-the-art while discarding the right approach of data-load balanced solutions. As it stands today, edge-parallel SDDMM by DGL and edge-parallel SpMM by Yang et al. are not the state-of-the-art solutions.

**Overview and Contributions.** Better performance from the unified data-load stage requires not only a data-load balanced design but also investigating the factors that indirectly affects its performance. In this work, we perform such investigations, and accordingly propose novel techniques to improve the data-load performance as outlined next as our contributions:

• **Reduction indirectly impacts Data-Load.** Our analysis reveals a *fundamental impact* that reduction stage has indirectly on data-load performance which remains unknown to the community.

Firstly, *reduction can impact data-load performance through the memory barrier.* To illustrate, consider 32 as feature-length in SDDMM, where individual threads of the warp are mapped along feature dimensions, i.e., 1 thread handles 1 vertex-feature before doing the reduction. Hence, a thread of the warp performs just 1

load of vertex-level feature before doing 5 rounds ($log_2^{32}$) of inter-thread communication [18]. The inter-thread communication also introduces a memory barrier, enforcing an ordering constraint on memory operations issued before and after the barrier instruction. Hence, the compiler cannot issue more than 1 outstanding feature load instruction, forcing the thread to wait after issuing every feature load. This impacts instruction-level parallelism (ILP), resulting in slower data-load performance.

Secondly, *reduction can impact data-load performance through excessive consumption of shared-memory or register*. To illustrate, consider nonzero-split from SpMV [27], which Yang et al [42] extended for SpMM as is. It continued materializing the individual dot products in SpMM in registers for *all NZE* that are assigned to a warp, and reduction is performed at the very end. This materialization leads to excessive usage of registers per thread, e.g. 32× than SpMV if the feature-length is 32. Yang et al claims that large register usage is the main reason for slowdown of their non-zero split based SpMM. Our investigation confirms that large register usage *lowers* GPU occupancy, thereby GPU cannot launch many concurrent CTA to allow hiding data-load latency with the computation. Hence, the GPU cannot issue sufficient data-load instructions leading to smaller ILP and hence poor data-load performance.

• **Two-Stage Data-Load and Symbiotic Thread Scheduler.** GnnOne propose a *two-stage data-load* and *symbiotic thread scheduler* to import data-load balancing and optimization to the unified data-load design by exploring factors that are responsible for poor data-load performance. The working flow is shown in Fig. 2.

*Stage 1* of unified data-load achieves a fully balanced data-load of NZEs and the corresponding edge features, if needed, using the edge-parallel method resulting in complete memory coalescing. This stage stores them in the shared memory of the GPU so that they can be reused in Stage 2. The advantage of separation is that GnnOne can determine the cache size in Stage 1 based on hardware (GPU) characteristics such as shared memory size, and memory barrier that is applied before reading from shared memory, and is fully *independent of row-length*– a novel contribution that is applicable for both SDDMM and SpMM.

*Stage 2* loads the vertex-level features of the cached NZEs from device memory to registers, while also performing the dot product as per individual kernel needs. The scheduler assigns *one thread to load many consecutive vertex-level features* so that the impact of the memory barrier in SDDMM can be minimized. The decision also enables more thread-local reduction and minimizes inter-thread communications. For example, one thread loads *4 consecutive vertex-level features* when the feature-length is 32. Thus, only 8 threads participate in inter-thread communication resulting in just *3 rounds* of it when using the tree reduction method. Further, the compiler can now issue up to 4 data-loads per thread before encountering the memory barrier resulting in improved ILP and better data-load performance. We explain how this is achieved without sacrificing memory coalescing (§4.2.1).

The scheduler creates *thread-groups* to utilize all threads of the warp. E.g., it creates 4 thread-group each containing 8 threads when feature-length is 32. Each thread-group handles different cached NZEs in a way that allows more thread-local reduction in SpMM. I.e., each thread-group processes consecutive cached NZEs, so that the reduction along the neighborhood dimension becomes thread-local to a large extent. This scheduling also allows SDDMM to cache the vertex-level features of rows and reuse them till a new row is encountered (§4.2.2), and offers a running reduction in SpMM to be performed to minimize the register usage (§4.3).

# 4 UNIFIED DATA LOAD: TWO-STAGE DESIGN

In the pursuit of optimizing data-load of NZEs and features, GnnOne is dedicated to enhancing data-load balance and promoting data-reuse. Our strategy to achieve this goal employs a two-stage data-load approach, which optimizes how the data is brought from GPU device memory to registers (or GPU shared memory) for computation which has proven to be instrumental in reaching our performance objectives for SDDMM and SpMM both. The challenge here is to design the stage by carefully understanding the factors that impact the performance of data-load stages and for the dynamic feature-length whose value varies based on the configuration of different model layers during runtime.

## 4.1 Stage-1 Data Load and Caching

In this stage, GnnOne divides the data-load of NZEs and corresponding edge-level features equally among warps using the edge-parallel data-load method to avoid any data-load imbalance among warps. To explain, each warp fetches *CACHE_SIZE* (multiple of 32) NZEs and corresponding edge features, if needed, in a coalesced way as shown in Listing 1, stage 1. So, if one row has 1000 non-zero columns and another only 10, this stage allocates 100× more threads to load the non-zero columns of the former than the latter.

```
1  void SDDMM_or_SpMM_Stage1 (edge_t* coo, float* etensor,
       int dim, ...) {
2    int warpId = threadIdx.x/32;
3    int warpLane = threadIdx.x % 32;
4    int absWarpId = blockIdx.x*blockDimx.x+warpId;
5    int eStart = absWarpId*CACHE_SIZE;
6    int eEnd   = eStart+CACHE_SIZE;
7    //Stage1: Load NZE and edge-features and Cache
8    for (int e=eStart+warpLane; e<eEnd; e+=32) {
9      sh_NZE[e-eStart]=coo[e];//Shared memory
10     //Line 11 is needed for SpMM only
11     sh_evalue[e-eStart]=etensor[e];//Shared memory
12   }
13   __syncwarp();
14   ...
15 }
```

**Listing 1: Sparse kernel sketch (simplified) for fully balanced data-load Stage 1**

**Data Reuse.** The loaded data is cached temporarily in the shared memory of the GPU so that it can be reused in stage 2. For *SDDMM*, the row ID and column ID of each NZE are needed in Stage 2 by multiple threads to fetch their vertex-features (vector), and their data-reuse is achieved by caching them in Stage 1. However, the Edge-parallel SDDMM in prior works, such as DGL [35], FeatGraph [18], etc. do not have NZE caching strategy at all, and therefore cannot benefit from the data-reuse of NZEs.

For *SpMM*, NZEs are also reused: the column ID of NZEs is reused by many threads to fetch their vertex-feature Stage 2, while the row ID is needed to reduce the dot products and write the result (vector) to output tensor. Stage 1 additionally caches the corresponding

edge-feature of every NZE which is needed by different threads to perform the dot product with the vertex-features. Therefore, The reuse factor for the cached elements in both SDDMM and SpMM is exactly as the feature dimension of vertex-features.

*4.1.1 CACHE_SIZE Determination and Discussion.* The separate Stage 1 in GnnOne allows us to find the cache size independent of row-length of sparse matrix or feature-length of each vertex, which is not the case in prior sparse kernel works, as we discuss soon.

**CACHE_SIZE and Memory Barrier.** GnnOne determines the cache size based on the hardware characteristic, such as thread count in a warp, shared memory size, etc., and not based on row-length of the dataset. Hence, any multiple of 32 is a good starting point for *CACHE_SIZE* as there are 32 threads in a warp. Kindly note that a memory barrier is needed before Stage 2 can read the cached data from the GPU shared memory. This incurs costs, specifically, memory-related instructions can not be reordered across a memory barrier. Hence, in the case of 32 as *CACHE_SIZE*, each warp thread only issues one data-load, and then waits for the memory barrier. However, by caching more than 32 elements, such as 128, the frequency of this barrier usage is reduced, allowing every warp thread to issue data-load of 4 NZE before waiting for the memory barrier. Hence, performance is better when Stage 1 caches 128 NZEs per warp than just 32, and is confirmed empirically (§5).

**Prior SDDMM Works.** Sputnik [11], FeatGraph [18], dgNN [47], DGL, and others have not cached NZEs in their SDDMM. Hence, they perform more data-load than necessary for loading NZE. Hence, Caching is very novel in SDDMM, and is enabled purely due to the similarity that we observe in this paper about sparse kernels.

**Prior SpMM Works and Row-Length.** Vertex-parallel SpMM approaches uses a fixed cache size of 32 per warp, but also dependent on row-length. E.g., workload-balanced SpMM designs such as GNNAdvisor [37] and Huang et al. [20] break each row into neighbor groups of 32 non-zero columns, and are forced to use cache size to 32. However, the last neighbor group and those rows that have less than 32 non-zero columns cache less than 32 NZE. GE-SpMM [19], caches only 32 NZEs per warp at any time. Even if such works do decide to increase the cache size, it is always limited by the row-length which varies for each row, as many rows that contain fewer non-zero columns are unable to fully utilize the increased cache size. *Hence, unlike GnnOne, prior SpMM works cannot make cache size independent of the row-length and actual computation, while doing no caching for SDDMM.*

**Prior SpMM works and Feature-Length.** Many SpMM works drop caching or have other behaviors when feature-length is less than 32, e.g., GE-SpMM drops caching. On the other hand, GNNAdvisor, and Huang et al keep the warp threads idle while loading the features of NZEs. Since such feature dimensions are common, especially in the last layer of the GNN, a separate Stage 1 fully independent of Stage 2 holds significance and does not distinguish the implementation based on feature-length.

## 4.2 Stage-2 Data Load & Symbiotic Scheduler

In SDDMM, this stage loads the vertex-features of row and column IDs of the cached NZE and performs their dot product. In SpMM, this stage loads vertex-features of the column ID of the cached NZEs, as edge-feature of NZEs are already loaded in Stage 1, and perform their dot product. This stage solves the challenges (§3.2) that reduction stage has on data-load performance using Symbiotic thread scheduler.

**Thread-Group and Need of Scheduling.** The symbiotic thread scheduler uses one thread to load at least 4 features so that the compiler can issue more data-load instructions before a memory barrier is hit due to the need for reduction (inter thread communication) in SDDMM (§3.2). This leads to forming thread groups within a warp to utilize all its threads for various feature-lengths. E.g., if the feature-length is 32, then 8 threads are dedicated to loading them forming one thread group. Hence, there are a total of 4 thread groups within a warp. But in case 16 as feature-length, 4 threads form a thread group, resulting in 8 thread groups within a warp. Also, the count of NZEs processed simultaneously within a warp depends directly on the thread-group count within the warp. I.e., if the thread-group count is 4 (feature-length 32), each thread group handles one NZE, leading to 4 NZEs handled simultaneously by the warp. Further, the cached NZEs are divided equally among thread groups to process. E.g., if the *CACHE_SIZE* is 128 then each thread-group is assigned 32 NZEs 128/4) to process independently.

This leads to *two scheduling decisions to be made*: a) which 4 features are allocated to a thread (§4.2.1); and b) which NZEs are allocated to a thread group (§4.2.2).

*As we explain next, such decisions present challenges on how to keep the coalesced memory access of the warp so that the data-load does not degrade.*

*4.2.1 Vertex-Level Feature Assignment Policy.* Allowing a thread to issue four outstanding data-load instructions for feature loading may break the memory coalescing as the warp (or thread-group) cannot fetch consecutive 32 features (128 bytes) at one time even when the feature-length is 32. It requires careful design to allow memory coalescing. GnnOne turns to a CUDA's native support for vector data-load instruction, such as *float4*. This enables each thread to fetch 4 consecutive vertex features by using a single vector data-load instruction. So, for 32 feature-length, a thread-group consisting of 8 threads achieves full coalesced memory access of 128 bytes. Moreover, it requires just 3 rounds of inter-thread communication ($log_2^8$) compared to the earlier 5 rounds. Results in §5 shows that switching to the proposed thread-grouping leads to a significant speedup of over the vanilla feature-parallel method for feature-length of 32.

The huge speedup in SDDMM is because of performing more loads(each thread loads 4 features now) before the memory barrier, which leads to higher instruction-level parallelism (ILP) for data-load instructions. Hence, if one can come up with a different solution to increase ILP for data-load, it would provide similar optimization in SDDMM. However, not all solutions may perform better if they compromise with memory coalescing. E.g., an extreme design would be to dedicate a single thread to handle all vertex-features of an NZE to perform a full thread-local reduction. This will impact Stage 2 data-load performance warp cannot produce coalesced memory access.

Since, using *float4* easily provided the mechanism, we did not look for any other mechanism. Though prior works have used vector data-load for various other purposes at different stages of sparse kernels, *the unique proposal of GnnOne remains novel in*

*explaining the unknown impact that inter-thread communication creates on the data-load performance through the memory barrier.*

Though SpMM is different than SDDMM, thread-grouping is applicable to SpMM. So, each thread uses *float4* data type during Stage 2 data-load. However, the usage of float4 does not bring significant speedup for SpMM because it does not perform inter-thread communication at the end of each NZE thereby observing no issues due to the memory barrier.

*4.2.2 NZE Assignment Policy.* We explore two thread-group scheduling strategies to decide which cached NZE should be assigned to each thread group within a warp. We explain these two strategies by using 4 thread-group (32 as feature-length), and 128 as CACHE_SIZE. In the **Consecutive** method, the preferred method, the first thread-group is assigned the first 32 cached NZEs(0, 1, 2, ..., 31). The next thread-groups are assigned the next block of 32 NZEs, and so on. Each thread within the thread group handles different features of the same NZE. In the **Round-robin** method, the first thread-group gets NZEs stored in 0, 4, 8, 12, ..., $124^{th}$ location in the cache. The second thread group gets NZEs stored in 1, 5, 9, 13, ..., $125^{th}$ location in the cache, and so on. Listing 2 shows both methods of scheduling for thread groups of a warp.

```
void SDDMM_or_SPMM_Stage2 (edge_t* coo, float* etensor,
     int dim, ...) {
  ...
  //--Start of Stage 2
  //Calculate thread group details
  int thdGrpCount = 4;//Example
  int thdsInThdGrp = 8; //Example
  int thdGrpId = warpLane / thdsInThdGrp;
  int thdGrpLane = warpLane % thdsInThdGrp ;
  //EITHER Consecutive method
  int start = thdGrpId*CACHE_SIZE/thdGrpCount;
  int end = start + CACHE_SIZE/thdGrpCount;
  for (int i = start; i < end; i++){
    e = sh_edge[i]; //fetch the cached NZE
    ...
  }
  //OR Round-robin method
  for(int i=grpId; i<CACHE_SIZE; i+=thdGrpCount){
    e = sh_NZE[i]; //fetch the cached NZE
    ...
  }
}
```

**Listing 2: Sparse kernel sketch (simplified) for data-load Stage 2, showing thread-group scheduling.**

We now investigate how reduction and data reuse influence the choice, and whether one can provide better data-load performance than the other.

**Data-Reuse Analysis: Consecutive method is Better.** *In SDDMM*, the Consecutive method introduces *data-reuse of vertex-features of row ID* for many NZEs until a row split is encountered. This is because the COO is usually arranged in a CSR way [2], and a block of consecutive NZEs is assigned to a thread group. On the other hand, the Round-robin method offers very little data-reuse chance as consecutive NZEs are not allocated to the same thread group. *In SpMM*, no such data reuse exists irrespective of the scheduling method as vertex-features of a row ID are never fetched.

**Reduction Analysis: Consecutive method is Better**. *In SpMM*, the Consecutive method is more useful for reduction as each thread group processes consecutive NZEs, which are likely to have the same row ID. Hence thread-local reduction can be performed along the neighborhood dimension by each thread-group. Kindly, note the Consecutive method needs just one round of dot product exchange among thread-groups for the reduction in case two consecutive thread-groups have some NZEs of the same row ID. However, the Round-robin method requires more inter-thread communications for reductions as consecutive NZEs are assigned in a round-robin fashion to different thread groups. This is likely going to be more costly than a pure thread-local as this method needs to exchange 4096 items (32 features* 4 thread-group * 32 NZEs per thread-group) in the worst case. *SDDMM* reduction has no role in deciding the scheduling method.

It is clear that the Consecutive method is better than the Round-robin method. Moreover, a higher CACHE_SIZE in Stage 1 favors the Consecutive method as each thread group handles more NZE, therefore it can do more thread-local reduction to minimize the thread communication in SpMM. Hence, the Consecutive method in Stage 2 is better suited to complement the Stage 1 design of caching more elements.

## 4.3 Reduction Design

After the data-load phase produces the dot products, the reduction phase reduces those dot products and then writes the reduced results to the output tensor. Thanks to our data-load Stage 2 design which is based on reduction analysis, the final reduction design becomes very simple. All the sparse kernels use the thread-grouping and the Consecutive scheduling method. Due to this, both the kernels rely on the maximum thread-local reduction of dot products as discussed next.

**SDDMM.** A thread performs 4 thread-local reductions, followed by inter-thread communication within the thread group for final reduction. E.g., for 32 as feature-length, reduction only performs 3 rounds of inter-thread communication. The results are written immediately to the output.

**SpMM.** We do a *running reduction* (thread-local), i.e., each thread in a thread group performs reduction immediately after new dot product computation in Stage 2 while handling consecutive NZEs, before a row split is observed by a thread group. This minimizes the register count or shared memory usage that otherwise would have been needed to keep individual dot products. Discovering a row-split is easy in the Consecutive scheduling method because of the usage of COO where every NZE contains its row ID. However, we directly use atomic instruction to write the thread-local reduced value to the output tensor. We are fine with this approach as the results show in §5, and leave the exploration of alternate approaches as a future work.

**Format Selection.** The GnnOne can fit in any format if we can quickly locate the row and column ID from each non-zero element. In this work, we choose the COO format since it not only meets the requirement but also is the standard format, supported by a wide variety of libraries and frameworks. In the following text, we discuss the selection of the sparse storage format and presents various trade-off that guide our choice.

CSR is naturally *not* suited for workload balanced solutions. Hence prior works rely on custom formats. On the other hand,

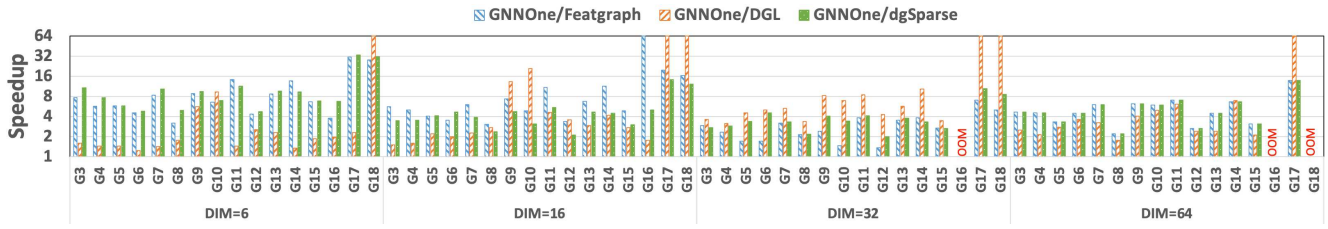


**Fig. 3: SDDMM results: GnnOne speedup over prior works for different feature lengths (dim). A speedup of 64 means that baselines has OOM not GnnOne((log scale, higher is better)**

the COO format has a natural property of workload balancing as done in GnnOne. In addition, the COO format brings a unique advantage in the GNN use cases. GAT [32], GaAN [49], and many other GNNs also invoke SDDMM variants which are naturally suited for edge-parallel computation as the output tensor is at edge-level. Current systems, such as DGL, Pytorch-Geometric, etc. rely on the COO format. Therefore, optimizing SpMM so that it can use the COO format makes these GNN frameworks rely on a single format minimizing memory consumption amid better performance.

Moreover, the vendor libraries such as CuSparse still support the COO format arranged in the CSR way. This indicates that many scientific computation also relies on the COO format, and not just the GNN use-cases. Therefore, having high-performance sparse kernels(SpMM and SDDMM) in the COO format is likely to benefit many scientific workflows.

## 4.4 Discussion

**Is GnnOne SpMM an extension of Nonzero Split SpMV [6, 27, 31]?** We discussed in §3.2 how nonzero-split SpMM by Yang et al. [42] suffers from performance drop. That is because it does not incorporate the understanding of the differences between SpMV and SpMM, and how the reduction phase impacts data-load performance. Hence, GnnOne SpMM is *not* a direct extension of nonzero-split SpMV. We claim that nonzero-split SpMV is a special case of GnnOne SpMM, and not the other way around, based on another fundamental insight that we developed between the classes of two nonzero-split SpMV.

**Trade-off between two classes of SpMV** is between how the NZEs are fetched (coalesced or non-coalesced) and the reductions are performed (limited thread-local or fully inter-thread). *Dalton et al.* [6] (one class of nonzero-split SpMV) fetches NZEs and edge-features in a coalesced manner that forbids any thread-local reduction. Hence, inter-thread reduction is performed by materializing the dot product to the shared memory. On the other hand, *Merrill et al.* [27] (another class) forgoes coalesced fetch of NZE, and edge features (i.e., a thread fetches N consecutive NZEs, thus a warp is not able to coalesce them). However, this decision does enable limited thread-local reduction over N dot products.

We note that GnnOne removes this trade-off in SpMM due to caching NZEs and edge-features, which are reused by many threads to fetch vertex-feature from dense matrix. Hence, in SpMM, we enabled both the coalesced access of NZEs and edge-features plus limited thread-local reduction. *Hence, GnnOne SpMM is not an extension of any of these two SpMV designs. It has different design choices than SpMV.*

In other words, caching in Stage 1 is the key difference in SpMM, which is of no use in spMV as feature-length is just 1. Hence, if GnnOne has to be extended to SpMV, caching in stage 1 is dropped, making our SpMV implementation one of Dalton et al. or Merrill et al. *Thus, the two classes of nonzero-split SpMV are special cases of GnnOne SpMM when the vertex-feature length is 1.*

**Discussion on Feature-Length.** In GNN, the feature-length in sparse kernels is usually a multiple of 4, such as 16, 32, 64, etc. Hence, the usage of float4, and thread-group remains useful. It is only the last layer in some GNN, where the feature-length of sparse kernels is determined by the classification category, such as 6 in Citeseer. In such cases, thread-group remains useful. However, we do replace float4 with other vector data-load instructions, such as float3 when the feature-length is 6, as float4 cannot be used due to memory alignment issues. It still provides speedup in such odd feature-length cases in SDDMM as more than 1 data-load instruction is issued before hitting the memory barrier.

# 5 EVALUATION

The datasets for the experiments are listed in Table 1. We treat them as undirected graphs, as GNN frameworks such as DGL expect this. Hence, the edges are doubled. The edge count in the Table indicates this. The graphs are downloaded from DGL, SNAP [30], and the University of Florida Sparse matrix collection [7]. Kron-21 is a synthetic graph generated using Graph500 generator [13]. The experiment is run using an Nvidia A100 GPU (40GB memory).

This section shows that for various feature lengths, GnnOne[1] outperforms all the prior works significantly on individual kernel measurements as well as on GNN training against a system that has gained additional performance using kernel fusion, while GnnOne relied on individual kernels. It also shows the impact of individual design choices.

## 5.1 SDDMM

Fig. 3 shows speedup of GnnOne over dgSparse [3] (used by dgNN [47]), CuSparse, Sputnik [11], FeatGraph [18], and DGL [35]. The average speedup is 6.02× across all the feature lengths except over Sputnik and Cusparse. The speedup also depends on the feature-length. For feature-length 32, GnnOne achieves an average speedup

[1]Code can be accessed from https://github.com/the-data-lab

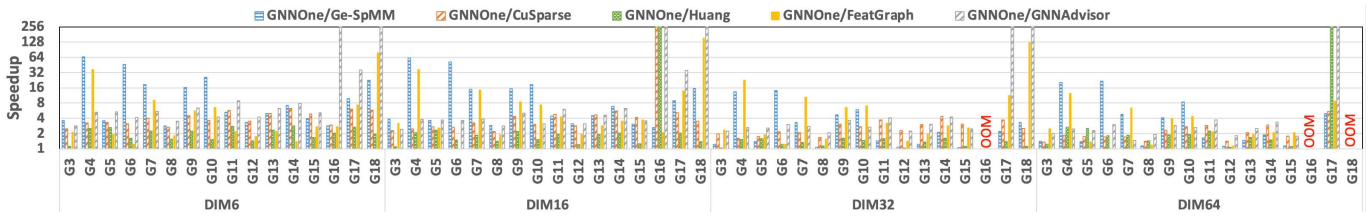


**Fig. 4: SpMM results: GnnOne speedup over prior works for different feature lengths (dim). A speedup of 256 means that baselines have OOM not GnnOne. OOM means every system ran out-of-memory ((log scale, higher is better)**

**Table 1: Graph datasets. * denotes labeled dataset. F = Input feature-length, C = Prediction categories. GNN models deploys a linear layer to project the feature-length to a lower intermediate feature-length (e.g., 16 or 32) before sparse kernels are called.**

| Graph Dataset | Vertex Count | Edge Count | F | C |
|---|---|---|---|---|
| Cora(G0)* | 2,708 | 10,858 | 1,433 | 7 |
| Citeseer(G1)* | 3,327 | 9,104 | 3,703 | 6 |
| PubMed(G2)* | 19,717 | 88,648 | 500 | 3 |
| Amazon(G3) | 400,727 | 6,400,880 | 150 | 6 |
| wiki-Talk(G4) | 2,394,385 | 10,042,820 | 150 | 6 |
| roadNet-CA(G5) | 1,971,279 | 11,066,420 | 150 | 6 |
| Web-BerkStand(G6) | 685,230 | 15,201,173 | 150 | 6 |
| as-Skitter(G7) | 1,696,415 | 22,190,596 | 150 | 6 |
| cit-Patent(G8) | 3,774,768 | 33,037,894 | 150 | 6 |
| sx-stackoverflow(G9) | 2,601,977 | 95,806,532 | 150 | 6 |
| Kron-21(G10) | 2,097,152 | 67,108,864 | 150 | 6 |
| hollywood09(G11) | 1,069,127 | 112,613,308 | 150 | 6 |
| Ogb-product(G12)* | 2,449,029 | 123,718,280 | 100 | 47 |
| LiveJournal(G13) | 4,847,571 | 137,987,546 | 150 | 6 |
| Reddit(G14)* | 232,965 | 229,231,784 | 602 | 41 |
| orkut(G15) | 3,072,627 | 234,370,166 | 150 | 6 |
| kmer_P1a(G16) | 139,353,211 | 297,829,982 | 150 | 6 |
| uk-2002(G17) | 18,520,486 | 596,227,524 | 150 | 6 |
| uk-2005(G18) | 39,459,925 | 1,872,728,564 | 150 | 6 |

of 3.00×, 5.53×, 4.07× over FeatGraph, DGL, and dgSparse respectively. E.g., in the case of Ogb-Product(G12) dataset, GnnOne exhibited a runtime of 11.70 milliseconds (ms), outperforming dgSparse (23.67ms), DGL (50.22ms), and Featgraph (14.76ms) comprehensively. A similar pattern is also noticed for feature-length of 64.

For smaller feature-length (e.g., 16 or 6), a few prior works keep some warp threads idle. *Hence, the speedup achieved by GnnOne is generally higher due to its thread-grouping.* For example, the average speedup is 7.49×,4.70×, 5.04× over FeatGraph, DGL, and dgSparse for feature-length 16. A greater speedup for feature-length 6 also indicates that GnnOne cares for the last GNN layer. The *overall* minimum speedup we observed is against DGL on G6 (1.24×) for feature-length 6, indicating how effective GnnOne techniques are.

In addition, Sputnik and CuSparse encountered errors when $|V|$ exceeds a certain threshold, which seems around 2 Million, and hence is not plotted. Sputnik, which is open-source, allocates $|V|^2$ thread-blocks, and hence this number becomes large enough that is not supported by CUDA. For datasets where $|V|$ is less than this threshold, GnnOne demonstrated significant speedup, E.g., for the Reddit(G14) dataset, GnnOne achieved speedup exceeding 90× and 40× speedups over Sputnik and CuSparse, respectively.

The huge speedup over DGL proves our intuition that edge-centric design alone is not sufficient unless data-reuse techniques are introduced. In comparison to others, a good data-load balanced solution, minimizing the impact of memory barrier on data-load performance, and data-reuse techniques play the main role in the optimization that GnnOne has achieved. We do evaluate these techniques separately in §5.4

## 5.2 SpMM

Fig. 4 shows the speedup of GnnOne over Ge-SpMM [19], CuSparse, Huang et al. [20], FeatGraph [18], and GNNAdvisor [37]. Despite the long line of work in SpMM, including some solutions designed specifically for workload balancing including relying on custom formats, GnnOne out-performed all of them, achieving an average 6.25× speedup across different feature lengths. For feature-length 32, GnnOne outperforms GE-SpMM, CuSparse, GNNAdvisor, and Huang et al by an average of 3.84×, 2.65×, 2.90×, and 1.34× speedup respectively across all datasets. A similar performance trend is observed for 64 feature-length. For feature-length less than 32 (e.g., 16, and 6), GnnOne achieves even better performance as some of the prior works cannot incorporate their proposed methods. E.g., GE-SpMM cannot use caching, while Huang et al and GNNAdvisor keep some warp threads idle when feature-length is less than 32. Hence GnnOne achieves speedup of 13.90× (15.16×), 3.57× (4.20×), 6.25× (7.52×), 1.71× (2.08×) for feature-length 16 (6) respectively over Ge-SpMM, CuSparse, GNNAdvisor, and Huang et al. respectively.

Speedup over FeatGraph is highest (11.30×). This is because its script prints runtime over many block sizes and block count combinations, but crashes after a few combinations. Though we picked the best run-time, it is possible that if the bug is fixed, some other combination might produce a better runtime. However, their authors informed us that it would remain slower than CuSparse.

When feature-length is 32, the minimum speedup for workload-balanced solutions by Huang et al. and GNNAdvisor are negative, i.e., they are slower than Ge-SpMM, a vanilla vertex-parallel SpMM. This indicates the presence of overhead to achieve workload balancing by these works. However, GnnOne still achieves 1.06× minimum speedup compared to Ge-SPMM in this case, and the speedup becomes even more obvious for other feature lengths.

It is clear that a workload-balanced solution of GnnOne always performs better than vanilla vertex-parallel Ge-SpMM, and better than the neighbor grouping method of workload balancing by

Huang et al and GNNAdvisor. These results show the impact of data-load centric design of GnnOne compared to the existing methods, specifically how prior works still suffer from workload imbalance, and how the solution proposed by GnnOne can provide better workload-balancing and data reuse to achieve greater performance. We do evaluate these techniques separately in §5.4

## 5.3 GNN Training

We used three GNN models, namely GCN, GIN, and GAT. We used two-layer GCN, with an intermediate feature size of 16, 5-layer GIN with an intermediate feature size of 64, and 5-layer GAT with an intermediate feature size of 16. The feature-length of the last layer in all these models usually depends on the classification classes. Thus, GNN training procedures contain a sequence of multiple SpMM and SDDMM with different feature dimensions.

**Baseline.** We compare against two baselines DGL and dgNN [47]. DGL uses CuSparse for its SpMM while designing its own SDDMM. dgNN is a highly optimized system that not only optimizes the kernels but also fuses them to achieve even better performance. However, it only supports attention-based GNNs, such as GAT, which we compare against.

Due to various pitfalls present in prior single-gpu GNN systems, as discussed by Gong et al. [12], comparing against them, such as GNNAdvisor [37], Huang et al [20], TLPGNN [10], etc. is not fair. We also do not compare using smaller datasets, as it implies a comparison of framework overhead [12]. PyG [9] ran out of memory for Reddit and OGB-Product datasets when we used its original design of using COO format. This is also reported by prior works [5, 20]. Ge-SpMM and FeatGraph are integrated with DGL, however, we could not run FeatGraph despite trying various GitHub branches of DGL as suggested by their authors. GE-SpMM provides files that needed to be replaced in DGL but due to code reorganization since then, those DGL files have been removed. So, these works are used only for kernel evaluation.

GnnOne is integrated into the GNNBench benchmarking platform [44] to avoid these pitfalls. The platform allows to use non-labeled datasets by using generated labels and features, whose dimensions are listed in Table 1, and cannot be used to measure training accuracy. We rely on them for performance measurements only due to the limited number of labeled datasets.

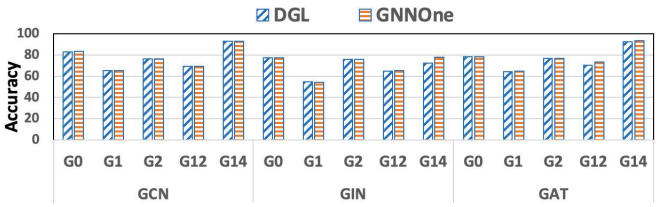


**Fig. 5: GNN training Accuracy results on three GNN models for various datasets, showing that the design proposed by GnnOne can be applied to GNN training correctly.**

*5.3.1 Accuracy Comparison.* Fig. 5 shows that integrating the kernels to a deep learning framework is possible, and works correctly as accuracy is the same as DGL. This test is used to show that the kernel implementation is working.

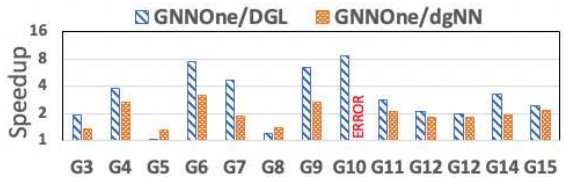


**Fig. 6: End-to-end speedup of GnnOne for GNN training(200 epoch) compared to DGL and dgNN. (Log Scale, higher is better). dgNN produced an error while training G10.**

*5.3.2 Training Time.* **GAT.** Fig. 6 shows the end-to-end GAT training time(200 epochs) compared GnnOne to dgNN and DGL among different large datasets. GnnOne archives 3.68× and 2.01× speedup over DGL and dgNN respectively. Kindly note that dgNN uses the fused kernel calling just one fused kernel instead of individual kernels such as a series of SpMM and SDDMM variants as well as ReLU, dropout layers, etc. that GnnOne invokes individually, yet GnnOne has performance speedup compared to the dgNN, showing the impact of optimizations of individual kernels. We believe kernel fusion would provide even better performance to GnnOne, which we left as future work.

**GCN and GIN.** Fig. 7 shows the speedup achieved by GnnOne over DGL for training GCN and GIN models for 200 epochs, achieving 1.89× and 1.27× average speedup respectively for their training. Further, GnnOne could train GCN on G17 (UK-2002) due to memory saving enabled by GnnOne by keeping a single storage format while DGL ran out-of-memory. For other datasets (G16, and G18), both the systems ran out of memory.

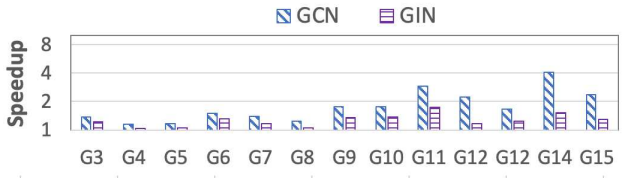


**Fig. 7: Training speedup of GnnOne over DGL: GnnOne could train GCN on G17, while DGL shows out-of-memory conditions (200 epoch). (Log Scale, higher is better).**

It should be noted that GNN models also include many other kernels such as linear function, ReLU, softmax, etc. for which both rely on PyTorch. Hence, it is very clear that optimizing SpMM, and SDDM alone brings significant speedup in GNN training. For example, in Reddit, GnnOne kernel can achieve 6.26× for 16 feature dimensions over CuSparse, which translates to 4.05× speedup over DGL for end-to-end GCN training.

## 5.4 Design Choice Evaluation

We now evaluate the impact of the design decisions that we have proposed in this work.

*5.4.1 Impact of Different Optimizations.* Fig. 8 shows the performance speedup achieved by SDDMM for different optimization techniques presented in the paper. *Baseline* represents data-load balanced solution using COO format but without giving thought to data-reuse and impact of memory barrier. This roughly mimics

the DGL SDDMM design ideas. *+Data-reuse* adds two techniques on top of the baseline solution: caching NZE in stage-1 data-load, and reuse of features of row ID while handling many NZEs. The data-reuse alone results in average 2.78× speedup over baseline.


**Fig. 8: Performance comparison in SDDMM between baseline, baseline + data reuse, baseline + data reuse + float4. The plot uses 32 as the feature length.**

*+Float4* minimizes the impact of the memory barrier on stage-2 data-load: it shows the impact of loading 4 vertex-features per thread instead of 1 to alleviate the impact of the memory barrier on data-load performance which is implicitly applied due to the inter-thread communication needed during the reduction stage in SDDMM. Specifically, for feature-length of 32, it should be noted that both approaches do vertex-feature load using full memory coalescing. However, the approach proposed by GnnOne (4 features per thread) leads to a further 1.80× speedup achieving a total of 4.59× average speedup compared to the baseline performance.

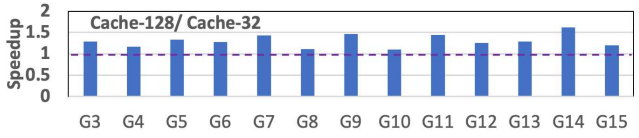


**Fig. 9: Caching 128 NZEs benefits the performance in comparison to caching just 32 NZEs in SpMM. Plot uses 16 as feature-length.**

*5.4.2 Impact of Stage-1 Data-load Cache Size.* Fig. 9 shows that caching 128 NZEs per warp brings additional performance (1.31× speedup) in comparison to caching only 32 NZEs per warp in SpMM. This is because caching more NZEs minimizes the impact of memory barrier on data-load for NZEs and edge-level tensor, and hence improves the SpMM performance.


**Fig. 10: Consecutive method of workload division within a warp achieves better data load performance than the Round-robin.**

*5.4.3 Consecutive vs Round-Robin.* Fig. 10 compares the performance of the two scheduling methods using the COO format for SpMM case for their data-load performance. Kindly recall that the Consecutive method is better for data-reuse in SDDMM and needs to perform less inter-thread communication for reduction in SpMM. However, in this result, we show that the Consecutive method also has a better data-load performance as it achieves slightly above 10% performance than the Round-Robin. We implemented the Consecutive method using Listing 4.2.2, and a similar code was written for Round-Robin, where we did not include the final reduction. Including reduction would have provided even better performance as Consecutive requires much lower inter-thread communication than Round-Robin method.

To explain the performance difference, we note that Consecutive method lets the warp threads work on consecutive NZEs which are likely to be part of same row and hence observes better data-locality than the Round-robin method.

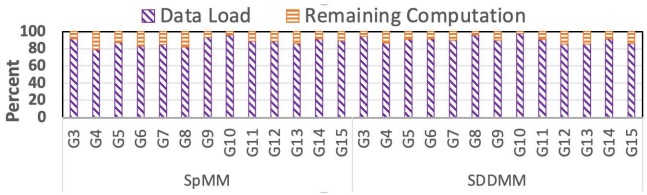


**Fig. 11: Breakdown of the data load performance to show that our initial observation about data load being the costly operation holds.**

*5.4.4 Proving the Observation about Data-Load being Costly Phase.* GnnOne basic premise is the hypothesis that data-load (from GPU memory to registers) is the phase that should take a longer time than the rest of the computation (actual reduction and writing the results to global memory). Fig. 11 shows the breakdown for the dataset. We measure the total time using the end-to-end prototype of GnnOne, and the load time using a partial prototype so that the run-time does not include the impact of the reduction and result write-back. Clearly, the data loading of NZE and features is the main phase that takes more time even after optimization.

*5.4.5 Advantage of COO over Custom Format.* Prior works have proposed various custom storage formats, specifically for SpMM that need a pre-processing step. We leave the pre-processing cost of such custom format out as a one-time cost and focus on understanding their run-time behavior. Firstly, results show that GnnOne is faster than recent SpMM that rely on custom storage for workload balancing [11, 20, 37].

*Secondly*, custom formats developed as part of nonzero-split can work with GnnOne, but it introduces a trade-off compared to COO: 1) Each NZE in the COO format knows its row ID immediately without any additional work. Though, this is achieved by performing 4 extra bytes of load for each NZE when using COO. 2) The custom storage still needs to bring the additional metadata (less than 4 bytes per NZE). However, it uses only a few threads of the warp to do this operation followed by broadcasting the metadata to every thread, and then each thread performs an online search on this metadata to find the row ID of each NZE.

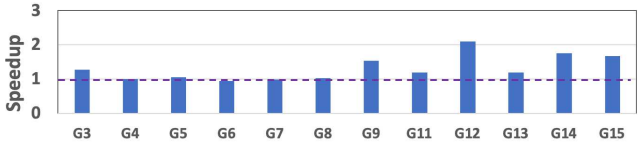


**Fig. 12: GnnOne SpMV speedup compared to Merge-SpMV. GnnOne performs comparably to Merge-SpMV. Merge-SpMV crashed for K21(G10), so is not plotted.**

With the SIMT nature of GPUs, GnnOne uses all the threads to bring the additional row ID along with each NZE using coalesced memory access, while a custom format will keep threads idle while bringing the metadata, followed by additional broadcast that requires thread synchronization or a memory barrier, while the search leads to additional overhead. Hence, we believe COO is fine in place of a custom format, and use SpMV to empirically prove this point next.

Fig. 12 compares COO-based SpMV (GnnOne) against Merge-spMV [27] that uses a custom format. In this case, there is no usage of Stage 1 caching, so we followed the Merge-SpMV idea so that the impact of COO could be evaluated compared to a custom format. GnnOne achieved better or equal performance on all the datasets. For the Reddit and OGB-Product dataset, GnnOne achieves 1.74× and 2.09× speedup. Though the focus of this paper is not SpMV, the results are just an indication that when COO is deployed for nonzero-split, it can perform similarly or better than custom format based nonzero-split SpMV. As discussed earlier, the trade-off between these two solutions is a data load of 4 additional bytes (COO) in GnnOne versus loading less than 4 bytes additional data per NZE but then relying on additional mechanisms such as broadcast and an online search to locate the row ID in the custom format. It seems that this causes more overhead than the additional 4 bytes data-load that GnnOne does on most of the datasets.

## 6 RELATED WORK

Many GNN system optimizations have been proposed recently [10, 17, 21, 22, 25, 34, 38, 41, 43, 46]. Internally, they rely on SDDMM and SpMM sparse kernels, which we focused on in this paper. CuSparse has supported SpMM for a long time on CSR and COO formats. However, it has introduced SDDMM recently supporting CSR format only. Our measurements show that it performs extremely slow, hence, GNN systems, like DGL do not rely on CuSparse SDDMM, but have implemented their own version.

**SDDMM.** DGL uses custom SDDMM based on the COO format for a workload-balanced design, while still relying on CuSparse for SpMM. DGL's SDDMM design has no data reuse: neither caching NZEs nor reusing vertex-features of row ID, proving our earlier point (§1) that workload balancing alone is only an enabling condition for better SDDMM performance. Besides DGL, others [11, 18] adopt vertex-parallel variant, both of which not only lack workload balancing but also uses no caching. Further, Sputnik [11] does not reuse vertex-level features of row ID.

**SpMM.** GE-SpMM [19], FeatGraph [18], TLPGNN [10] follow vertex-centric SpMM. Hence, they still suffer from workload imbalance. Yang et al. [42] that we discussed throughout the paper is an edge-centric SpMM, however, it performs even slower than vanilla vertex-parallel SpMM as reported by it.

GNNAdvisor [37], Huang et al [20], Sputnik [11], ASpt [16] proposed a **custom storage format** using a pre-processing step that produces additional metadata to provide workload balancing in SpMM. GNNAdvisor [37], Huang et al [20] split a row into several groups 32 non-zero columns, generating a metadata that contains explicit row ID and length for every neighbor group as well as for rows whose row-length is less than 32. As row-lengths are hardly multiples of 32, they still suffer from workload imbalance. Sputnik's row swizzling is based on the internal knowledge of the warp scheduler that produces additional array of row ID in decreasing row-length for SpMM. However, it follows a different strategy for SDDMM. ASpt [16] custom format is different for SpMM and SDDMM, and hence it is a system with two custom formats.

Vanilla feature-parallel methods used by prior works [18–20, 37, 42] assign one thread to fetch one feature irrespective of whether the feature-length is a multiple of 32 or not. We have shown how the memory barrier impacts data-load performance and proposed thread-group to truly realize its potential for data-load performance and optimizing the inter-thread communication cost.

**Kernel Fusion in GNN** [5, 39, 47] fuse many GPU kernels using vertex-parallel approach, thereby compromising SDDMM performance for which edge-parallel is better suited. GnnOne have shown that individual edge-parallel SDDMM and SpMM is better than the previous fused vertex-parallel GNN kernels for training.

**SpMV and Graph Processing.** We already discussed non-zero split SpMV in §4.4. A few SpMV/graph processing works have proposed row binning [26, 36] for workload imbalance. Such works generate bins (arrays) such as 4 bins of rows based on their row-length using a pre-processing step. It invokes four kernels, one for each bin where either one thread, one warp, one CTA, or a grid is assigned to each row of the bins respectively. Such works still suffer from the workload imbalance within each bin.

## 7 CONCLUSION

We analyzed the sparse kernels and showed that their fundamental differences do not have any bearing on unifying their design from data-load perspective. We presented many new insights showing how reduction can impact the data-load performance. Finally, a new design is presented to improve the data-load balance and optimization as well as data-reuse in SDDMM and SpMM, and presented many trade-offs that remained unknown in comparison to the well-studied area of SpMV. Our evaluation confirmed that GnnOne achieves better performance. We hope that our study can influence the GNN systems, and vendor libraries to a great extent, as well as sparse models.

## ACKNOWLEDGMENT

The authors would like to thank the anonymous reviewer for their effort and feedback. The work is supported in part by National Science Foundation grant 2245849. The views, opinions, and findings presented in this material are those of the authors, and does not represent the official views or policies of National Science Foundation.

# REFERENCES


[1] CuSparse. https://docs.nvidia.com/cuda/pdf/CUSPARSE_Library.pdf.

[2] cuSPARSE Storage Formats. https://docs.nvidia.com/cuda/cusparse/storage-formats.html?highlight=coo#coordinate-coo.

[3] dgSparse. https://dgsparse.github.io/.

[4] X. Bresson and T. Laurent. Residual gated graph convnets. *arXiv preprint arXiv:1711.07553*, 2017.

[5] Z. Chen, M. Yan, M. Zhu, L. Deng, G. Li, S. Li, and Y. Xie. fuseGNN: Accelerating Graph Convolutional Neural Network Training on GPGPU. In *Proceedings of the 39th International Conference on Computer-Aided Design*, pages 1–9, 2020.

[6] S. Dalton, S. Baxter, D. Merrill, L. Olson, and M. Garland. Optimizing Sparse Matrix Operations on GPUs using Merge Path. In *2015 IEEE International Parallel and Distributed Processing Symposium*, pages 407–416. IEEE, 2015.

[7] T. A. Davis and Y. Hu. The University of Florida Sparse Matrix Collection. *ACM Trans. Math. Softw.*, 38(1), dec 2011.

[8] W. Fan, Y. Ma, Q. Li, Y. He, E. Zhao, J. Tang, and D. Yin. Graph neural networks for social recommendation. In *The World Wide Web Conference*, pages 417–426, 2019.

[9] M. Fey and J. E. Lenssen. Fast graph representation learning with PyTorch Geometric. *arXiv preprint arXiv:1903.02428*, 2019.

[10] Q. Fu, Y. Ji, and H. H. Huang. TLPGNN: A Lightweight Two-Level Parallelism Paradigm for Graph Neural Network Computation on GPU. In *Proceedings of the 31st International Symposium on High-Performance Parallel and Distributed Computing*, pages 122–134, 2022.

[11] T. Gale, M. Zaharia, C. Young, and E. Elsen. Sparse GPU Kernels for Deep Learning. In *2020 SC20: International Conference for High Performance Computing, Networking, Storage and Analysis (SC)*, pages 219–232. IEEE Computer Society, 2020.

[12] Y. Gong, A. Tarafder, S. Afrin, and P. Kumar. Single-GPU GNN Systems: Traps and Pitfalls. *arXiv preprint arXiv:2402.03548*, 2024.

[13] Graph500. http://www.graph500.org/.

[14] W. Hamilton, P. Bajaj, M. Zitnik, D. Jurafsky, and J. Leskovec. Embedding logical queries on knowledge graphs. *Advances in Neural Information Processing Systems*, 31:2026–2037, 2018.

[15] W. Hamilton, Z. Ying, and J. Leskovec. Inductive representation learning on large graphs. In *Advances in neural information processing systems*, pages 1024–1034, 2017.

[16] C. Hong, A. Sukumaran-Rajam, I. Nisa, K. Singh, and P. Sadayappan. Adaptive Sparse Tiling for Sparse Matrix Multiplication. In *Proceedings of the 24th Symposium on Principles and Practice of Parallel Programming*, pages 300–314, 2019.

[17] J. Hu, S. Qian, Q. Fang, Y. Wang, Q. Zhao, H. Zhang, and C. Xu. Efficient graph deep learning in tensorflow with tf_geometric. In *Proceedings of the 29th ACM International Conference on Multimedia*, pages 3775–3778, 2021.

[18] Y. Hu, Z. Ye, M. Wang, J. Yu, D. Zheng, M. Li, Z. Zhang, Z. Zhang, and Y. Wang. FeatGraph: a flexible and efficient backend for graph neural network systems. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, pages 1–13, 2020.

[19] G. Huang, G. Dai, Y. Wang, and H. Yang. Ge-spmm: General-purpose sparse matrix-matrix multiplication on gpus for graph neural networks. In *SC20: International Conference for High Performance Computing, Networking, Storage and Analysis*, pages 1–12. IEEE, 2020.

[20] K. Huang, J. Zhai, Z. Zheng, Y. Yi, and X. Shen. Understanding and bridging the gaps in current GNN performance optimizations. In *Proceedings of the 26th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pages 119–132, 2021.

[21] A. Jangda, S. Polisetty, A. Guha, and M. Serafini. Accelerating Graph Sampling for Graph Machine Learning using GPUs. In *Proceedings of the Sixteenth European Conference on Computer Systems*, 2021.

[22] I. Kim, J. Jeong, Y. Oh, M. K. Yoon, and G. Koo. Analyzing GCN Aggregation on GPU. *IEEE Access*, 10:113046–113060, 2022.

[23] T. N. Kipf and M. Welling. Semi-supervised classification with graph convolutional networks. In *5th International Conference on Learning Representations (ICLR-17)*, 2017.

[24] E. Krahmer, S. v. Erk, and A. Verleg. Graph-based generation of referring expressions. *Computational Linguistics*, 29(1):53–72, 2003.

[25] S. Liang, Y. Wang, C. Liu, L. He, L. Huawei, D. Xu, and X. Li. EnGN: A High-Throughput and Energy-Efficient Accelerator for Large Graph Neural Networks. *IEEE Transaction on Computers*, 70(9):1511–1525, 2020.

[26] H. Liu and H. H. Huang. Enterprise: Breadth-First Graph Traversal on GPUs. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis (SC)*, 2015.

[27] D. Merrill and M. Garland. Merge-based parallel sparse matrix-vector multiplication. In *SC'16: Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, pages 678–689. IEEE, 2016.

[28] R. Perera and P. Nand. Recent advances in natural language generation: A survey and classification of the empirical literature. *Computing and Informatics*, 36(1):1–32, 2017.

[29] M. Schlichtkrull, T. N. Kipf, P. Bloem, R. Van Den Berg, I. Titov, and M. Welling. Modeling relational data with graph convolutional networks. In *European Semantic Web Conference*, pages 593–607. Springer, 2018.

[30] SNAP: Stanford Large Network Dataset Collection. http://snap.stanford.edu/data/.

[31] M. Steinberger, R. Zayer, and H.-P. Seidel. Globally Homogeneous, Locally Adaptive Sparse Matrix-Vector Multiplication on the GPU. In *Proceedings of the International Conference on Supercomputing*, pages 1–11, 2017.

[32] P. Veličković, G. Cucurull, A. Casanova, A. Romero, P. Lio, and Y. Bengio. Graph Attention Networks. *6th International Conference on Learning Representations (ICLR-18)*, 2018.

[33] H. Wang, H. Ren, and J. Leskovec. Entity Context and Relational Paths for Knowledge Graph Completion. *arXiv preprint arXiv:2002.06757*, 2020.

[34] L. Wang, Q. Yin, C. Tian, J. Yang, R. Chen, W. Yu, Z. Yao, and J. Zhou. FlexGraph: A Flexible and Efficient Distributed Framework for GNN Training. In *Proceedings of the Sixteenth European Conference on Computer Systems*, pages 67–82, 2021.

[35] M. Wang, D. Zheng, Z. Ye, Q. Gan, M. Li, X. Song, J. Zhou, C. Ma, L. Yu, Y. Gai, T. Xiao, T. He, G. Karypis, J. Li, and Z. Zhang. Deep Graph Library: A Graph-Centric, Highly-Performant Package for Graph Neural Networks. *arXiv preprint arXiv:1909.01315*, 2019.

[36] 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 SIGPLAN Symposium on Principles and Practice of Parallel Programming*, 2016.

[37] Y. Wang, B. Feng, G. Li, S. Li, L. Deng, Y. Xie, and Y. Ding. {GNNAdvisor}: An Adaptive and Efficient Runtime System for {GNN} Acceleration on {GPUs}. In *15th USENIX Symposium on Operating Systems Design and Implementation (OSDI 21)*, pages 515–531, 2021.

[38] Y. Wang, B. Feng, Z. Wang, G. Huang, and Y. Ding. {TC-GNN}: Bridging Sparse {GNN} Computation and Dense Tensor Cores on {GPUs}. In *2023 USENIX Annual Technical Conference (USENIX ATC 23)*, pages 149–164, 2023.

[39] Y. Wu, K. Ma, Z. Cai, T. Jin, B. Li, C. Zheng, J. Cheng, and F. Yu. Seastar: Vertex-centric Programming for Graph Neural Networks. In *Proceedings of the Sixteenth European Conference on Computer Systems*, pages 359–375, 2021.

[40] K. Xu, W. Hu, J. Leskovec, and S. Jegelka. How powerful are graph neural networks? *7th International Conference on Learning Representations (ICLR-19)*, 2019.

[41] M. Yan, L. Deng, X. Hu, L. Liang, Y. Feng, X. Ye, Z. Zhang, D. Fan, and Y. Xie. HyGCN: A GCN Accelerator with Hybrid Architecture. In *2020 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, pages 15–29. IEEE, 2020.

[42] C. Yang, A. Buluç, and J. D. Owens. Design Principles for Sparse Matrix Multiplication on the GPU. In *European Conference on Parallel Processing*, pages 672–687. Springer, 2018.

[43] Z. Ye, R. Lai, J. Shao, T. Chen, and L. Ceze. SparseTIR: Composable Abstractions for Sparse Compilation in Deep Learning. In *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 3*, pages 660–678, 2023.

[44] P. K. Yidong Gong. GNNBENCH: Fair and Productive Benchmarking for Single-GPU GNN System. *arXiv preprint arXiv:2404.04118*, 2024.

[45] R. Ying, R. He, K. Chen, P. Eksombatchai, W. L. Hamilton, and J. Leskovec. Graph convolutional neural networks for web-scale recommender systems. In *Proceedings of the 24th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining*, pages 974–983, 2018.

[46] B. Zhang, R. Kannan, and V. Prasanna. BoostGCN: A Framework for Optimizing GCN Inference on FPGA. In *2021 IEEE 29th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM)*, pages 29–39. IEEE, 2021.

[47] H. Zhang, Z. Yu, G. Dai, G. Huang, Y. Ding, Y. Xie, and Y. Wang. Understanding GNN Computational Graph: A Coordinated Computation, IO, and Memory Perspective. *Proceedings of Machine Learning and Systems*, 4:467–484, 2022.

[48] J. Zhang, X. Shi, J. Xie, H. Ma, I. King, and D. Yeung. GaAN: Gated Attention Networks for Learning on Large and Spatiotemporal Graphs. In *Proceedings of the Thirty-Fourth Conference on Uncertainty in Artificial Intelligence*, pages 339–349, 2018.

[49] J. Zhang, X. Shi, J. Xie, H. Ma, I. King, and D.-Y. Yeung. Gaan: Gated Attention Networks for Learning on Large and Spatiotemporal Graphs. In *Uncertainty in Artificial Intelligence*, 2018.

[50] Y. Zheng, C. Gao, L. Chen, D. Jin, and Y. Li. DGCN: Diversified Recommendation with Graph Convolutional Networks. In *Proceedings of the Web Conference 2021*, pages 401–412, 2021.

[51] M. Zitnik, M. Agrawal, and J. Leskovec. Modeling polypharmacy side effects with graph convolutional networks. *Bioinformatics*, 34(13):i457–i466, 2018.