FASTEN: Fast GPU-accelerated Segmented Matrix Multiplication for Heterogeneous Graph Neural Networks

Keren Zhou
kzhou6@gmu.edu
George Mason University
Fairfax, VA, USA

Karthik Ganapathi Subramanian
kganapa@ncsu.edu
North Carolina State University
Raleigh, NC, USA

Po-Hsun Lin
plin8@ncsu.edu
North Carolina State University
Raleigh, NC, USA

Matthias Fey
matthias.fey@tu-dortmund.de
Kumo.AI
Mountain View, CA, USA

Binqian Yin
byin2@gmu.edu
George Mason University
Fairfax, VA, USA

Jiajia Li
jiajia.li@ncsu.edu
North Carolina State University
Raleigh, NC, USA

ABSTRACT

This paper introduces FASTEN, a cutting-edge library developed to address the computational challenges inherent in Heterogeneous Graph Neural Networks (HGNNs). The key focus of FASTEN is the optimization of segmented matrix multiplication, a critical operator where existing GNN frameworks and linear algebra libraries often fall short. FASTEN offers an array of solutions to these challenges, including a routing table designed for efficient workload scheduling, adaptive algorithms tailored for handling segments of different shapes and segmented dimensions, and a performance model-guided autotuner to select the best configurations. Furthermore, FASTEN implements interfaces to integrate with widely-used frameworks like PyG, ensuring straightforward adoption in existing HGNN models with minimal adjustments. We have performed comprehensive benchmarks on advanced GPU architectures, including NVIDIA H100, A100, and RTX4090, to demonstrate that FASTEN significantly improves both operator-wise and end-to-end performance across various datasets and HGNNs.

CCS CONCEPTS

• **Computing methodologies** → **Massively parallel algorithms**; *Machine learning*; • **General and reference** → Performance.

KEYWORDS

Graph Neural Networks, GPUs, Matrix Multiplication, Batch Processing, Performance Modeling

ACM Reference Format:
Keren Zhou, Karthik Ganapathi Subramanian, Po-Hsun Lin, Matthias Fey, Binqian Yin, and Jiajia Li. 2024. FASTEN: Fast GPU-accelerated Segmented Matrix Multiplication for Heterogeneous Graph Neural Networks. In *Proceedings of the 38th ACM International Conference on Supercomputing (ICS '24), June 04–07, 2024, Kyoto, Japan.* ACM, New York, NY, USA, 14 pages. https://doi.org/10.1145/3650200.3656593

ICS '24, June 04–07, 2024, Kyoto, Japan

ACM ISBN 979-8-4007-0610-3/24/06
https://doi.org/10.1145/3650200.3656593

1 INTRODUCTION

Graph Neural Networks (GNNs) [53] have gained increasing prevalence in a wide spectrum of applications [13, 17, 29, 52]. GNNs typically represent entities as nodes in a graph, connected by edges that represent a relation. Heterogeneous GNNs (HGNNs) [57] are a natural extension to GNNs, renowned for their ability to model heterogeneous and complex modes of relationships between entities commonly found in real-world data, including social networks [9], biological networks [3], molecular graphs [54], source code [1], and knowledge graphs [33]. For example, in academic networks, relationships such as "Teacher" entities *mentoring* "Student" entities, with both *contributing* to publications at "Conference" entities, are modeled.

The growing use of GNNs has catalyzed the development of open source machine learning frameworks like DGL [47] and PyG [10]. These frameworks are designed to facilitate the development of advanced models and enhance the efficiency of processing large datasets. They often integrate hardware vendor-provided libraries, such as cuBLAS [36], CUTLASS [38], and cuSPARSE [37], to take advantage of the high bandwidth and parallelism of GPUs for acceleration. However, recent studies [16, 23, 55] suggest that the computational power of GPUs is not yet fully exploited in GNNs. This challenge is more pronounced in HGNN computations, where complexity and heterogeneity arise from assigning unique weights to various types of relationships.

Among heterogeneous operations, segmented matrix multiplication (*matmul*) is particularly time-intensive during HGNN training and inference. Segmented matmul involves handling a batch of input matrices of different sizes ($\mathcal{I}_i$), each paired with a distinct weight matrix ($\mathcal{W}_i$) that represents a type in HGNNs, and computing $\mathcal{I}_i \times \mathcal{W}_i$ for each pair. Existing dense linear algebra libraries, such as cuBLAS [36], do not provide specific routines for scenarios involving variable input dimensions. Consequently, deep learning frameworks often launch dense matrix multiplication routines for each segment on the host and execute them on the GPU in sequence. This approach, however, incurs high kernel launch overhead and redundant memory access costs. CUTLASS provides a generalized grouped matmul method [39], but this method is not optimized for segmented scenarios where only one dimension varies and all input matrices are allocated from the same source, causing issues

Table 1: The FLOP Utilization (TF32) of CUTLASS grouped matmul for various datasets on a NVIDIA A100 80GB GPU.

	AIFB	AM	BGS	MUTAG
Feature size = 32	1.45%	2.32%	2.26%	1.95%
Feature size = 64	4.68%	7.49%	7.14%	6.22%

such as extra overhead in indexing matrices and workload imbalance across different Cooperative Thread Arrays (CTAs). As shown in Table 1, the highest FLOPS utilization achieved by CUTLASS's grouped matmul kernel falls below 7.49% in tests conducted on graph datasets.

While a line of research focuses on optimizing HGNN performance, few studies have delved into the sophisticated optimization of computations in GPU kernels. Most of research has instead concentrated on high-level scheduling mechanisms [34], GPU kernel fusion [16, 55], and graph Intermediate Representation (IR) designs [51, 55, 56]. In this paper, we demonstrate that optimal performance of segmented matmul on GPUs requires very careful kernel design, with a focus on fully utilizing GPU resources by exploring factors such as matrix locality, segment shapes, workload scheduling among CTAs, and various GPU architectural features.

To address these issues, we introduce FASTEN, a comprehensive library featuring efficient algorithms specifically designed for segmented matmul in HGNNs. FASTEN achieves high performance across various NVIDIA GPU architectures and makes the following contributions:

- It includes a routing table that efficiently guides the selection of segments to be processed by each CTA on the GPU.
- It features adaptive algorithms, capable of efficiently handling segments of varying shapes and different segmented dimensions.
- It incorporates a performance model guided tuning framework, which fine-tunes a wide range of parameters, from data structures and algorithms to resource usage, at a moderate cost.
- It provides interfaces that adapt data structures and modules for seamless integration with PyG.

To demonstrate the effectiveness of FASTEN[1], we conducted comparative analyses with a vendor-provided library (i.e., CUTLASS [38]), and a compiler-based optimization framework (i.e., Graphiler [55]). We evaluated widely used HGNNs, including HGT [20], RGCN [41], and RGAT [7]. Our experiments, performed on NVIDIA A100, RTX4090, and GH200 GPUs, have shown up to a 117.54× speedup, with an average speedup of 13.65× and 4.72× in operator-wise benchmarks compared to CUTLASS and cuBLAS, respectively. Furthermore, HGNNs utilizing FASTEN demonstrated an average of 1.86× and 4.02× end-to-end speedups relative to vanilla PyG models and those compiled with Graphiler, respectively.

2 BACKGROUND

In our model, we consider a graph $G = (V, E, \mathcal{T})$, with V as the set of nodes, E as the edges connecting them, and $\mathcal{T}$ as the types associated with nodes and edges. Each node is associated with a unique feature vector $I_{v_i} \in \mathbb{R}^K$, residing in a K-dimensional space.

GNNs adopt the *Message Passing* [14] mechanism to propagate information from a source node to its neighbors. This mechanism divides the computation into three individual phases: *message*, *aggregate*, and *update*. Consider a GNN model with N modules linked in sequence; the output of the n-th module can be represented as

$$I_{v_i}^{n+1} = update\left(aggregate_{v_j \in \text{neighbors}(v_i)}\left(message\left(I_{v_j}^n, I_{v_i}^n\right)\right)\right) \quad (1)$$

The *message* phase applies custom functions to the features of v_i and v_j if there is an edge connecting them. Then, we *aggregate* features from the neighbors of v_i together with a reduction function (e.g., *sum*). Finally, the *update* function updates the feature vector of v_i and uses it as input for the next module.

HGNNs define custom *message* functions that apply computations to nodes based on their relationships with neighbors. For example, in RGCN [7], a linear transformation is applied to each neighbor's feature vector $I_{v_j} \in \mathbb{R}^K$ and the weight of the relationship $W_{\tau_{v_i,v_j}} \in \mathbb{R}^{K \times Q}$, where τ_{v_i,v_j} denotes the type of relation between v_i and v_j, and Q denotes the number of features of each node used in the subsequent GNN module. The transformation can be represented as shown in Equation 2.

$$message(I_{v_j}^n, I_{v_i}^n) = I_{v_j}^n \mathbf{W}_{\tau_{v_i,v_j}}^n \quad (2)$$

If we process all nodes in a graph in a batch, we can compute messages for all relations $\mathcal{T}$ as shown in Equation 3, where $I_\tau^n \in \mathbb{R}^{|\tau| \times K}$ and $\mathbf{W}_\tau^n \in \mathbb{R}^{K \times Q}$. Here, $message(E)$ represents the batched computation of messages for all edges in E, with each edge belonging to a type $\tau \in \mathcal{T}$.

$$message(E) = I_\tau^{n-1} \mathbf{W}_\tau^n, \text{ for } \tau \text{ in } \mathcal{T}. \quad (3)$$

Equation 3 illustrates a **segmented matmul** operation, **where a single dimension (i.e., $\mathcal{T}$) in the input and weight matrices is segmented to form segment pairs, with each pair performing a dense matmul**. In addition to the forward phase, it is worth noting that the backward phase of the message function also performs segmented matmul. Let us denote $d\mathbf{I}_\tau^{n+1} \in \mathbb{R}^{|\tau| \times Q}$ as the gradient of the output of the n-th module. The gradients of the input features and weights can be represented by Equations 4 and 5, respectively.

$$dI_\tau^n = d\mathbf{I}_\tau^{n+1} (W_\tau^n)^\top, \text{ for } \tau \text{ in } \mathcal{T}. \quad (4)$$

$$dW_\tau^n = (I_\tau^n)^\top d\mathbf{I}_\tau^{n+1}, \text{ for } \tau \text{ in } \mathcal{T}. \quad (5)$$

Similar to RGCN, other HGNNs, such as HGT [20] and RGAT [7], also incorporate segmented matmul computations. HGT and RGAT both apply segmented matmul for the "query" and "key" matrices. Additionally, HGT performs segmented matmul to compute the mutual attention between the features of the source and target nodes.

Fig. 1 illustrates the ratio of time consumed by segmented matmul computations during training across various HGNNs on different NVIDIA GPUs. It shows that segmented matmul is one of the most time-consuming processes that needs optimization.

[1] Our code is available at https://github.com/Deep-Learning-Profiling-Tools/fasten

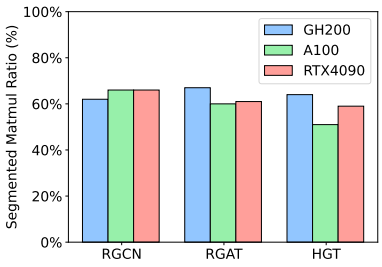


Figure 1: Ratio of time spent on segmented matmul-related computations in training various HGNNs. RGCN and RGAT were trained using the AM dataset [41], while HGT was trained using the Freebase dataset [6].

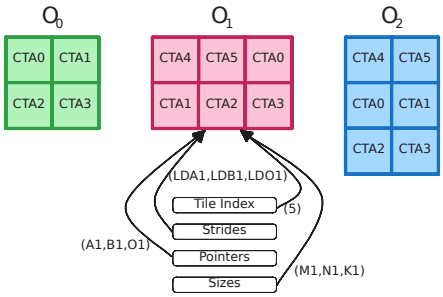


Figure 2: Grouped matmul's scheduling when handling three output matrices. CTA2 is responsible for a tile of $O_1 = A_1 B_1$, where $A_1 \in \mathbb{R}^{M_1 \times K_1}$ and $B_1 \in \mathbb{R}^{K_1 \times N_1}$. In order to load the matrix data, CTA2 must read its relative tile index within this group, as well as the strides, pointers, and sizes of the matrices being processed.

3 EXISTING WORK

We describe several common methods for computing Equation 3 and discuss their shortcomings, which motivates the design of algorithms in FASTEN.

Loop over matmuls. [44] This approach involves sequentially launching dense GPU matmul kernels from the host CPU for each relation type (e.g., $\mathbf{I}_\tau^n \mathbf{W}_\tau^n$), utilizing vendor-optimized libraries like cuBLAS for each kernel. The method becomes increasingly costly with a growing number of relations in $\mathcal{T}$, due to the kernel launching overhead. Moreover, on advanced GPU architectures, such as the NVIDIA H100 with more than a hundred streaming processors, significant load imbalance issues may arise, particularly for relations with a small number of nodes or edges.

Batched matmul. [43] An alternative to the loop over matmuls approach is to use a single batched matmul kernel, which requires that all input matrices have the same shape. One common strategy to adapt batched matmul for segmentation involves broadcasting each weight matrix according to the number of instances in the input. This changes the number of weight matrices from $|\mathcal{T}|$ to $\sum |\tau|$ for all $\tau \in \mathcal{T}$, ensuring that each input instance is associated with a replicated weight matrix from the original. *Although this approach could eliminate the kernel launch overhead using a single kernel, it significantly increases the memory footprint and the number of CTAs, leading to redundant memory access and CTA scheduling overhead.*

Grouped matmul. [39] CUTLASS's **grouped matmul** kernel facilitates batched matrix multiplications by **taking arrays of input and weight matrices, each potentially from different sources and with varied dimensions, and performing a dense matmul operation between each paired matrix independently**. As shown in Fig. 2, this approach employs a scheduling strategy that allocates the computation of one or more tiles, each of equal size, of the resulting matrix to each CTA. Adapting grouped matmul for segmented matmul in HGNNs, however, encounters several major issues, leading to inefficiencies in memory access, scheduling, and workload balance. 1) *Indirect memory access.* In grouped matmul, input matrices of varying sizes may originate from different allocations, thus it requires accessing arrays of pointers to input, weight, and output matrices, as well as the dimensions and strides of them, before loading matrix data. In contrast, segmented matmul involves input matrices sourced from a single allocation, where each matrix differs from the others in at most one dimension. 2) *Scheduling.* The round-robin scheduling mechanism in the grouped matmul of CTAs does not take data locality into account, particularly when the feature size is small, as further elaborated in Section 5.2. 3) *Workload imbalance.* The grouped matmul implementation partitions workload only based on the output tile sizes, but overlooks the tile sizes of input and weight matrices assigned to each CTA. As a result, significant workload imbalances may exist when the accumulation dimension is different among input matrices, as elaborated in Section 5.3.

These insights motivate our approach to segmented matmul, which outperforms the aforementioned methods with highly optimized strategies that minimize memory access overhead and enhance workload balance.

4 OUR APPROACH

Segmented matmul differs from grouped matmul in its approach to index matrices. Unlike grouped matmul, which requires a separate pointer for each matrix, segmented matmul uses only a single pointer for each type of matrix—input, weight, and output. This is because each matrix type is allocated in contiguous memory space from the same original source, allowing for simplified pointer management and accesses. Additionally, in segmented matmul, there is no need to create size arrays for each matrix, as matrices belonging to the same type have at most one dimension that varies in size; that dimension with variable sizes is the one to be divided. For example, in Equation 3, the dimensions K and Q are fixed, and the workload can be divided along with the $\mathcal{T}$ dimension.

The differences between segmented matmul and grouped matmul motivate the design of a **Routing Table** to assign workloads among CTAs. Fig. 3 demonstrates how FASTEN's basic segmented matmul algorithm operates, with K serving as the accumulation dimension. We divide the $\mathcal{T}$ dimension using a TILE_$\mathcal{T}$ parameter, ensuring that each CTA is responsible for computing a [TILE_$\mathcal{T}$, TILE_Q] tile in the output matrix. The results of this division are stored in the routing table. Each row in the table contains three entries: the first entry indicates the *type* of the corresponding weight matrix (i.e., $\mathcal{W}_\tau$), while the second and third entries specify the *start*

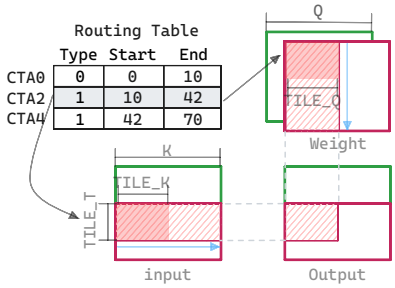


Figure 3: Segmented matrix multiplication algorithm and a routing table, where K is the accumulation dimension, $\sum|\tau| = 70$, and TILE_$\mathcal{T} = 32$. Distinct types are visually differentiated by colors: green represents type 0, and red represents type 1.

and *end* offsets for the $\mathcal{T}$ dimension, respectively. Initially, each CTA retrieves its starting and ending offsets for the $\mathcal{T}$ dimension by accessing the routing table using its CTA ID. These offsets enable the CTA to calculate the addresses of the input and output matrices. Similarly, the CTA calculates the addresses of the weight matrix using the type obtained from the same row in the routing table. Subsequently, the CTA loops through tiles of the input and weight matrices along the accumulation dimension. In each iteration, it performs a dot operation and accumulates the results in an output tile that is eventually stored back in global memory.

Algorithm 1 further presents the major steps of segmented matmul with *Multi-stage Buffering* and *Tensor Core* support.

Algorithm 1 Basic Segmented Matmul Algorithm.

```
Input: ctaID, routingTable, inputPtr, weightPtr, outputPtr
1  row ← GetRow(ctaID)
2  type, start, end ← row[0], row[1], row[2]
3  inputAddrs ← GetAddrs(inputPtr, routingTable, start, end)
4  weightAddrs ← GetAddrs(weightPtr, routingTable, type, type + 1)
5  outputAddrs ← GetAddrs(outputPtr, routingTable, start, end)
6  outputTile ← Zeros(end - start, TILE_Q)
7  inputTiles ← Allocate(N, TILE_T, TILE_K)
8  weightTiles ← Allocate(N, TILE_K, TILE_Q)
9  for i = 0 to N_stages − 2 do                          ▷ Prefetch tiles
10     inputTile[i] ← AsyncLoad(inputAddrs, i)
11     weightTile[i] ← AsyncLoad(weightAddrs, i)
12 for i = 0 to K / TILE_K - 1 do                        ▷ Loop over tiles
13     tileIdx ← i % (K / TILE_K)
14     AsyncWait(tileIdx)
15     A ← Convert(inputTile[tileIdx])                   ▷ Shared to registers
16     B ← Convert(weightTile[tileIdx])                  ▷ Shared to registers
17     outputTile ← outputTile + TensorDot(A, B)
18     nextIdx ← i + N_stages - 1
19     nextTileIdx ← nextIdx % (K / TILE_K)
20     inputTile[nextTileIdx] ← AsyncLoad(inputAddrs, nextIdx)
21     weightTile[nextTileIdx] ← AsyncLoad(weightAddrs, nextIdx)
22 C ← Convert(outputTile)                               ▷ Registers to registers
23 Store(outputAddrs, C)
```

Multi-stage Buffering. Using single tiles in shared memory to store input or weight data may not always be efficient, as it may not fully overlap the computation with the memory accesses [21]. To overcome this limitation, we have implemented a multi-stage buffering technique that allows for fetching multiple tiles while performing computation. As indicated in Line 7 and Line 8, we allocate *stages* buffers to create a pipeline that interlaces compute and memory accesses, where *stages* will be adjusted according to resource limitation.

Moreover, to optimize register usage, we employ asynchronous memory load instructions (`cp.async` [35]) on NVIDIA GPUs, which directly transfer data from global memory to shared memory, bypassing both the L1 cache and registers. Additionally, we use a 2D address swizzling technique [4] to prevent bank conflicts in shared memory. Before the loop begins, we start $N_{\text{stages}} - 1$ asynchronous copy transactions. Then, before the computation in each iteration, we ensure that only necessary asynchronous transactions are completed, as shown in line 14. We also carefully mask out-of-bound memory accesses within the loop to prevent any unintended side effects.

Tensor Core. FASTEN leverages tensor core instructions to achieve high compute throughput for TF32 and FP16 precisions. Unlike conventional CUDA cores, utilizing tensor cores requires specific data layouts [35]. This requirement means that each thread must hold values at specific coordinates to produce corresponding results. Therefore, the input operands are converted before applying each tensor core dot operation, as shown in Line 15 and Line 16. Since the output of `mma` and `wgmma` instructions also has a specific layout, a conversion of the output tile (Line 23) is necessary before storing it in global memory.

5 OPTIMIZATIONS

Though the basic segmented matmul algorithm (Algorithm 1) offers a simple version that simplifies memory access and gains higher performance (see Fig. 13) compared to grouped matmul, it lacks efficiency in several scenarios. To address this issue, this section presents several adaptive algorithms tailored to various shapes and accumulation dimensions, including dynamic tiling for small matrices, register blocking for the small accumulation dimension, and 3D parallelization for the dynamic accumulation dimension.

5.1 Dynamic Tiling

In real datasets, we observed that each relation type might have a varying number of corresponding edges and show a long-tail phenomenon [5] that a large portion of the types only have a small number of edges. Fig. 4 illustrates the distribution of the number of edges for each type of the AM dataset (details described in Table 3). It is evident that many relations have fewer than 512 edges. In such a scenario, if a large tile size (e.g., 512) is employed, many CTAs end up processing fewer than 512 edges. This leads to a waste of computational resources, as the CTAs are forced to perform calculations on zeros that have been padded to these oversized tiles. This inefficiency highlights the need for a more dynamic approach [28] to tile size allocation.

To address this problem, we introduce the concepts of *virtual tile* and *physical tile*. The *physical tile* refers to the maximum amount of shared memory used by the kernel, which must be a static number determined before a kernel launch. The *virtual tile*, on the other hand, refers to the actual tile size used by a specific block. Rather than statically choose the tile sizes (Lines 7 and 8 in Algorithm 1), we dynamically decide them for each CTA . Based on the number of edges (i.e., *end* − *start*), we allocate only a portion of the original shared memory. Besides, since tensor core instructions

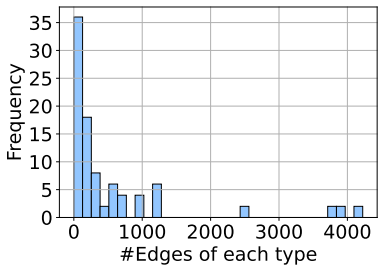
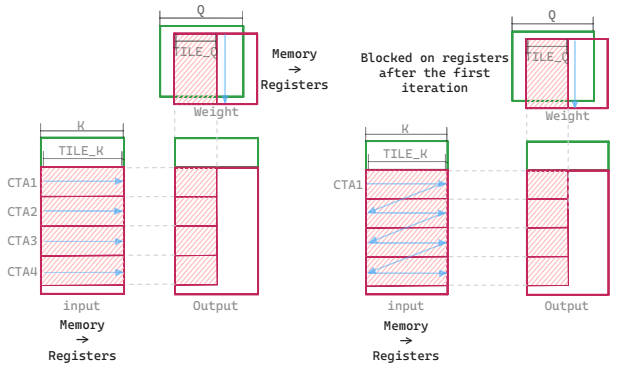


Figure 4: The distribution of edge counts per relation type in the AM dataset, with each histogram bin representing an edge count interval of 128.



(a) Without register blocking. **(b) With register blocking.**

Figure 5: An example of register blocking where CTA1 will process four tiles instead of one and block the weight tile on registers to reduce memory transactions.

(e.g., `mma.m16n16k8`, `mma.m16n8k8`) require specific input operand shapes, we round up the actual tile size to the nearest multiple of 16 in our code and pad any out-of-bound elements with zeros.

However, dynamic tiling can reduce the efficiency of the instruction cache and introduce branch instructions. To mitigate this overhead, we calculate a *Utilization Factor* as shown in Equation 6, which represents the ratio between the actual amount of computation and the theoretical amount of computation. Utilization factor is calculated and associated with a routing table prior to segmented matmul. Dynamic tiling is enabled only when the utilization factor falls below a predefined threshold (e.g., 0.5).

$$\text{Utilization Factor} = \frac{\sum_{\tau\in\mathcal{T}} |\tau|}{\sum_{\tau\in\mathcal{T}} \lceil \frac{|\tau|}{\text{TILE\_}\mathcal{T}} \rceil \times \text{TILE\_}\mathcal{T}} \tag{6}$$

5.2 Register Blocking

As previously mentioned, the dimensionality of features in HGNNs typically falls within a modest range, such as 16 to 128. When K is the accumulation dimension and $Tile_K = K$, each CTA processes only a single tile of the input and weight matrices. Furthermore, subsequent tiles of the input matrix, when belonging to the same type, may share the same weight tile. This access pattern, as shown in Fig. 5a, can lead to inefficiencies for two reasons. First, redundant

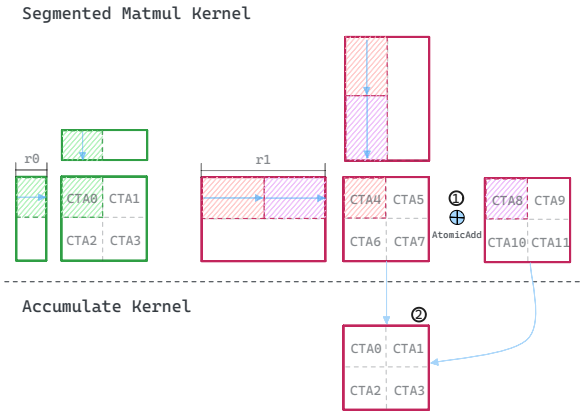


Figure 6: Two 3D parallelization strategies are employed to accelerate segmented matmul when the accumulation dimension varies among input and weight matrices. Option ① is utilized when determinism is not required. On the other hand, option ② is employed when determinism is necessary, involving the use of intermediate memory.

memory instructions may be issued by multiple CTAs to access the same weight tile. Second, each CTA processes only a small tile, which can increase the overhead in CTA scheduling, particularly when processing a large number of edges.

To mitigate these issues, we adopt a *persistent processing method* [18], which tasks a single CTA with the sequential handling of multiple tiles, as shown in Fig. 5. This method incorporates a loop that allows a CTA to iterate over several input tiles assigned to it. In each iteration, only the input tiles are loaded using the multistage buffer approach described in Algorithm 1. In contrast, the weight tile is loaded into registers just once during the initial iteration and is subsequently reused for all dot operations. The results are stored in global memory at the end of each iteration. With this strategy, we not only reduce the number of memory transactions, but also decrease the CTA scheduling overhead.

5.3 3D Parallelization

In Equations 3 and 4, we use static accumulation dimensions K and Q correspondingly, which are known when an HGNN model is initialized. In comparison, Equation 5 uses a dynamic accumulation dimension, τ, to calculate the gradients of the weight matrix. This variability poses a challenge, as the standard grouped matmul scheduling, depicted in Fig. 2, does not account for the fluctuating computational demands associated with different $|\tau|$ sizes. However, as illustrated in Fig. 4, the disparity in the number of edges of each edge type in real datasets can lead to significant computational inefficiencies.

To overcome this challenge, we devised a novel 3D parallelization algorithm that extends parallelization to include the accumulation dimension in Equation 5. Unlike the standard split-k matmul [36], which only applies to dense matmul, our approach utilizes the routing table to record the corresponding start and end offsets for each CTA, thus balancing workloads among a batch of matmuls of

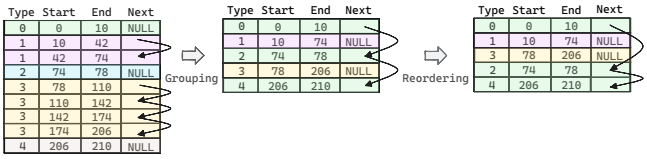


Figure 7: An example of using tile grouping and reordering to reduce the memory consumption and allow for simple indexing.

different sizes along the accumulation dimension. FASTEN implements two versions of the 3D parallelization algorithm: the *non-deterministic* and the *deterministic* versions. The *non-deterministic* version launches a single segmented matmul kernel. If multiple CTAs split a matrix on the accumulation dimension, they collaborate on a single final tile. In this case, we use `atomicAdd` instructions to resolve conflicts when storing results in global memory. For example, Fig. 6 shows that CTA4 and CTA8 collaborate on a single final tile. Otherwise, if a single CTA is responsible for the final tile, such as CTA0 in Fig. 6, we use `store` instead of `atomicAdd` for better throughput.

When determinism is paramount, particularly in model training, FASTEN implements a *deterministic* version that adopts a *dual-kernel* strategy. The first kernel mirrors the computation of the non-deterministic version but substitutes atomic operations with direct store instructions. If a single CTA suffices for an output tile, the result is written directly to the final buffer; otherwise, intermediate output tiles (e.g., CTA4 and CTA8 in Fig. 6) go to an intermediate buffer. The second kernel launches with fewer CTAs to accumulate these intermediate results in the final buffer, with one CTA per final output tile to ensure determinism.

Our 3D parallelization algorithm adeptly handles workload variations among CTAs due to the dynamic accumulation dimension. In practice, the deterministic version exhibits performance similar to that of its non-deterministic counterpart, making it a practical solution for ensuring computational precision when there is sufficient GPU memory for the intermediate buffer.

5.4 Tile Grouping and Reordering

Both the register blocking and the 3D parallelization variants require that each CTA persistently processes multiple tiles. We define a **block** as the collection of all tiles processed by a single CTA, with the maximum being N_{blocks} tiles per block. This section outlines our design that supports this mechanism and improves workload balance.

A straightforward method to allow each CTA to process multiple blocks is to replicate the routing table N_{blocks} times, allowing us to locate the subsequent tile for each CTA. This technique is similar to the scheduling policy used in CUTLASS. However, replicating the routing table not only increases memory usage, but also adds overhead from accessing additional rows in the routing table. Instead, we developed the *grouping* method to address the problem. As demonstrated in Fig. 7, we initially group neighboring tiles of size TILE_$\mathcal{T}$ into a single *large* tile, adjusting the start and end addresses accordingly, with a maximum grouping of N_{blocks} tiles. It is important to recognize that *small* tiles, defined as those with a size less than TILE_$\mathcal{T}$, are not combined into *large* tiles. If we assign individual CTAs to handle these small tiles, imbalance issues can arise, particularly when the utilization factor (Equation 6) is low. Therefore, we extend the routing table with a *next* field. A *NULL* value in the next field indicates a large tile. On the other hand, we link small tiles, whose cumulative edge count is less than TILE_$\mathcal{T} \times N_{\text{blocks}}$, into a single small block by setting the proper indexing in the *next* field.

To streamline the process of identifying the starting tile of a block for each CTA, we utilize a straightforward indexing function, such as dividing the CTA ID by the number of tiles per row. However, this indexing mechanism is incompatible with a routing table after grouping, which requires that CTAs handling small blocks be directed to the start tile in the next field chain. To resolve this issue, a *reordering* approach for tiles within the routing table is essential. We position the larger tiles and the initial small tiles of each block at the forefront of the routing table. This ensures initial access to these tiles. Subsequently, we update the next fields in tiles of the same block to reflect this new arrangement. In addition, small tiles within the same group are placed close to each other to enhance memory efficiency.

In summary, the combination of tile grouping and reordering achieves a balanced assignment of workload through the routing table and makes it compatible with efficient indexing functions.

6 PERFORMANCE MODEL GUIDED TUNING

In this section, we a performance-model guided approach to optimize segmented matmul performance across various architectures and matrix shapes. Our proposed models are coarse-grained, primarily aimed at correlating time with tuning knobs such as tile sizes, rather than precisely estimating running time.

6.1 Performance Modeling

Following previous studies [21, 31] on modeling dense matmul performance, we model segmented matmul's parallelism and decompose the time spent in each CTA into different phases. Initially, we use Equation 7 to estimate the total time:

$$Time_{\text{total}} = N_{\text{waves}} \times average(Time_{\text{wave}}) \tag{7}$$

In this model, a "wave" is defined as a group of CTAs running concurrently on the GPU. We introduce both the inter-wave and intra-wave performance models.

Inter-wave Modeling. The total number of waves, N_{waves}, is calculated as follows:

$$N_{\text{waves}} = N_{\text{partial\_waves}} + N_{\text{full\_waves}} \tag{8}$$

Here, N_{waves} is the sum of *partial* and *full* waves. *Partial* waves are characterized by underutilized Streaming Multiprocessors (SMs), unlike *full* waves, where SMs are fully utilized. Typically, in segmented matmul scenarios, only the final wave is a partial wave.

We define the ideal number of waves as:

$$\begin{aligned} N_{\text{waves}}^{\text{ideal}} &= \frac{N_{\text{CTAs}}}{Occupancy_{\text{CTAs}}} \\ &= \frac{\sum_{\tau \in \mathcal{T}} \lceil \frac{|\tau|}{\text{TILE\_}\mathcal{T} \times N_{\text{blocks}}} \rceil \times \lceil \frac{Q}{\text{TILE\_}Q} \rceil \times \lceil \frac{K}{\text{TILE\_}K} \rceil}{Occupancy_{\text{CTAs}}} \end{aligned} \tag{9}$$

Here $Occupancy_{\text{CTAs}}$ refers to the maximum number of CTAs that run concurrently on an SM. Consequently, we can define *Parallel Efficiency* as the ratio between the number of ideal waves and the number of actual waves:

$$Eff_{\text{parallel}} = \frac{N_{\text{waves}}^{\text{ideal}}}{N_{\text{waves}}} \tag{10}$$

Intra-wave Modeling. To estimate the time within each wave, we differentiate between large and small block processing times. CTAs dealing with large blocks access the routing table once, whereas CTAs dealing with small blocks do so N_{blocks} times. The ratio of large blocks, $\mathcal{R}$, can be determined based on τ, N_{blocks}, and $TILE\_\mathcal{T}$. The wave time is then calculated as:

$$Time_{\text{wave}} = \mathcal{R} \times Time_{\text{wave}}^{\text{large}} + (1 - \mathcal{R}) \times Time_{\text{wave}}^{\text{small}} \tag{11}$$

The respective times for large and small blocks are:

$$Time_{\text{wave}}^{\text{large}} = Time_{\text{indexing}} + N_{\text{block}} \times Time_{\text{loop}} \tag{12}$$

$$Time_{\text{wave}}^{\text{small}} = N_{\text{block}} \times (Time_{\text{indexing}} + Time_{\text{loop}}) \tag{13}$$

$Time_{\text{loop}}$ is the duration spent in the compute loop, as demonstrated in Algorithm 1 at Line 12. The detailed $Time_{\text{indexing}}$ is given by:

$$\begin{aligned} Time_{\text{indexing}} =& N_{\text{block}} \times \left(Hit_{\text{L2}}^{\text{routing\_table}} \times Latency_{\text{L2}}\right. \\ &\left. + (1 - Hit_{\text{L2}}^{\text{routing\_table}}) \times Latency_{\text{DRAM}}\right) \end{aligned} \tag{14}$$

We assume an increase in $Hit_{\text{L2}}^{\text{routing\_table}}$ with a reduction in the size of the routing table. Furthermore, $Time_{\text{loop}}$ is decomposed into various components:

$$Time_{\text{loop}} = Time_{\text{comp}} + Time_{\text{wait}} + Time_{\text{sync}} + Time_{\text{store}} \tag{15}$$

$Time_{\text{comp}}$ relates to the duration of dot operations:

$$Time_{\text{comp}} = \frac{FLOPS_{\text{tile}} \times N_{\text{iters}} \times Occupancy_{\text{CTAs}}}{FLOPS_{\text{SM}}} \tag{16}$$

$Time_{\text{wait}}$ represents the waiting period required for the dot operands to be ready, while $Time_{\text{sync}}$ represents the time spending on synchronizing threads to avoid shared memory data race.

This waiting period is reduced as N_{stages} or $Occupancy_{\text{CTA}}$ increase, helping to hide latency.

$$Time_{\text{wait}} = \max(0, Time_{\text{load}} - Time_{\text{comp}}) \tag{17}$$

Suppose most of the L1 cache is reserved for shared memory, we load data either from the L2 cache or DRAM at each iteration with a bandwidth of BW_{L2} and BW_{DRAM}, respectively. Then we can define $Time_{\text{load}}$ as the time spent loading tiles for N_{iters} iterations.

$$Time_{\text{load}} = \frac{Size_{\text{tile}} \times N_{\text{iters}} \times Occupancy_{\text{CTAs}}}{Hit_{\text{L2}}^{\text{tile}} \times BW_{\text{L2}} + (1 - Hit_{\text{L2}}^{\text{tile}}) \times BW_{\text{DRAM}}} \tag{18}$$

$Time_{\text{store}}$ is triggered once in the 3D parallelization and basic algorithms, but multiple times in the register blocking algorithm. $Time_{\text{sync}}$ is the time spent synchronizing all threads within each CTA, which varies based on the number of warps per CTA. Based on the decomposition of $Time_{loop}$, we can define Eff_{comp} as the compute time relative to the load time and other durations:

$$Eff_{\text{comp}} = \frac{Time_{\text{comp}}}{Time_{\text{load}} + Time_{\text{sync}} + Time_{\text{indexing}} + Time_{\text{store}}} \tag{19}$$

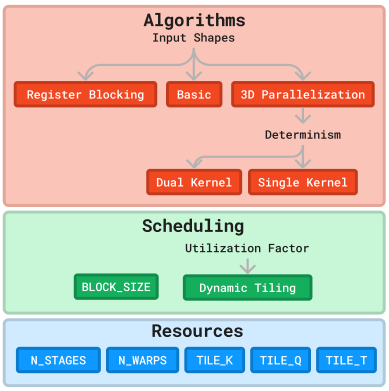


Figure 8: The multi-layered autotuning approach to kernel optimization. Each layer's decision influences the subsequent layer to ensure optimal performance based on underlying hardware resources.

Guided by this performance model, we aim to enhance segmented matmul's performance by identifying and adjusting knobs to achieve high compute and parallel efficiency.

6.2 Autotuning

As illustrated in Fig. 8, we categorize the tuning knobs that impact the performance of segmented matmul into three distinct categories: *Algorithms*, *Scheduling*, and *Resources*. Given the complexity of these knobs' relationship with the performance of segmented matmul, a one-size-fits-all configuration is unlikely to yield optimal results. Consequently, FASTEN employs an autotuning approach to identify the most effective configuration based on the specific characteristics of input shapes and types.

FASTEN benchmarks the given problem using a set of configurations, runs a few iterations, obtains the median running time, and then selects the best configuration to associate with the corresponding "key". Subsequently, inputs that trigger the same "key" will bypass further autotuning. The following paragraphs detail our approach to key selection, configuration generation, and pruning.

Key Design. In FASTEN, individual routing tables are created for the computation of the forward, input gradient, and weight gradient. Metadata information is associated with each table, such as the ratio of large blocks ($\mathcal{R}$) and the utilization factor ($\mathcal{U}$). The keys are chosen based on their impact on compute and parallel efficiency. Although we could naively use the combination of *algorithm*, $\{|\tau|$, for τ in $\mathcal{T}\}$, K, and Q as the key, this approach risks excessive reruns due to minor variations in the number of edges or edge types.

To minimize rerun cost, we have optimized our key selection to include *algorithm*, $stddev(|\tau|)$, $average(|\tau|)$, K, and Q. Only a large difference of $stddev(|\tau|)$ and $average(|\tau|)$ will trigger the rerun of the tuning process.

Configuration Generation and Pruning. A straightforward approach would be to set up several candidate values for each tuning knob, enumerate all possible configurations, and benchmark each configuration exhaustively. A simple combination could include more than 20,000 configurations, leading to excessive tuning time.

To address this, we have established the following set of configuration pruning rules:

- *Algorithm-based Pruning*: The register blocking algorithm is preferable when the accumulation dimension is small (i.e., ≤ 32). The 3D parallelization algorithm is reserved for cases where the accumulation dimension varies. Otherwise, the basic algorithm is chosen.
- *Efficiency-based Pruning*: We evaluate the parallel and compute efficiency of configurations, focusing only on those with the highest efficiency. We approximated $Hit_{\text{L2}}^{\text{tile}}$ in Equation 18 based on the maximum tile reuse rate, considering the input shape and the tile values. Additionally, to account for $Time_{\text{sync}}$, the configurations are separated into groups according to their number of threads per CTA, ranked according to efficiency.
- *Resource Constraint-based Pruning*: We estimate the shared memory required for each configuration and exclude those that exceed the maximum shared memory capacity. Configurations that trigger excessive register spills during JIT compilation are also discarded.
- *Shape-based Pruning*: Configurations with tile dimensions or tile dimension multiples N_{stages} are eliminated to avoid wasteful instruction cycles.
- *Rule-based Pruning*: The dual kernel approach is reserved for scenarios requiring determinism, as requested by users. Dynamic tiling is applied only when the Utilization Factor is low.

7 IMPLEMENTATION

This section describes the implementation details of how FASTEN is integrated with GNN frameworks.

7.1 Integration in PyG

In addition to access to raw segmented matmul APIs in FASTEN, we have integrated FASTEN operators with PyG [10], a state-of-the-art GNN framework supporting various HGNNs, to demonstrate the performance enhancements provided by FASTEN and offer user-friendly interfaces.

We provide utilities to assist users in converting existing HGNN modules to those accelerated by FASTEN. `FastenModule(module)` is a wrapper for existing HGNN modules, maintaining structural similarity in module initialization and forward call interfaces as per PyG standards. The key addition is the `TensorSlice(data, types)` data structure as a parameter for the forward function, which stores the input data and its corresponding types for each segment. This data structure incorporates a `create_routing_table` API and a `get_routing_table` API, both utilized internally by each `FastenModule`.

For creating a routing table, we offer two approaches. In the default mode, most optimal settings based on our performance models are used to create the routing table before executing a segmented matmul. Alternatively, with `autotune` set to true, we benchmark configurations, select the most performant one, and link this configuration with input shapes as keys in the `TensorSlice`, facilitating configuration reuse for identical keys.

7.2 Adapting Different HGNNs

FASTEN supports various HGNNs, including RGCN [41], RGAT [7], and HGT [20], all utilizing segmented matmul in their implementations.

RGCN and RGAT. Both RGCN and RGAT employ segmented matmul in their message passing layers, segmenting operations based on the relational edge types. In practice, multiple RGCN or RGAT modules are stacked with consistent feature sizes, allowing the routing table to be computed once, stored in `TensorSlice`, and reused across modules.

HGT. HGT leverages meta information that contains node and edge types in heterogeneous graphs to parameterize weight matrices. Segmented matmul is utilized for both node and edge types related computations. We preprocess the meta information to construct two `TensorSlice` objects accordingly, one for node-type data and another for edge-type data, and pass them to the forward function.

Optimizations. For full graph processing, where node or edge types remain constant during GNN computation, we sort the data according to its type, cache the related `TensorSlice` object, and reuse this `TensorSlice` object when the data is loaded again in iterative training. In scenarios involving subgraph sampling, we sort the data each time it is sampled and use the related `TensorSlice` only for the current iteration. FASTEN also facilitates the fusion of simple element-wise prologues or epilogues. For instance, when using bias in the prologue, we load the bias segment based on the type handled by each CTA into registers, accumulate bias with each segment's output, and store the results in global memory to minimize the overhead of transferring results from global memory to registers in the bias kernel. Unlike fusion in dense operators, which reduces the launch of only one kernel, fusion of segmented operators in the prologue can reduce the launch of up to $|\mathcal{T}|$ kernel.

8 EVALUATION

Platforms. We evaluated FASTEN's performance on three distinct platforms, as detailed in Table 2, all using the TF32 format with tensor cores.

Datasets. We used both real and synthesized random datasets. The properties of the real datasets used are summarized in Table 3. The AIFB, MUTAG, BGS, and AM datasets [41] only have edge types. While other datasets, including Freebase [6], DBLP [11], ACM [30], and IMDB [2], have both edge and node types.

Our experiments are categorized into two types: the first focused on operator performance across different datasets, and the second on end-to-end training performance, benchmarked against existing implementations of popular HGNNs. Experiments were measured using the Proton profiler [58].

8.1 Operator Performance

In this section, we compare FASTEN's segmented matmul performance against two state-of-the-art variants: *CUTLASS* [39], which adopts the *grouped matmul* algorithm, and *cuBLAS*čitecuBLAS, which uses the *loop over matmuls* method that launches multiple kernels. FASTEN's A100 and RTX4090 implementations employ `mma`

Platform	Memory	CPU	GPU Specs
GH200	480GB	NVIDIA Grace CPU Superchip	96GB GPU Memory, 132 SMs, 989 TF32 TFLOP/s, 4TB/s Bandwidth
A100 SXM	256GB	AMD EPYC 7543	80GB GPU Memory, 108 SMs, 156 TF32 TFLOP/s, 2TB/s Bandwidth
RTX4090	512GB	AMD Ryzen Threadripper PRO5975WX	24GB GPU Memory, 128 SMs, 82.6 TF32 TFLOPS, 1TB/s Bandwidth

Table 2: Evaluation Platforms

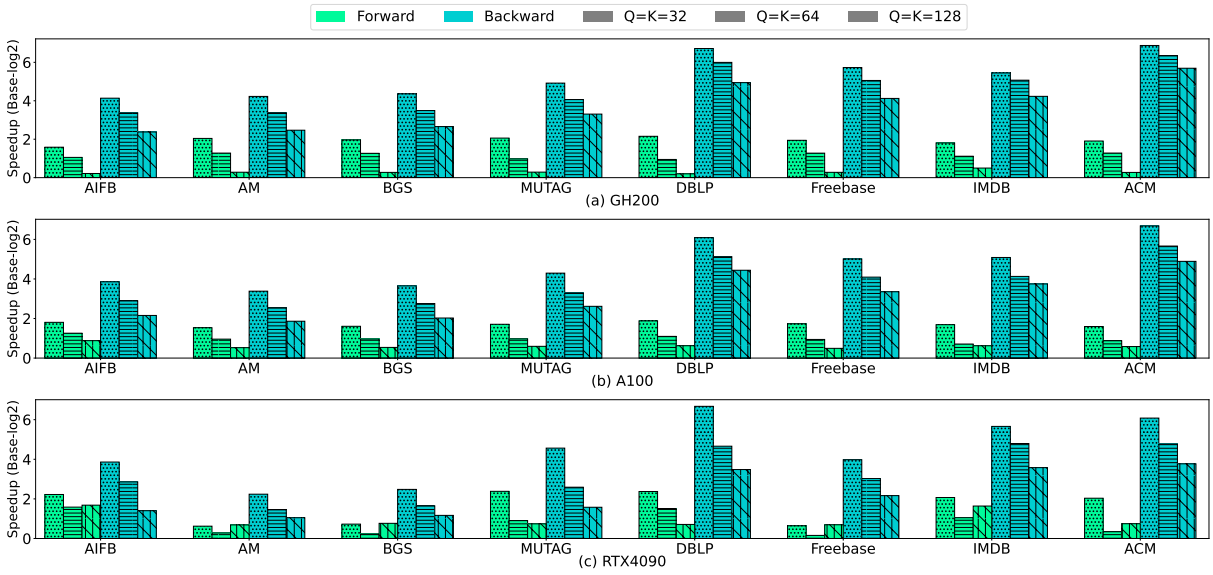


Figure 9: Operator performance speedups of FASTEN over CUTLASS.

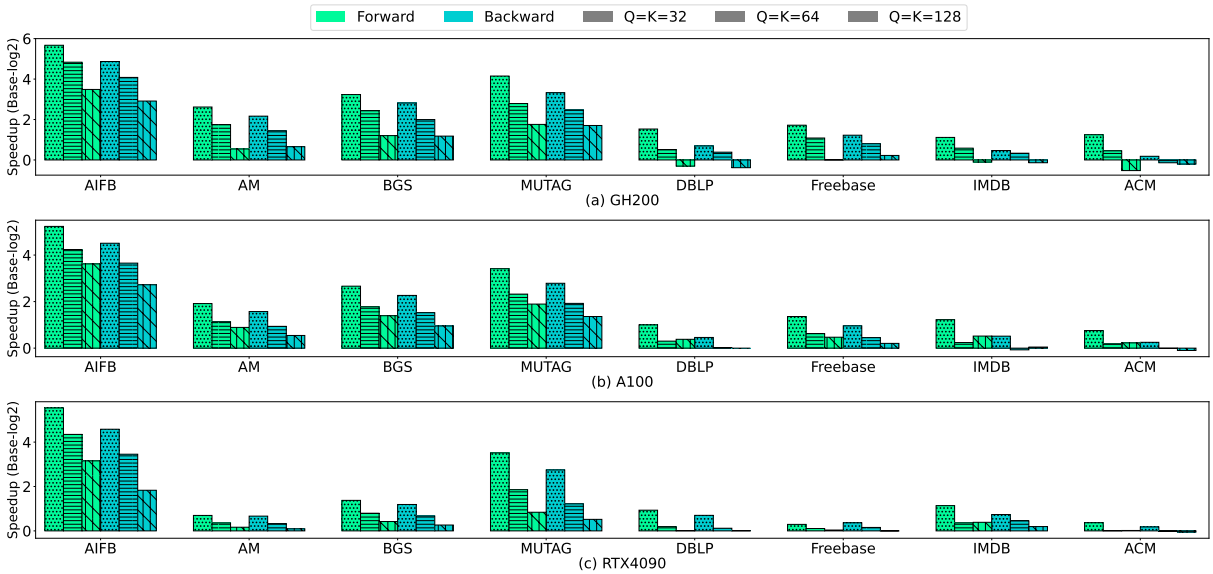


Figure 10: Operator performance speedups of FASTEN over cuBLAS.

instructions to utilize tensor cores, whereas the GH200 implementation employs both `mma` and `wgmma` instructions. The TMA (Tensor Memory Access) function unit was not utilized in our GH200 implementation for memory transfers. Experiments were performed with CUDA 12.2 [40] and PyG at commit `a37af2e`.

Dataset	Nodes	Edges	Types
AIFB	8,285	58,086	90
MUTAG	23,644	148,454	46
BGS	333,845	1,832,398	206
AM	1,666,764	11,976,642	266
Freebase	180,098	1,057,688	36 (Edge), 8 (Node)
DBLP	26,128	119,783	3 (Edge), 4 (Node)
ACM	10,942	273,936	4 (Edge), 4 (Node)
IMDB	21,420	43,321	3 (Edge), 4 (Node)

Table 3: Real Datasets Properties

Real Datasets. We evaluated the performance of the forward and backward phases for different feature sizes ($k = Q = 32, 64, 128$) on the GH200, A100, and RTX 4090 platforms. The forward phase corresponds to Equation 3, and the backward phase computes Equations 4 and 5. Fig. 9 and Fig. 10 illustrate the performance on eight real datasets, all using edge types to segment input samples.

FASTEN consistently outperforms CUTLASS in both phases. In the forward phase, FASTEN achieved speedups ranging from 1.11× to 5.21×; in the backward phase, the speedups ranged from 2.07× to 117.54×. FASTEN achieved higher speedups in the forward phase when the feature size is low because the CUTLASS implementation did not select appropriate tile sizes and lacked optimizations such as tile grouping and register blocking, as demonstrated in our incremental improvement studies. As the number of feature sizes increases, FASTEN continues to outperform CUTLASS, due to reduced indexing overhead. In the backward phase, FASTEN significantly surpasses CUTLASS, primarily due to the benefits of 3D parallelism, which leverages the routing table to balance the workload along the accumulation dimension.

Comparing with cuBLAS, FASTEN achieved speedups ranging from 0.69× to 51.01× in the forward phase and from 0.76× to 29.25× in the backward phase. When the number of types is small, cuBLAS efficiently handles both phases as it only needs to launch very few kernels, each of which is highly optimized even when the accumulation dimension is large. However, as the number of types increases, as observed in datasets like AIFB, MUTAG, BGS, and AM, cuBLAS's performance decreases due to the increased overhead from launching more kernels and the reduced GPU utilization rate of each kernel. Overall, FASTEN achieved an average speedup of 5.73× and 3.72× over cuBLAS in forward and backward phases correspondingly.

Synthetic Datasets. In addition to real datasets, which typically have unbalanced distributions, we compared the performance of FASTEN with that of CUTLASS using synthetic datasets. To generate these datasets, we set the total number of instances at one million (i.e., $\sum_{\tau \in \mathcal{T}} |\tau|$). Each instance was randomly assigned to one of the $\mathcal{T}$ types, using a uniform probability distribution to assess performance when each type is associated with a similar number of instances. We evaluated the number of types ranging from 100 to 1900 in steps of 200. We also calculated the upper performance bound limited by DRAM bandwidth using the Roofline model [49], which we denote as the theoretical peak performance. Fig. 11 (a) and 11 (b) show the performance of the three variants using feature sizes of 32 and 128 in the forward and backward phases.

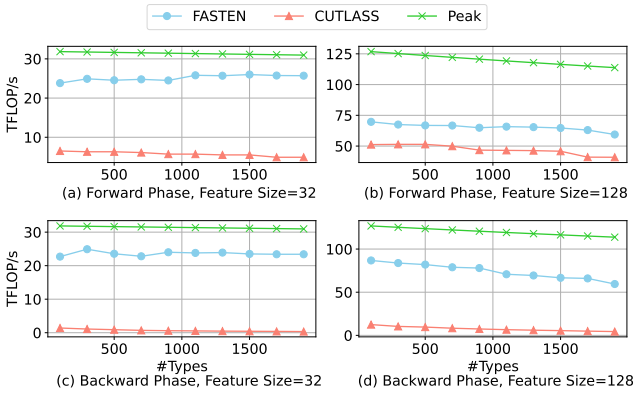


Figure 11: Performance comparison between FASTEN and CUTLASS on randomly generated instances on the GH200 GPU.

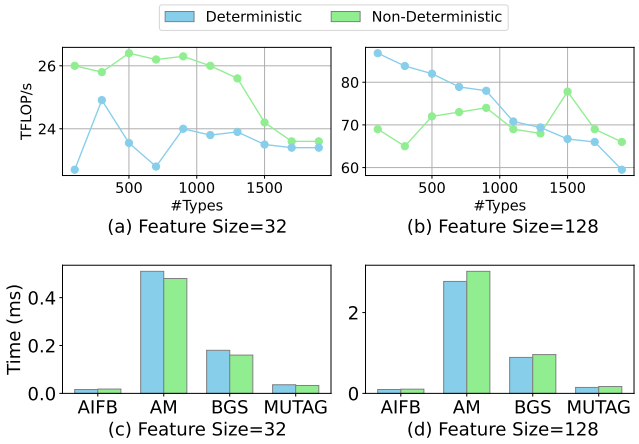


Figure 12: Performance comparison of FASTEN's deterministic and non-deterministic backward implementations using real and synthetic datasets on the GH200 GPU.

We observed that FASTEN is highly optimized, achieving between 55% and 84% of the peak performance. In comparison, CUTLASS achieves only 1%-40% of the peak performance. Interestingly, smaller feature sizes often reach higher peaks compared to their larger counterparts.There are two main factors contributing to the performance degradation observed with larger feature sizes. Firstly, our detailed analysis of instruction-level profiling results [59] reveals significant stalls due to barrier instructions in the code generated by *ptxas*. These stalls occur while waiting for `WGMMA` operations. By grouping these `WGMMA` operations and synchronizing them less frequently, we might reduce these overheads. Additionally, when the feature size is set to 128, the performance of the forward pass is noticeably poorer than that of the backward phase. This discrepancy arises from the size of the accumulation dimension (τ), which leads to a tile size that is too large for effective register blocking, but too small to allow the overlap of compute tasks with load and store operations.

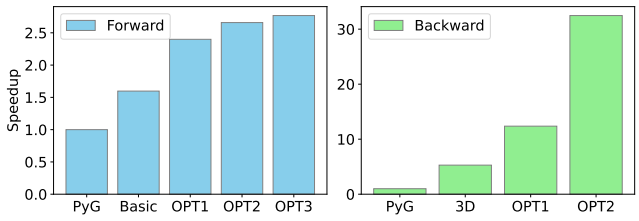


Figure 13: Incremental optimization effects on the forward and backward phases. OPT1: tile size tuning, OPT2: tile grouping & reordering, OPT3: register blocking.

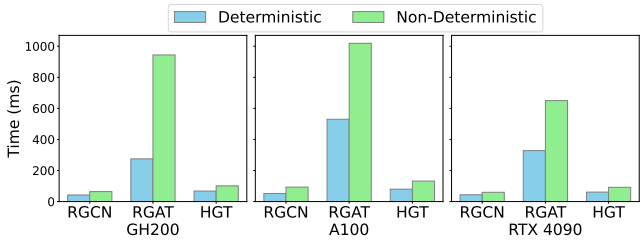


Figure 14: End-to-end training time (four iterations) of FASTEN comparing with PyG. RGCN and RGAT used the AM dataset, and HGT used the Freebase dataset.

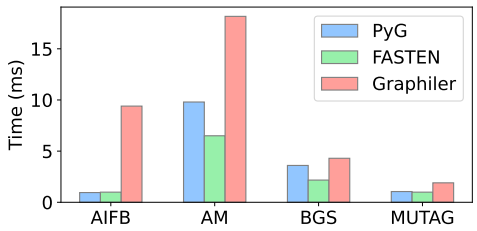


Figure 15: End-to-end inference time of FASTEN comparing with Graphiler and PyG.

Deterministic vs. Non-Deterministic Comparisons. Fig. 12 (a) and (b) illustrate that the deterministic implementations of FASTEN in the backward phase perform comparably to the non-deterministic version across various types using randomly generated datasets. Performance was also assessed on real datasets with feature sizes of 32 and 128, respectively, as shown in Figs. 12 (c) and (d). Overall, the non-deterministic version demonstrates performance similar to its deterministic counterpart on real datasets. Notably, for larger feature sizes, the deterministic version outperforms the non-deterministic one. This improvement is attributed to the fact that the deterministic version employs `store` operations instead of `atomicAdd`, resulting in higher throughput. In contrast, for smaller feature sizes, the second kernel in the dual kernel strategy takes a relatively larger portion of the total runtime. Thus, the non-deterministic version outperforms the deterministic version because there is no additional overhead in accessing the *intermediate buffer*.

Incremental Improvement. We also evaluated the incremental effects of optimizations employed in FASTEN's segmented matmul operator, which are divided into multiple stages: initial, tile size tuning using the autotuner, tile grouping & reordering, and register blocking. The initial implementation of the forward phase employs the basic Algorithm 1, while the backward phase utilizes the 3D parallelization algorithm with a N_{blocks} of one. Fig. 13 illustrates the performance of both the forward and backward phases using the AM dataset with a feature size of 32 on the GH200 GPU.

For a fair comparison, we used the same number of threads, tile sizes, stages, and the same `mma` modifier as CUTLASS initially uses. We observe that our initial forward implementation (Basic) outperforms CUTLASS by 1.61×; FASTEN incurs lower indexing costs because its routing table stores only a single dimension of varying sizes. Additionally, our initial backward implementation (3D) is 5.34× faster than PyG, due to the adoption of 3D parallelization for workload balance. By selecting appropriate tile sizes, we can optimize both forward and backward phases. Tile grouping achieves a 1.10× speedup in the forward phase, and a 2.42× speedup in the backward phase. This higher speedup in the backward phase can be attributed to the larger workload per CTA after grouping, which also reduces conflicts in atomic operations. Register blocking results in an additional 1.04× speedup in the forward phase.

8.2 End-to-end Performance

In this section, we compare the end-to-end performance of FASTEN with PyG and Graphiler. RGCN and HGT were evaluated using a feature size of 32, and GAT was assessed with a feature size of 8.

FASTEN vs. PyG. In the default implementation of PyG's examples, HGT and RGCN adopt the CUTLASS grouped matmul approach for segmented matmul during the forward phase. However, during the backward phase, they employ the cuBLAS *loop over matmul* method. PyG's RGAT differs by utilizing the *batched matmul* method for segmented matmul, leading to a significant increase in memory footprint. As a result, PyG's RGAT can only be operated with a restricted feature size. In contrast, FASTEN's RGAT model is capable of supporting feature sizes up to 32, even on RTX4090. As depicted in Fig. 14, FASTEN outperforms PyG in all HGNNs on three platforms, achieving a speedup ranging between 1.37× and 3.53×. Notably, the maximum speedup was observed on the GH200 platform with the RGAT model. We also made an interesting observation regarding the feature size: When operating with smaller feature sizes, PyTorch's `bmm` operation defaults to using `ffma` instructions instead of tensor cores on the GH200 platform. This contrasts with its behavior on the A100 and RTX4090 platforms, where it uses tensor cores regardless of feature sizes.

FASTEN vs. Graphiler. Graphiler depends on CUDA 11 and thus has compatibility issues with newer GPU platforms such as the RTX4090 and GH200. Consequently, our evaluation of Graphiler was limited to the A100 platform. In experiments, we focused on the forward phase performance[2] of Graphiler's RGCN using a feature size of 32, comparing it with PyG and FASTEN across four real

[2]Graphiler only supports compilation of the forward computation graph.

datasets. In Fig. 15, our findings revealed that FASTEN achieved a 1.9-9.4× speedup against Graphiler. More notably, Graphiler demonstrated slower performance compared to PyG in all datasets[3]. This outcome stands in stark contrast to the optimization effects claimed in Graphiler's original publication. Considering that Graphiler was developed two years ago, a period during which PyG had not yet adopted CUTLASS for segmented matmul operations, this performance discrepancy highlights a crucial insight: the mere fusion of operations in computation graphs, without integrating low-level optimizations, is inadequate to achieve optimal performance for sophisticated operators.

9 RELATED WORK

This section provides an overview of related work on GNN performance optimizations and compares them with FASTEN.

CTA Scheduling. CTA scheduling is essential to maximize efficiency on GPUs. NVIDIA's default hardware-based scheduling engine Gigathread [50] initially follows a round-robin policy and then updates to dynamic scheduling [27]. Lee et al. [25] designed a new CTA scheduler that monitors workloads and dynamically determines the number of CTAs assigned to each core. Kim et al. [24] studied improving CTA scheduling specifically for GEMM and convolution algorithms. In addition to the simulator-based approaches mentioned above, there are software-based solutions. Li et al. [26] proposed CTA clustering techniques to group CTAs with potential reuse on the same SM, in order to maximize cache reuse. Similarly, Ukarande et al. [46] designed software clustering techniques to improve cache locality for texture accesses in game applications. Unlike these general approaches, FASTEN's CTA scheduling is specifically tailored for segmented matmul operations and uses a routing table for guidance.

Irregular Batch Matmul. We refer to operations that support various indexing mechanisms and shapes as irregular batch matmul, as they are not restricted by the size constraints inherent in standard batch matmul. CUTLASS [38] has provided general grouped matmul operators, but these often fall short in real-world scenarios in terms of performance and functionality. Unlike CUTLASS, MAGMA [32] supports variable dimensions by grouping matrices with different shapes through the "z" dimension when starting a kernel. Li et al. [28] further optimized this method for small matrices by grouping matrices according to the tile size used. Block sparse matmul [15] is similar to the segmented matmul, as only one dimension can vary, but the result is stored in a global sparse matrix instead of a batch of dense matrices. MEGABLOCKS [12] optimizes the block sparse computation in Mixture-of-Experts [42] modules with a new matrix format.

GNN Performance Optimizations. Previous studies that improve inefficiencies in homogeneous GNNs [8, 22, 48] cannot be directly adopted for HGNNs due to their new sparsity in relational dimension. Most existing research on HGNNs has instead focused on high-level optimizations such as scheduling mechanisms [34], matrix format conversion [45], kernel fusion [16, 55], and IR designs [51, 55, 56]. These approaches either leverage existing kernels in cuBLAS or CUTLASS, or use tensor cores, often without detailed analysis and optimizations. Furthermore, existing compiler-based approaches [51, 55] are limited to the forward phase. In contrast, FASTEN investigates low-level inefficiencies and redesigns the segment matmul operator, demonstrating improved efficiency across multiple GPU architectures.

[3]Graphiler utilizes smaller datasets, which selectively filter out nodes and edges from the datasets used in FASTEN and PyG evaluations.

10 DISCUSSIONS

Emerging GPU Features. It is noteworthy that, beginning with the Hopper architecture, NVIDIA has introduced support for CTA clusters and hardware-level data transfer. We will explore this advancement, which could provide more granular control in segmented matmul as the weight matrix is shared among instances of the same type. Furthermore, our future work will include performance optimizations for other segmented operations on GPUs, such as segmented sort, scan, and attention.

Scalability. To support large graphs, such as OGB datasets [19], users can apply FASTEN to multiple GPUs by dividing segments of different types on different GPUs. However, this initial approach is not optimal due to the overlooking of communication issues, such as loading redundant weight matrices and aggregating weight gradients on multiple GPUs. To make FASTEN more efficient, we plan to develop a centralized scheduling module to address these issues.

Portability. FASTEN has been modularized into *nn*, *op*, and *kernel* layers. While the *nn* layer is integrated with PyG, the *op* layer or the *kernel* layer does not depend on PyG. Therefore, we envision that it would be straightforward to integrate the op or kernel layer with other GNN frameworks such as DGL [47].

11 CONCLUSIONS

This paper introduces FASTEN —a high-performance library tailored for segmented matrix multiplication operations. The motivation behind FASTEN is that there is a lack of algorithms and implementations on GPUs well suited for batch processing segmented with irregular and different sizes, which commonly exist in training heterogeneous graph neural networks. FASTEN encompasses a set of sophisticated strategies that focuses primarily on reducing indexing costs, effective CTA scheduling, and data reuse across CTAs. Evaluation results show that FASTEN is significantly faster than state-of-the-art vendor-provided operator libraries [38], as well as research tools [55] that optimize compiler passes. Future work on FASTEN will be extending the methodology on other segmented operations, utilizing advanced GPU features, and fine-tuning it for multi-GPU training.

ACKNOWLEDGMENTS

This project was supported by resources provided by the Office of Research Computing at George Mason University (URL: https://orc.gmu.edu) and George Mason University's Faculty Startup Fund of the Computer Science Department. This project was also supported in part by National Science Foundation (Award Numbers 2018631 and 2316201).

REFERENCES

[1] Miltiadis Allamanis, Marc Brockschmidt, and Mahmoud Khademi. 2017. Learning to represent programs with graphs. *arXiv preprint arXiv:1711.00740* (2017).

[2] Yunsheng Bai, Hao Ding, Song Bian, Ting Chen, Yizhou Sun, and Wei Wang. 2019. Simgnn: A neural network approach to fast graph similarity computation. In *Proceedings of the twelfth ACM international conference on web search and data mining*. 384–392.

[3] Albert-Laszlo Barabasi and Zoltan N Oltvai. 2004. Network biology: understanding the cell's functional organization. *Nature reviews genetics* 5, 2 (2004), 101–113.

[4] Ganesh Bikshandi and Jay Shah. 2023. A Case Study in CUDA Kernel Fusion: Implementing FlashAttention-2 on NVIDIA Hopper Architecture using the CUTLASS Library. arXiv:2312.11918 [cs.LG]

[5] Alpheus Bingham and Dwayne Spradlin. 2011. *The long tail of expertise*. Pearson Education.

[6] Antoine Bordes, Nicolas Usunier, Alberto Garcia-Duran, Jason Weston, and Oksana Yakhnenko. 2013. Translating embeddings for modeling multi-relational data. *Advances in neural information processing systems* 26 (2013).

[7] Dan Busbridge, Dane Sherburn, Pietro Cavallo, and Nils Y Hammerla. 2019. Relational graph attention networks. *arXiv preprint arXiv:1904.05811* (2019).

[8] Zhaodong Chen, Mingyu Yan, Maohua Zhu, Lei Deng, Guoqi Li, Shuangchen Li, and Yuan Xie. 2020. fuseGNN: Accelerating graph convolutional neural network training on GPGPU. In *Proceedings of the 39th International Conference on Computer-Aided Design*. 1–9.

[9] David Easley, Jon Kleinberg, et al. 2012. Networks, crowds, and markets. *Cambridge Books* (2012).

[10] Matthias Fey and Jan Eric Lenssen. 2019. Fast graph representation learning with PyTorch Geometric. *arXiv preprint arXiv:1903.02428* (2019).

[11] Xinyu Fu, Jiani Zhang, Ziqiao Meng, and Irwin King. 2020. Magnn: Metapath aggregated graph neural network for heterogeneous graph embedding. In *Proceedings of The Web Conference 2020*. 2331–2341.

[12] Trevor Gale, Deepak Narayanan, Cliff Young, and Matei Zaharia. 2023. MegaBlocks: Efficient Sparse Training with Mixture-of-Experts. *Proceedings of Machine Learning and Systems* 5 (2023).

[13] Thomas Gaudelet, Ben Day, Arian R Jamasb, Jyothish Soman, Cristian Regep, Gertrude Liu, Jeremy BR Hayter, Richard Vickers, Charles Roberts, Jian Tang, et al. 2021. Utilizing graph machine learning within drug discovery and development. *Briefings in bioinformatics* 22, 6 (2021), bbab159.

[14] Justin Gilmer, Samuel S Schoenholz, Patrick F Riley, Oriol Vinyals, and George E Dahl. 2017. Neural message passing for quantum chemistry. In *International conference on machine learning*. PMLR, 1263–1272.

[15] Scott Gray, Alec Radford, and Diederik P. Kingma. 2017. Block-Sparse GPU Kernels. https://blog.openai.com/block-sparse-gpu-kernels/. Accessed: 1-14-2024.

[16] Yuntao Gui, Yidi Wu, Han Yang, Tatiana Jin, Boyang Li, Qihui Zhou, James Cheng, and Fan Yu. 2022. HGL: accelerating heterogeneous GNN training with holistic representation and optimization. In *SC22: International Conference for High Performance Computing, Networking, Storage and Analysis*. IEEE, 1–15.

[17] Yixin Guo, Pengcheng Li, Yingwei Luo, Xiaolin Wang, and Zhenlin Wang. 2022. Exploring gnn based program embedding technologies for binary related tasks. In *Proceedings of the 30th IEEE/ACM International Conference on Program Comprehension*. 366–377.

[18] Kshitij Gupta, Jeff A Stuart, and John D Owens. 2012. *A study of persistent threads style GPU programming for GPGPU workloads*. IEEE.

[19] Weihua Hu, Matthias Fey, Marinka Zitnik, Yuxiao Dong, Hongyu Ren, Bowen Liu, Michele Catasta, and Jure Leskovec. 2020. Open graph benchmark: Datasets for machine learning on graphs. *Advances in neural information processing systems* 33 (2020), 22118–22133.

[20] Ziniu Hu, Yuxiao Dong, Kuansan Wang, and Yizhou Sun. 2020. Heterogeneous graph transformer. In *Proceedings of the web conference 2020*. 2704–2710.

[21] Guyue Huang, Yang Bai, Liu Liu, Yuke Wang, Bei Yu, Yufei Ding, and Yuan Xie. 2023. ALCOP: Automatic Load-Compute Pipelining in Deep Learning Compiler for AI-GPUs. *Proceedings of Machine Learning and Systems* 5 (2023).

[22] Guyue Huang, Guohao Dai, Yu Wang, and Huazhong Yang. 2020. 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*. IEEE, 1–12.

[23] Kezhao Huang, Jidong Zhai, Zhen Zheng, Youngmin Yi, and Xipeng Shen. 2021. 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*. 119–132.

[24] Hyeonjin Kim and William J. Song. 2023. LAS: Locality-Aware Scheduling for GEMM-Accelerated Convolutions in GPUs. *IEEE Transactions on Parallel and Distributed Systems* 34, 5 (2023), 1479–1494. https://doi.org/10.1109/TPDS.2023.3247808

[25] Minseok Lee, Seokwoo Song, Joosik Moon, John Kim, Woong Seo, Yeongon Cho, and Soojung Ryu. 2014. Improving GPGPU resource utilization through alternative thread block scheduling. In *2014 IEEE 20th International Symposium on High Performance Computer Architecture (HPCA)*. 260–271. https://doi.org/10.1109/HPCA.2014.6835937

[26] Ang Li, Shuaiwen Leon Song, Weifeng Liu, Xu Liu, Akash Kumar, and Henk Corporaal. 2017. Locality-Aware CTA Clustering for Modern GPUs. In *Proceedings of the Twenty-Second International Conference on Architectural Support for Programming Languages and Operating Systems* (Xi'an, China) *(ASPLOS '17)*. Association for Computing Machinery, New York, NY, USA, 297–311. https://doi.org/10.1145/3037697.3037709

[27] Ang Li, Shuaiwen Leon Song, Mark Wijtvliet, Akash Kumar, and Henk Corporaal. 2016. SFU-driven transparent approximation acceleration on GPUs. In *Proceedings of the 2016 International Conference on Supercomputing*. 1–14.

[28] Xiuhong Li, Yun Liang, Shengen Yan, Liancheng Jia, and Yinghan Li. 2019. A Coordinated Tiling and Batching Framework for Efficient GEMM on GPUs. In *Proceedings of the 24th Symposium on Principles and Practice of Parallel Programming* (Washington, District of Columbia) *(PPoPP '19)*. Association for Computing Machinery, New York, NY, USA, 229–241. https://doi.org/10.1145/3293883.3295734

[29] Yangyang Li, Yipeng Ji, Shaoning Li, Shulong He, Yinhao Cao, Yifeng Liu, Hong Liu, Xiong Li, Jun Shi, and Yangchao Yang. 2021. Relevance-Aware Anomalous Users Detection in Social Network via Graph Neural Network. In *2021 International Joint Conference on Neural Networks (IJCNN)*. 1–8. https://doi.org/10.1109/IJCNN52387.2021.9534136

[30] Qingsong Lv, Ming Ding, Qiang Liu, Yuxiang Chen, Wenzheng Feng, Siming He, Chang Zhou, Jianguo Jiang, Yuxiao Dong, and Jie Tang. 2021. Are we really making much progress? revisiting, benchmarking and refining heterogeneous graph neural networks. In *Proceedings of the 27th ACM SIGKDD conference on knowledge discovery & data mining*. 1150–1160.

[31] Sangkug Lym, Donghyuk Lee, Mike O'Connor, Niladrish Chatterjee, and Mattan Erez. 2019. DeLTA: GPU performance model for deep learning applications with in-depth memory system traffic analysis. In *2019 IEEE international symposium on performance analysis of systems and software (ISPASS)*. IEEE, 293–303.

[32] Rajib Nath, Stanimire Tomov, and Jack Dongarra. 2010. An improved magma gemm for fermi graphics processing units. *The International Journal of High Performance Computing Applications* 24, 4 (2010), 511–515.

[33] Maximilian Nickel, Kevin Murphy, Volker Tresp, and Evgeniy Gabrilovich. 2015. A review of relational machine learning for knowledge graphs. *Proc. IEEE* 104, 1 (2015), 11–33.

[34] Israt Nisa, Minjie Wang, Da Zheng, Qiang Fu, Umit Çatalyürek, and George Karypis. 2023. Optimizing Irregular Dense Operators of Heterogeneous GNN Models on GPU. In *2023 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*. 199–206. https://doi.org/10.1109/IPDPSW59300.2023.00042

[35] NVIDIA. 2023. *Parallel Thread Execution ISA*. https://docs.nvidia.com/cuda/parallel-thread-execution/index.html Accessed: 12-25-2023.

[36] NVIDIA Corporation. 2023. cuBLAS: The NVIDIA CUDA Basic Linear Algebra Subroutines library. https://developer.nvidia.com/cublas. Accessed: 12-16-2023.

[37] NVIDIA Corporation. 2023. cuSPARSE: Basic Linear Algebra for Sparse Matrices on NVIDIA GPUs. https://developer.nvidia.com/cusparse. Accessed: 12-16-2023.

[38] NVIDIA Corporation. 2023. CUTLASS: CUDA C++ template abstractions for implementing high-performance matrix-matrix multiplication. https://github.com/NVIDIA/cutlass. Accessed: 12-16-2023.

[39] NVIDIA Corporation. 2023. CUTLASS Grouped Kernel Schedulers. https://github.com/NVIDIA/cutlass/blob/main/media/docs/grouped_scheduler.md. Accessed: 12-16-2023.

[40] NVIDIA Corporation. 2024. CUDA Toolkit Documentation. https://developer.nvidia.com/cuda-toolkit Accessed: 01-06-2024.

[41] Michael Schlichtkrull, Thomas N Kipf, Peter Bloem, Rianne Van Den Berg, Ivan Titov, and Max Welling. 2018. Modeling relational data with graph convolutional networks. In *The Semantic Web: 15th International Conference, ESWC 2018, Heraklion, Crete, Greece, June 3–7, 2018, Proceedings 15*. Springer, 593–607.

[42] Noam Shazeer, Azalia Mirhoseini, Krzysztof Maziarz, Andy Davis, Quoc Le, Geoffrey Hinton, and Jeff Dean. 2017. Outrageously large neural networks: The sparsely-gated mixture-of-experts layer. *arXiv preprint arXiv:1701.06538* (2017).

[43] PyTorch Geometric Team. 2024. PyTorch Geometric (PyG). https://github.com/pyg-team/pytorch_geometric/blob/master/torch_geometric/nn/conv/rgcn_conv.py Accessed: 04-04-2024.

[44] PyTorch Geometric Team. 2024. PyTorch Geometric (PyG) Lib. https://github.com/pyg-team/pyg-lib/blob/master/pyg_lib/csrc/ops/autograd/matmul_kernel.cpp Accessed: 04-04-2024.

[45] Thiviyan Thanapalasingam, Lucas van Berkel, Peter Bloem, and Paul Groth. 2022. Relational graph convolutional networks: a closer look. *PeerJ Computer Science* 8 (2022), e1073.

[46] Aditya Ukarande, Suryakant Patidar, and Ram Rangan. 2021. Locality-aware cta scheduling for gaming applications. *ACM Transactions on Architecture and Code Optimization (TACO)* 19, 1 (2021), 1–26.

[47] Minjie Wang, Da Zheng, Zihao Ye, Quan Gan, Mufei Li, Xiang Song, Jinjing Zhou, Chao Ma, Lingfan Yu, Yu Gai, et al. 2019. Deep graph library: A graph-centric, highly-performant package for graph neural networks. *arXiv preprint*

arXiv:1909.01315 (2019).

[48] Yuke Wang, Boyuan Feng, Gushu Li, Shuangchen Li, Lei Deng, Yuan Xie, and Yufei Ding. 2021. GNNAdvisor: An adaptive and efficient runtime system for GNN acceleration on GPUs. In *15th USENIX symposium on operating systems design and implementation (OSDI 21)*. 515–531.

[49] Samuel Williams, Andrew Waterman, and David Patterson. 2009. Roofline: an insightful visual performance model for multicore architectures. *Commun. ACM* 52, 4 (2009), 65–76.

[50] Craig M Wittenbrink, Emmett Kilgariff, and Arjun Prabhu. 2011. Fermi GF100 GPU architecture. *IEEE Micro* 31, 2 (2011), 50–59.

[51] Kun Wu, Mert Hidayetoğlu, Xiang Song, Sitao Huang, Da Zheng, Israt Nisa, and Wen-mei Hwu. 2023. PIGEON: Optimizing CUDA Code Generator for End-to-End Training and Inference of Relational Graph Neural Networks. *arXiv preprint arXiv:2301.06284* (2023).

[52] Shiwen Wu, Fei Sun, Wentao Zhang, Xu Xie, and Bin Cui. 2022. Graph neural networks in recommender systems: a survey. *Comput. Surveys* 55, 5 (2022), 1–37.

[53] Zonghan Wu, Shirui Pan, Fengwen Chen, Guodong Long, Chengqi Zhang, and S Yu Philip. 2020. A comprehensive survey on graph neural networks. *IEEE transactions on neural networks and learning systems* 32, 1 (2020), 4–24.

[54] Zhenqin Wu, Bharath Ramsundar, Evan N Feinberg, Joseph Gomes, Caleb Geniesse, Aneesh S Pappu, Karl Leswing, and Vijay Pande. 2018. MoleculeNet: a benchmark for molecular machine learning. *Chemical science* 9, 2 (2018), 513–530.

[55] Zhiqiang Xie, Minjie Wang, Zihao Ye, Zheng Zhang, and Rui Fan. 2022. Graphiler: Optimizing graph neural networks with message passing data flow graph. *Proceedings of Machine Learning and Systems* 4 (2022), 515–528.

[56] Zihao Ye, Ruihang Lai, Junru Shao, Tianqi Chen, and Luis Ceze. 2023. 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*. 660–678.

[57] Jie Zhou, Ganqu Cui, Shengding Hu, Zhengyan Zhang, Cheng Yang, Zhiyuan Liu, Lifeng Wang, Changcheng Li, and Maosong Sun. 2020. Graph neural networks: A review of methods and applications. *AI open* 1 (2020), 57–81.

[58] Keren Zhou. 2024. Proton: A Profiler for Triton. https://github.com/openai/triton/tree/main/third_party/proton Accessed: 04-021-2024.

[59] Keren Zhou, Xiaozhu Meng, Ryuichi Sai, and John Mellor-Crummey. 2021. GPA: A GPU performance advisor based on instruction sampling. In *2021 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*. IEEE, 115–125.