Accelerating Sparse-Sparse-Dense Graph Matrix Operator using Asynchronous Execution in GPUs

Hanan Khan
College of Engineering
University of Connecticut
Storrs, CT USA
abdul.hanan@uconn.edu

Omer Khan
College of Engineering
University of Connecticut
Storrs, CT USA
khan@uconn.edu

***Abstract*—Graph neural networks and transformers are increasingly adopted for learning large-scale, irregular graph data, but their execution is often constrained by high computational inefficiencies on modern parallel throughput hardware. While unstructured pruning has emerged as an effective software optimization, its performance is limited by synchronization overhead and poor parallel hardware utilization. This paper explores architectural optimizations for the Sparse-Sparse-Dense (SSD) matrix operator introduced in MaxK, which is employed during model inference with unstructured model pruning. The inherent synchronization and scheduling bottlenecks fundamentally limit the performance scalability of this operator. To overcome these limitations, a novel asynchronous decoupled execution model is proposed that leverages independent thread scheduling to maximize latency hiding, minimize synchronization and data dependency stalls, and improve the effective hardware utilization. The proposed operator achieves ∼50% geometric mean speedup over the state-of-the-art MaxK operator on the NVIDIA H100 GPU and better exploits sparsity for performance.**

***Index Terms*—Graph neural networks and transformers; Graph processing; Model pruning; Throughput parallel processor; Independent thread scheduling.**

I. Introduction

Graph-based learning has become a foundational tool for learning over graph-structured data, with applications spanning social networks, molecular modeling, and recommendation systems. Graph Neural Networks (GNNs) [1]–[11] and Graph Transformers [12]–[16] have demonstrated significant success in capturing structural information and relational dependencies in graphs. GNNs typically rely on neighborhood aggregation or message-passing to learn node and graph representations, while Graph Transformers leverage attention mechanisms to model long-range dependencies and global interactions effectively. Despite their success, training and specifically, latency-critical inference for GNNs and Graph Transformers remains computationally expensive, given that real-world graphs are unstructured and sparse. The Sparse-Dense Matrix Multiplication (SpMM) parallel operator has been identified as a performance bottleneck in GNNs and many Graph Transformers [17]–[20]. In SpMM, the sparse graph adjacency matrix is multiplied by a dense feature or embedding matrix. However, the irregular memory access patterns and unstructured workload distribution arising from graph sparsity often lead to poor utilization of the underlying parallel hardware resources.

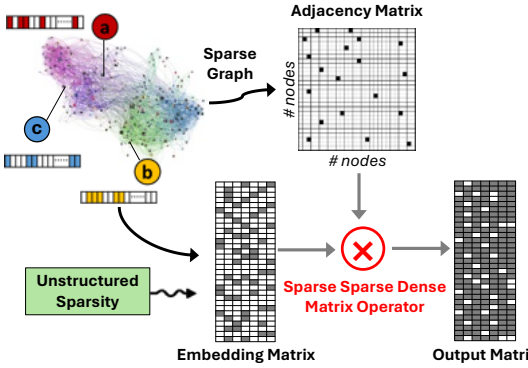


Fig. 1: Sparse-Sparse-Dense matrix multiplication operator resulting from unstructured model pruning.

To mitigate the computational challenge, recent works have explored hardware acceleration, sparsity-aware optimizations, kernel fusion, and pruning techniques. Among these, model pruning has shown promise in reducing both computation and memory footprint by selectively removing redundant weights while preserving model accuracy. Structured pruning [21], [22] is a model pruning technique that enforces sparsity at the level of entire rows or columns of the embedding matrix, whereas unstructured pruning [23]–[25] removes individual elements without strict regularity. Unstructured pruning achieves significantly higher sparsity compared to structured pruning because it allows finer-grain control over parameter removal, retaining only the most critical connections while eliminating redundant ones. This results in a significant reduction in the number of floating-point operations (FLOPs) compared to structured pruning. However, while unstructured pruning provides a theoretical boost, it introduces irregular memory access patterns and data dependencies that limit its practical performance gains.

The state-of-the-art (SOTA) implementation of unstructured model pruning is MaxK [23], which prunes by selecting the top-k values in each row of the embedding matrix. Pruning results in a row-balanced sparsified embedding matrix. As a result, it comes with its own challenges. Instead of the sparse adjacency matrix being multiplied with a dense matrix, it now gets multiplied with a sparsified embedding matrix, resulting in a Sparse-Sparse Matrix Multiplication (SpGEMM)

1530-2075/26/$31.00 ©2026 IEEE
DOI 10.1109/IPDPS65963.2026.00048

operator. However, intermediate partial products often yield a dense output matrix, which is referred to as Sparse-Sparse-Dense Matrix Multiplication (SSD) operator, as shown in Figure 1. To accommodate the dense nature of the output matrix, MaxK introduces a row-based matrix multiplication strategy that coalesces output updates. The graph adjacency matrix is partitioned using nonzero splitting to form work groups, each of which executes the SSD operator in two phases: accumulation and update. In the accumulation phase, partial products are computed and locally accumulated. These accumulated values are then written to the dense output matrix during the update phase.

A performance characterization using a modern parallel throughput GPU processor reveals three major inefficiencies in the MaxK implementation: (1) The coupling of accumulation and update phases among work groups necessitates synchronization between the two phases, resulting in stalling the update phase even for the work groups that have completed their accumulation phase; (2) The update phase is always carried out using atomic write operations, even for work groups that produce independent row outputs, leading to unnecessary serialization overhead due to data dependency stalls; and (3) At higher pruning rates, the data bandwidth required to perform partial products is much less than the bandwidth needed for the update phase. This leads to an imbalance in the hardware utilization between the two phases.

The SOTA does not fully exploit sparsity for performance. The objective is to devise an optimized dataflow implementation of the SSD operator that accelerates model pruning to overcome the aforementioned inefficiencies. Modern parallel GPU processors increasingly support asynchronous execution that allows threads to operate independently, even when residing within a vector core. This is enabled by independent thread scheduling and fine-grain synchronization primitives at the hardware level. The proposed approach unlocks performance for sparsity using two key strategies: (1) Decoupled execution enables independent work groups to execute their accumulation and update phases asynchronously. This allows the operator to utilize the hardware resources and maximize the use of available memory bandwidth. Moreover, each work group uses fine-grain synchronization primitives to reduce the communication stalls. (2) Work groups with the knowledge of local or shared output updates reduce the use of blocking atomic operations and avoid data dependency stalls. The proposed decoupled execution of the SSD operator achieves efficient hardware and memory bandwidth utilization to maximize the performance gains afforded by model sparsity.

The evaluation using an NVIDIA H100 GPU over a diverse set of real-world representative graphs demonstrates ∼50% geometric mean speedup over the SOTA implementation of the SSD operator. The evaluation highlights the importance of architecture-aware optimizations and shows that careful handling of sparse irregularities significantly enhances the real-world performance of sparsity-driven graph analytics.

Fig. 2: Normalized Execution time of SpMM and GEMM matrix operators in a single representative forward propagation layer of a multi-layer GNN model.

II. Background

Building on classical deep learning, a variety of GNN models have been developed for learning graph-structured data in static and dynamic graphs, such as Graph Convolutional Network (GCN) [1], Simple Graph Convolution (SGC) [2], Topology Adaptive Graph Convolutional Network (TAGCN) [3], Graph Isomorphism Network (GIN) [4], GraphSAGE [5], Graph Attention Network (GAT) [6], [7], Gated Graph Attention Network (GaAN) [8], ResGCN [9], EvolveGCN [10], CD-GCN [11], and others. More recently, Graph Transformers have emerged to capture long-range graph connections by leveraging attention mechanisms. These transformers can either build on top of GNN blocks such as Graph Trans [12], Grover [13], GraphiT [14], or get stacked with a GNN layer [15], or execute in parallel with a GNN block [16].

Graph-based learning models typically iterate with two key phases: combination and aggregation. In the combination phase, node features are transformed into hidden representations via neural network layers. In the aggregation phase, embeddings from neighboring nodes are consolidated to enhance node representations. For example, in a layer of a GCN model, during forward propagation (used in both inference and training), the input to layer i is the aggregated output of the previous layer $Agg_{i-1} \in \mathbb{R}^{n \times d'}$, where n is the number of nodes and d' is the embedding dimension. The combination phase applies a dense matrix multiplication (GEMM) to transform embeddings with the weight matrix: $Comb_i = Agg_{i-1} \cdot W_i$, where $W_i \in \mathbb{R}^{d' \times d}$ is the weight matrix of the current layer and d is the hidden dimension of the current layer. The non-linear activation function, such as ReLU or Sigmoid, is then applied to $Comb_i$. After activation, the aggregation phase performs a sparse matrix multiplication (SpMM) to transform the embeddings with the graph adjacency matrix $A \in \mathbb{R}^{n \times n}$ to propagate information across connected nodes. In certain models, such as GAT, a sampled GEMM operator (Sparse-Dense-Dense Matrix Multiplication, SDDMM) updates the adjacency matrix with attention coefficients. The resulting adjacency matrix is then used in the SpMM operator to transform the embeddings.

Figure 2 shows the execution time breakdown of each operator involved during the model inference in a representative GCN model using the Reddit, Ogbn-products, and Yelp graphs. The evaluation is done using the NVIDIA cuBLAS and

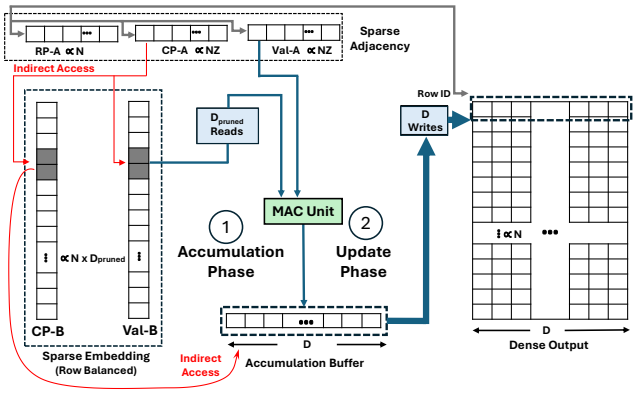


Fig. 3: Dataflow of the Sparse-Sparse-Dense (SSD) operator.

cuSPARSE libraries (version 12.3) executing on the NVIDIA H100 GPU [26]. Notably, the SpMM operator in the *Agg* phase accounts for >80% of the total execution time. This trend is consistent across a wide range of GNNs and many Graph Transformer models, where SpMM typically dominates the runtime during model inference. The SpMM operator is computationally expensive as it results in workload balancing and memory access challenges due to the power-law distribution of nonzeros in the graph adjacency matrix. The graph-proportional SpMM is acknowledged as a hard problem in the literature with solutions ranging from hardware accelerators to optimized GPU frameworks [27]–[33].

A. Sparsity for Performance

Model pruning has emerged as a software method to unlock sparsity for performance in GNNs. Sparse training frameworks are applied to either the weight matrix [21], [25], [34]–[38] or to the output of the combination phase through the non-linear activation functions, such as [23], [39], [40]. Structured pruning [21] [22] reduces the embedding and hidden dimensions by eliminating less influential weight parameters to remove redundant computations in a structured manner. The weight matrices are compressed into lower dimensions. This approach enables the natural propagation of sparsity through the matrix operators. The main drawback of structured weight pruning is its limitations in exploiting high sparsity rates. On the other hand, unstructured pruning [23] [24] [25] exploits irregular sparsity in the intermediate embeddings to unlock high sparsity rates (>90%) without model accuracy loss. By selectively removing individual elements within the embedding matrix, it produces highly sparse intermediate representations. Recent work has compared the pruning rate achieved with structured and unstructured pruning for different GNN models across multiple graphs at equivalent accuracy levels [21]. The work concludes that unstructured pruning consistently achieves higher pruning rates. Therefore, this paper focuses on unstructured pruning due to its superior sparsity potential.

MaxK [23] represents a state-of-the-art non-linearity universal approximator that exploits unstructured pruning in the intermediate embedding matrix without sacrificing accuracy.

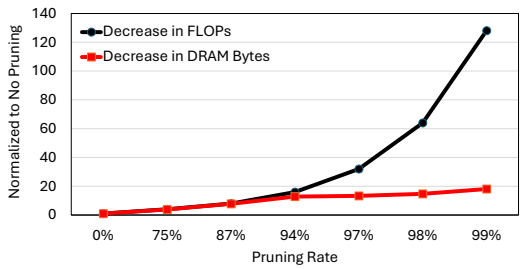


Fig. 4: Decrease in FLOPs and DRAM byte transactions achieved with the SSD matrix operator at increasing pruning rates normalized to the SpMM operator at 256 dimension size.

It employs a top-k selection as an activation function for each node embedding matrix in the GNN layers. The embedding matrix is pruned in a row-balanced manner by selecting the k elements from each row of the embedding matrix. This results in a sparse matrix where each row contains the same number of nonzero elements, though their locations are unstructured. MaxK adopts a Compressed Balanced Sparse Row (CBSR) format to store the nonzero value data and the column indices, which saves space compared to traditional compressed sparse formats (e.g., CSR). The nonlinearity approximator is applied to $Comb_i$ instead of ReLU. Hence, the combination phase retains the GEMM operator. However, after the activation function, the aggregation phase operates on two sparse matrices requiring a Sparse Matrix Matrix Multiplication (SpGEMM) operator instead of an SpMM. To leverage the CBSR format for coalesced global memory accesses, MaxK proposes a row-wise product variant of the SpGEMM operator, which outperforms the SpMM kernels of NVIDIA's cuSPARSE library and the GNNAdvisor's GPU implementation [31]. MaxK stores the output matrix in a row-major format instead of the CSR format. The irregular nature of nonzeros in both input matrices results in a dense accumulation, which updates the output at the unpruned dimension size. Therefore, the output is a dense matrix, resulting in a Sparse-Sparse-Dense (SSD) operator.

B. Sparse-Sparse-Dense (SSD) Matrix Operator Dataflow

Figure 3 illustrates the SSD matrix operator's dataflow, which performs row-wise matrix multiplication in two phases: *Accumulation* and *Update*. The sparse adjacency matrix is stored in the CSR format, where the column pointer and value array (*CP-A* and *Val-A*) are indexed by the row pointer array (*RP-A*). The sparse embedding matrix is represented using a column pointer and a value array (*CP-B* and *Val-B*). For each nonzero element in the adjacency matrix, the corresponding D_{pruned} nonzero values and column indices from the embedding matrix are accessed using the indices in *CP-A*. During the accumulation phase (1), a dot product is computed between each nonzero of the sparse adjacency matrix and the corresponding sparsified embedding nonzero. Pruning reduces the effective embedding dimension from the unpruned dimension D to D_{pruned}, thereby decreasing both memory accesses and multiply-add-accumulate (MAC) operations, resulting in the FLOPs for the SSD operator proportional

to D_{pruned}. To mitigate non-coalesced memory accesses, the dot products are accumulated into an unpruned-dimension-wide (D) accumulation buffer, where the locations of the D_{pruned} accumulated values are determined by the values in *CP-B*. The randomness of indices in *CP-B* necessitates a dense accumulation buffer despite the reduction in MAC operations achieved through pruning. Once the accumulation phase completes, the update phase transfers the intermediate results from the D-wide buffer to the dense output matrix stored in row-major order in global memory (2). Although pruning reduces computation and data movement during accumulation, the update phase always writes results at the full dimension size (D). Consequently, the overall memory requirement of the SSD operator exhibits diminished scaling at higher pruning rates. As shown in Figure 4 for the representative Ogbn-product graph, the DRAM bytes transferred decrease by $20\times$ at a 99% pruning rate, whereas the FLOPs reduce by $128\times$. This imbalance motivates the design of a parallel, throughput-oriented SSD operator that efficiently exploits sparsity for performance.

III. Parallel SSD Operator Limitations

MaxK proposes a parallel implementation for the SSD matrix operator using GPU throughput-oriented vector hardware. For illustration, NVIDIA GPU terminology is used in this paper. However, the parallelization methods are generally applicable to other vector parallel processors. A *warp* is a group of 32 consecutive threads that execute in a Single Instruction Multiple Thread (SIMT) manner. Warps are organized into *thread blocks*, with a maximum of 1024 threads (32 warps) per block on modern NVIDIA GPUs, such as the Hopper H100. The GPU hardware is composed of *Streaming Multiprocessors* (SMs), each of which contains compute cores, a large shared register file, and low-latency on-chip memory that is configurable as an L1 cache or scratchpad (termed as shared memory). Thread blocks are scheduled on SMs: multiple blocks may reside on the same SM if sufficient resources (registers and shared memory) are available. Each thread block executes entirely within a single SM before another thread block is scheduled for processing.

Building on this GPU execution model, MaxK implements a static nonzero splitting scheme for processing the adjacency matrix in parallel. It divides the nonzeros into Edge Groups (EGs), where each EG contains up to a fixed number of nonzeros from an adjacency matrix row. These EGs are generated through offline pre-processing. A thread block is assigned multiple EGs, which is equal to the total number of warps within the block. For each EG, MaxK allocates an unpruned dimension-wide buffer in the low-latency shared memory. Once the EGs are mapped to their respective warps, execution proceeds in the accumulation and update phases. The accumulation phase operates over the pruned dimensions and requires compute bandwidth proportional to the pruning rate. In contrast, the update phase writes the output matrix at full unpruned dimension size and requires significantly higher memory bandwidth.

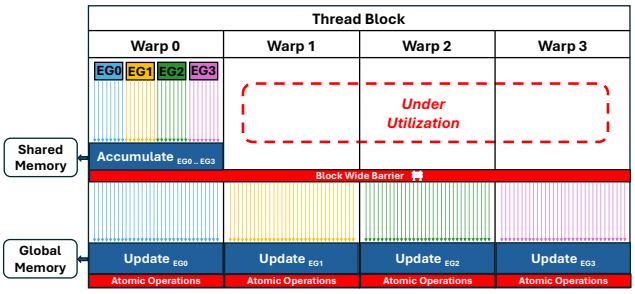


Fig. 5: A thread block level two-phase dataflow for MaxK's SSD operator at $D_{pruned} = 8$.

When the pruned dimension size is greater than or equal to the warp size, each EG is assigned to a single warp for accumulation. Otherwise, at higher pruning rates, the EGs within a block are accumulated by $\frac{D_{pruned}}{\text{Warp Size}} \times \text{warps\_per\_block}$ warps. This ensures that consecutive lanes within a warp perform coalesced accesses to the EG's metadata stored in global memory. The accumulated values for each EG are stored in a shared memory buffer at the full dimension width. This is shown in Figure 5, which illustrates a block-level dataflow of the SSD operator proposed by MaxK at a pruned dimension size of 8. Assuming that each block contains 4 warps, the block collectively executes 4 EGs. The EGs are mapped to the lanes of the first warp in the block for accumulation (since $\frac{D_{pruned}}{\text{Warp Size}} \times \text{warps\_per\_block} = \frac{8}{32} \times 4 = 1$). Warp 0 accumulates the result of the four EGs into their respective shared memory buffers. While this design improves memory coalescing for active warps, it leads to under-utilization of the vector compute units at higher pruning rates. As a result, warps 1 to 3 are idle during the accumulation phase. Following accumulation, a block-wide synchronization barrier is required to prevent data races when transitioning from accumulating dot products in shared memory buffers to the subsequent reads in the update phase. In the example, warps 1 to 3 stall until warp 0 completes its accumulation phase. After the block-wide barrier, the update phase commences, and the accumulated data in the buffers is written to the output rows for each EG. Since multiple EGs may update a shared row in the output, each warp performs atomic write operations to the global memory where the output matrix is stored.

MaxK's implementation of the SSD operator suffers from the following parallel performance limitations:

- **Underutilized Hardware Resources**: As the pruning rate increases, only a subset of the SIMT lanes remain active during accumulation, leading to diminished compute utilization as the warp's execution units become increasingly unused.
- **Block-level Synchronization**: All warps in a block are synchronized before moving on to the update phase. This coarse-grained inter-warp coupling of EGs results in unnecessary synchronization stalls, resulting in a decrease in parallel throughput for the update phase.
- **Atomic Operations**: The updates are always performed using atomic operations, regardless of whether an EG is

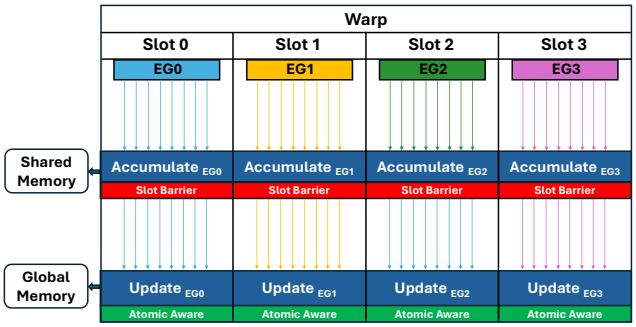


Fig. 6: A warp level two-phase ideal decoupled dataflow of the SSD operator at $D_{pruned} = 8$.

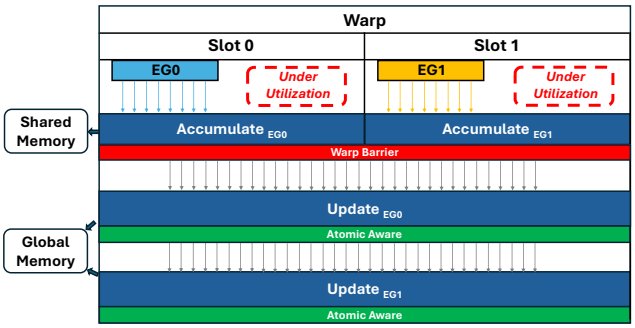


Fig. 7: A warp level two-phase proposed decoupled dataflow of the SSD operator at $D_{pruned} = 8$.

writing its output to a shared row or not. This static use of atomic operations introduces data dependency stalls that limit the parallel throughput for the update phase.

These limitations prevent the MaxK dataflow from exploiting the vector hardware and the memory bandwidth, as stalls and resource under-utilization degrade overall performance. This raises the question of whether the SSD dataflow can be redesigned to minimize these stalls and maximize the effective compute and memory bandwidth utilization.

IV. SSD Operator with Decoupled Execution

Modern parallel throughput GPUs perform independent thread scheduling that allows threads within a warp to execute and progress independently using the SIMT execution model. This decouples threads within a warp, which enables asynchronous thread execution for improved parallelism. However, explicit inter-warp synchronization is required when a collection of threads within a warp align at a specific point in code execution. GPU vendors expose these capabilities via APIs, such as Cooperative Groups [41] that allow developers to manage and synchronize groups of threads for asynchronous execution, resulting in flexibility from grid-wide to synchronizing a few lanes within a warp. To overcome the limitations of the tightly coupled accumulation and update phases for the SSD operator, we propose a decoupled dataflow, where each EG independently operates in its assigned thread group without incurring unnecessary synchronization and data dependency stalls.

Each EG is allocated a single unpruned dimension-sized shared memory buffer in which the assigned threads, equivalent to the pruned dimension size, perform parallel accumulations. Each warp is divided into groups of SIMT threads, termed as *slots*. The number of slots within a warp is determined by the ratio $\frac{\text{Warp Size}}{D_{pruned}}$. The EG assigned to a slot performs its accumulation phase and updates its allocated shared memory buffer. The threads within the slot synchronize to ensure the accumulations for the EG are complete before continuing with the update phase. The update phase for an EG commences independently of other EGs in the warp and performs the writes for its output matrix row. Additional metadata that tracks whether the output row is updated by multiple EGs is added during the pre-processing stage. This allows the update phase to avoid the expensive atomic write operations for EGs that do not update rows shared by other EGs, and reduce data dependency stalls.

Figure 6 illustrates the two-phase decoupled dataflow of the proposed SSD operator with atomic-aware updates. Note that the dataflow is illustrated at the warp level rather than the block level, since each warp within a thread block executes independently without inter-warp communication dependencies. Since only 8 lanes perform accumulation at a pruned dimension size of 8, each warp is divided into 4 slots. Each slot executes an EG for accumulation and stores the accumulated data in a dedicated shared memory buffer. Following accumulation, each slot performs a slot-level barrier and commences the atomic-aware update phase independent of the other slots. This decoupled datapath appears ideal, but it has two main limitations.

The shared memory in a modern NVIDIA GPU is allocated per thread block that executes on an SM. Each SM has a dedicated level-1 (L1) cache that is reconfigurable as a scratchpad to support shared memory. At higher pruning rates, the number of required slots per block increases — for instance, a single warp in a block may need 16 unpruned dimension-wide shared memory buffers when the pruned dimension size is 2, compared to just one buffer at a pruned dimension size of 32. However, since shared memory and the L1 cache share the same on-chip storage, allocating more shared memory reduces the available L1 cache capacity for the SM. As accesses to the sparsified embedding matrix are highly irregular, maintaining sufficient L1 cache capacity is essential for capturing the available temporal and spatial locality. Consequently, it is not always feasible to allocate shared memory buffers equal to the number of slots, since it reduces the effective L1 cache capacity and degrades performance. In this paper, the number of shared memory buffers per block is determined empirically for a given full dimension size. The associated tradeoff is further evaluated in Section VI-C. This constraint translates to a limited number of slots per warp, which in turn limits the number of EGs that can be processed concurrently within a warp.

Modern GPUs rely on coalesced data accesses to global memory for high memory throughput. When each slot executing a different EG in a warp performs independent up-

dates, memory accesses become scattered across different cache lines, resulting in increased sector requests and reduced memory coalescing efficiency. This results in higher memory transaction overheads and reduced effective bandwidth utilization. To overcome this limitation, all the slots in a warp perform a warp-wide synchronization to ensure all slots have completed their accumulation phase before performing a "fused" update phase at the warp granularity. All warp-mapped EGs perform their update phase sequentially, while fully utilizing the available memory bandwidth in a coalesced manner.

Figure 7 illustrates the proposed decoupled SSD operator, where the number of shared memory buffers per warp is constrained to two. Consequently, the number of slots is configured to map up to two EGs per warp. This leads to some under-utilization of threads for the accumulation phase, where each slot uses $D_{pruned} = 8$ lanes for accumulation. Even though this limitation does not fully utilize all SIMT threads in a warp, it still improves the accumulation bandwidth by more than 50% over MaxK. Following a warp-wide synchronization barrier, all lanes within a warp sequentially perform coalesced updates for the EGs assigned to that warp.

A. Smart Edge Groups

MaxK preprocesses the graph adjacency matrix by splitting the nonzeros into parallel edge groups (EGs). The metadata for each EG is stored in global memory and includes the row ID of the sparse matrix (row_id), the number of nonzeros for accumulation by the EG in that row (nnz), and the starting column ID of the nonzeros (nz_id). This information is sufficient for the SSD operator to perform its accumulation and update phases. We extend the metadata for each EG by introducing an additional field, $update\_mode$. The proposed smart EG populates $update\_mode$ with the knowledge of whether the update phase performs atomic updates or regular writes to the output matrix. The preprocessing phase of MaxK is extended to incorporate smart EG using the high-level flow described next. $Warp\_Size$ indicates the number of SIMT lanes per warp, $D_{unpruned}$ indicates the unpruned number of dimensions, and D_{pruned} indicates the pruned number of dimensions for the SSD operator.

The number of shared memory buffers per warp is determined using the minimum of $Warp\_Size/D_{pruned}$ and the maximum shared memory buffers allowed. This maximum limit is set by the user and is determined empirically for a given $D_{unpruned}$. Then, the number of slots per warp, $Slots\_Warp$ is set to the number of shared memory buffers per warp. The number of lanes allocated to each slot is computed using $Warp\_Size/Slots\_Warp$ and captured as the $Slot\_Lanes$ variable. Since each EG is mapped to a single slot in a warp, increasing slots due to increased shared memory buffers results in fewer warps getting spawned. In contrast, MaxK spawns warps equivalent to the total number of EGs. Each EG determines if it performs updates to a row shared by other EGs by comparing its row_id with the row IDs of its two adjacent EGs. If the row IDs match, then this EG is assigned as a shared row and must perform its update phase using atomic write operations. This decision is captured by setting $update\_mode$ to 1. If an EG is determined to update a row that is not shared, it performs its update phase using non-blocking write operations. This decision is captured by setting $update\_mode$ to 0.

In summary, the proposed SSD operator introduces smart EGs that enable atomic-aware updates. This reduces unnecessary data dependency stalls. Moreover, the proposed decoupled dataflow delivers improved throughput by reducing synchronization stalls and increasing hardware resource utilization.

Graph	Rows	Nonzeros	Avg. Deg.	Max. Deg.	Shared EGs(%)
Proteins_full	43,466	162,088	3	25	0
Twitter-Par	580,768	1,435,116	2	12	0
DD	334,925	1,686,092	5	19	0
OVCAR-8H	1,889,542	3,946,402	2	5	0
Yeast	1,710,902	3,636,546	2	6	0
SW-620H	1,888,584	3,944,206	2	5	0
pubmed	19,717	99,203	5	171	1
Flickr	89,250	899,756	10	5,425	4
Ogbn-arxiv	169,343	1,166,243	6	436	1
com-amazon	334,863	1,851,744	5	549	1
collab	235,868	2,358,104	9	671	4
amazon0505	410,236	4,878,874	12	2,760	2
amazon0601	402,439	5,478,357	13	2,547	7
am	881,680	5,668,682	6	154,828	3
youtube	1,138,499	5,980,886	5	28,754	3
citation	2,927,963	30,387,995	10	1,738	3
ppi	56,944	818,716	14	429	10
ddi	4,267	2,135,822	500	2,234	98
artist	50,515	1,638,396	32	1,469	29
Yelp	716,847	13,954,819	19	4,886	16
ppa	576,289	42,463,862	73	3,241	63
Ogbn-product	2,449,029	123,718,152	51	17,481	49
Ogbn-proteins	132,534	79,122,504	597	7,750	97
Reddit	232,965	114,615,892	492	21,657	99

TABLE I: Graph datasets used for evaluation.

V. Methodology

The evaluation is done using an NVIDIA H100 GPU in the GH200 Grace Hopper Superchip [26] with 96 GB HBM3 memory, 4 TB/sec memory bandwidth, and 144 streaming multiprocessors (SMs) that implement $9k$ double-clocked CUDA cores operating at 2 GHz clock frequency.

Table I summarizes the graph datasets used for evaluation. These graphs have been widely adopted in prior works on GNN pruning methods [21], [23], [42]. The selection includes 24 graphs evaluated by MaxK, providing a representative set for performance comparison. The evaluated graphs are categorized into structured and unstructured graphs. The structured graphs (first six in the table) exhibit a highly uniform distribution of nonzeros per row. In contrast, the remaining graphs are irregular in node and edge connectivity. Across all graphs, there is significant variation in the number of nodes, edges, average degree, and adjacency matrix size. Notably, the last eight graphs are both unstructured and relatively dense, with a higher fraction of EGs belonging to shared rows as shown in the Shared EGs (%) column.

The SSD operator from MaxK is used as the baseline since it has been shown to outperform the standard SpMM operator implemented in cuSPARSE as well as the operator used in GNNAdvisor [31] over unpruned embedding matrices. MaxK

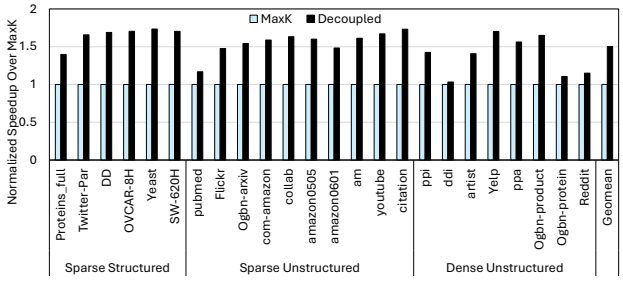


Fig. 8: Performance speedup of Decoupled over MaxK.

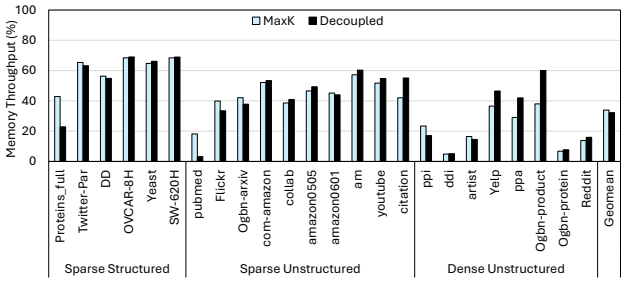


Fig. 9: Memory Throughput of Decoupled and MaxK.

generates the Edge Groups in a pre-processing phase. The EGs are extended with the Smart EG metadata using a data-parallel GPU thread mapping strategy. This pre-processing kernel takes $<$1% of the overall SSD operator execution time. Therefore, it is excluded from the evaluation. However, the performance of the MaxK SSD operator against each optimization for the proposed decoupled SSD operator is quantified.

The first variant, *w/o Atomic-Aware (AA)* follows the proposed decoupled operator but performs atomic updates unconditionally, regardless of whether an EG belongs to a shared row or not. The second variant, *w/o Slots* follows the proposed decoupled operator but restricts each warp to map a single EG by allocating only one shared memory buffer per warp. The Third variant *w/o Slots & AA* follows the proposed decoupled operator but restricts each warp to a single EG and also performs only atomic updates. Finally, the *Decoupled* operator integrates all proposed optimizations, including the decoupled dataflow, multiple slots (shared memory buffers) per warp, and atomic-aware updates (Smart Edge Groups). In the *Decoupled* and *w/o AA* variants, the maximum number of shared memory buffers per warp is empirically determined and fixed to two. For consistency with MaxK, the default full dimension size of the embedding matrix is set to 256, and then pruned from 75% to 99%, resulting in the pruned dimensions ranging from 64 to 2, respectively.

The Decoupled SSD operator aims to mitigate execution stalls and enhance parallel GPU throughput. To quantify the performance improvements, several metrics are measured using NVIDIA's Nsight profiling tool and CUDA version 12.1. These metrics include kernel execution time, memory throughput, and instruction issue stalls. Memory throughput is defined as the ratio of the achieved memory bandwidth to the peak memory bandwidth of the memory system. Stalls quantify the proportion of cycles during which warps are precluded

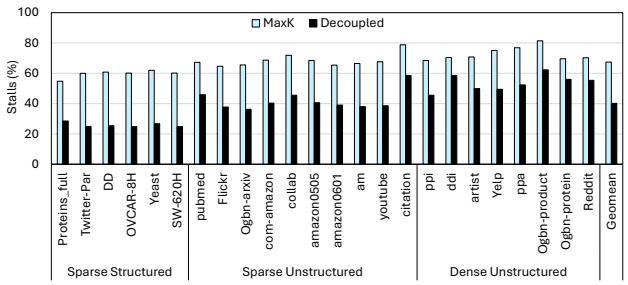


Fig. 10: Stalls of Decoupled and MaxK.

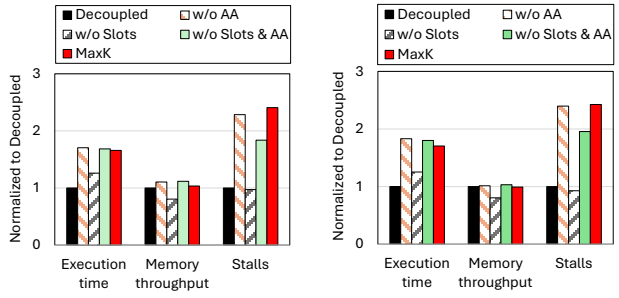
from issuing instructions. The cause of stalls may originate from synchronization barriers, scoreboard dependencies (long and short), branch resolution, pipeline drains, and various structural or resource contentions within the SM. Additionally, the L1 cache hit rate is reported to quantify the performance scaling behavior with respect to the number of shared memory buffers.

Since this work focuses on optimizing the SSD operator itself, an end-to-end evaluation of the model training or inference pipeline is not included. MaxK already provides such an evaluation, and accelerating the SSD operator directly translates to reduced inference latency when unstructured pruning is deployed.

VI. Evaluation

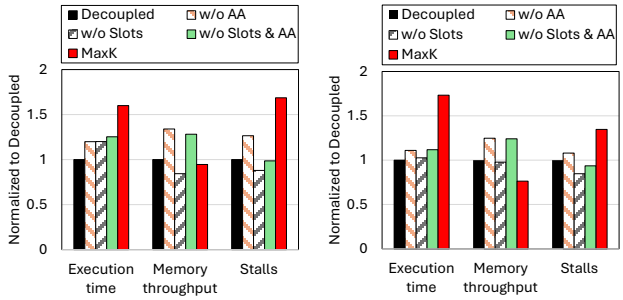
Figure 8 shows the kernel execution time speedup of the proposed Decoupled SSD operator relative to the MaxK baseline. Figure 9 shows the percentage of the achieved peak memory throughput of the evaluated GPU, while Figure 10 presents the percentage of execution cycles stalled at the warp granularity. Each bar represents the geometric mean across pruning rates ranging from 75% to 99% for all evaluated graphs from Table I.

The first category is the structured graphs (Proteins_full through SW-620H). These graphs have a low average degree ($\sim$2), resulting in EGs with very few nonzeros that update an output row after the accumulation phase. This is shown in the shared EGs column in Table I. Consequently, the accumulation phase is computationally less expensive compared to the update phase that operates at the full dimension size. MaxK performs the updates using atomic operations, requiring the output data to be read before writing it back to memory, resulting in data dependency stalls. The unnecessary reads also consume the constrained bandwidth of the memory system. The Decoupled SSD operator uses the knowledge of the output row updates confined to a single EG and performs non-blocking writes to memory, thereby avoiding data dependency stalls during the update phase. Moreover, the structured nature of nonzeros in the graph adjacency matrix accesses the sparse embedding matrix in contiguous memory locations, resulting in coalesced memory accesses that exploit the available memory bandwidth saved from not performing the memory reads during the update phase. The computation (accumulation) bandwidth enabled by the additional slots per warp further makes use of the available memory bandwidth for



(a) Twitter-Par Graph (b) SW-620H Graph

Fig. 11: Structured sparse graphs.



(a) amazon0501 graph (b) citation graph

Fig. 12: Unstructured sparse graphs.

performance. The performance gain of 64.7% for the sparse structured graphs is primarily attributed to the 2.3× reduction in stalls. Furthermore, the saved memory bandwidth from avoiding reads to the output data during the update phase is more efficiently utilized for the additional slots to perform faster accumulations. The only exception is the Proteins_full graph, where the memory bandwidth does not improve over MaxK. It is a small graph and does not need the saved memory bandwidth to unlock performance from the additional slots and local row updates. This trend is also observed in other small graphs, such as pubmed, Flikr, and ppi.

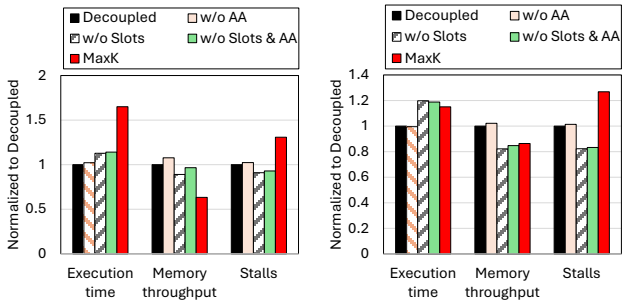
The second category is the unstructured sparse graphs (pubmed through citation). They are also sparse, but exhibit high variations in their number and the connections of the nonzeros in the graph adjacency matrix. The power law behavior of these graphs results in some EGs (1 to 4% in Table I) with non-zero values from different rows, while other EGs contain a few nonzeros. Consequently, most EGs perform local row updates and take advantage of the efficient writes in the Decoupled SSD operator. This reduces the data dependency stalls and frees up the memory bandwidth, as discussed earlier. A unique behavior in the sparse unstructured graphs is their non-contiguous memory accesses to the sparse embedding matrix. The indirect and uncoalesced accesses result in memory stalls during the accumulation phase. The Decoupled SSD operator unlocks the memory bandwidth using the additional slots and enables latency hiding of the stalls observed during the accumulation phase. The performance gain of 55% for the sparse unstructured graphs is attributed to the 1.6× reduction in stalls, which is relatively smaller compared to the structured graphs. This is due to fewer opportunities for hiding the stalls in both the accumulation and update phases. The unstructured sparse graphs observe better memory bandwidth utilization compared to MaxK due to more opportunities for the Decoupled operator to hide the memory latencies in the accumulation phase. This is evident in the citation graph, which observes a 31% improvement in memory throughput and 34.6% reduction in stalls compared to MaxK.

The third category is the unstructured dense graphs (ppi through Reddit) that have a high number of nonzeros in the graph adjacency matrix. This results in most EGs with shared rows (as seen in Table I), diminishing the opportunity for the update phase to avoid data-dependent updates. The average degree in the table shows variations from less dense (such as Yelp and obgn-product) to highly dense (such as Ogbn-proteins and Reddit) graphs. Graphs with low average degree show significant improvement in memory throughput since ∼50% of the EGs process local row updates, and the additional slots exploit the saved memory bandwidth to process multiple EGs in parallel. On the other hand, graphs with high average degree incur more indirect memory accesses to the embedding matrix in the accumulation phase and perform most updates to rows that are shared across EGs. Therefore, these graphs show the least reduction in stalls and are unable to exploit the available memory bandwidth using both MaxK and the Decoupled SSD operator. Overall, the dense unstructured graphs show a 37.9% performance gain over MaxK.

Across all evaluated graphs, the Decoupled SSD operator gives a geometric mean performance boost of 50.4% over MaxK. To evaluate the contribution of each proposed optimization, the performance of w/o AA (Atomic-Aware), w/o Slots, and w/o Slots & AA variants is analyzed and compared against the proposed Decoupled operator, which integrates all optimizations.

A. Evaluation of Decoupled SSD Operator Variants

Two representative graphs, Twitter-Par and SW-620H are picked from the structured sparse category. The kernel execution time, memory throughput, and stalls for each variant and MaxK are compared and normalized against the Decoupled operator, as shown in Figure 11. In the w/o AA variant, the additional slots do not yield any performance improvement over MaxK as the atomic updates saturate the memory bandwidth with unnecessary loads. These data-dependent updates also prevent the reduction in stalls. In the w/o Slots variant, atomic-aware updates reduce the data dependency stalls in the update phase. The removal of unnecessary loads also reduces stress on the memory system, which lowers the memory bandwidth utilization. On the other hand, the additional slots in the Decoupled operator take advantage of the memory bandwidth from non-blocking updates and achieve higher performance due to increased compute bandwidth for accumulations. Finally, both MaxK and the w/o Slots & AA variant fail to mitigate the data-dependency stalls due to blocking updates



(a) Ogbn-product graph (b) Reddit graph

Fig. 13: Unstructured dense graphs.

and hence underperform compared to the Decoupled SSD operator.

Two representative graphs, amazon0501 and citation are picked from the unstructured sparse category, as shown in Figure 12. The Decoupled operator achieves the best balance between stall reduction and the utilization of the memory bandwidth, resulting in the highest performance gain over MaxK. All three variants (w/o Slots & AA, w/o Slots, w/o AA) exhibit comparable speedups over MaxK. However, the reason for performance improvement is different in each variant. In w/o Slots & AA, just decoupling significantly reduces the communication stalls compared to MaxK. The graphs in this category have an unstructured pattern of nonzeros, which makes the accumulation phase of each warp highly irregular. This makes the block-wide barrier expensive, resulting in communication stalls observed in MaxK. Decoupling the warps significantly reduces these stalls, and the performance boost is observed in w/o Slots & AA. Adding slots to this variant (w/o AA) increases the memory throughput by performing more parallel loads during the accumulation phase. However, these loads are irregular and uncoalesced, leading to higher stalls. Hence, no significant performance boost is observed by this variant over the w/o Slots & AA. Similarly, w/o Slots achieves the biggest reduction in stalls as decoupling reduces the communication stalls and atomic-aware updates decrease the data dependency stalls. However, the memory bandwidth saved from the update phase is not utilized, limiting the accumulation bandwidth. Finally, the Decoupled operator observes an increase in stalls compared to w/o Slots due to its increased irregular parallel loads. However, the increased memory bandwidth utilization helps the Decoupled operator hide the latency of these stalls, translating to the lowest execution time among all variants and MaxK.

Two representative graphs, Ogbn-product and Reddit are picked from the unstructured dense category, as shown in Figure 13. For these graphs, the update phase mostly employs atomic operations. Hence, variants with atomic-aware updates (w/o Slots & AA and w/o Slots) do not aid performance gains as significantly as variants with increased accumulation bandwidth (w/o AA). This effect is more pronounced for Reddit, which contains 99% shared EGs. The increased accumulation bandwidth in the w/o AA and the Decoupled SSD operator

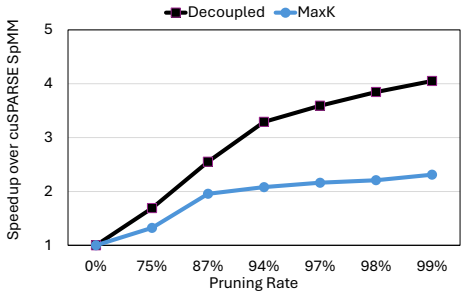
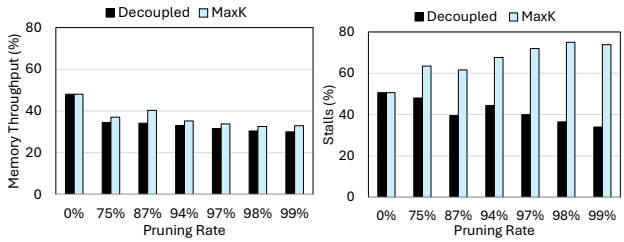


Fig. 14: Performance scaling with unstructured pruning.



(a) (b)

Fig. 15: **(a)** Memory Throughput and **(b)** Stalls of Decoupled and MaxK SSD operators at increasing pruning rates.

improves memory throughput over MaxK, translating to the highest performance gains.

B. Evaluation at Different Pruning Rate

Figure 14 illustrates the performance scaling of MaxK and the proposed Decoupled SSD operator. The geometric mean kernel execution times across all evaluated graphs are normalized to the NVIDIA cuSPARSE library's SpMM operator at 0% Pruning Rate (dimension size of 256). As discussed earlier and quantified in Figure 4, the SSD operator achieves significant reductions in FLOPs as the pruning rate increases. However, MaxK's increased under-utilization of the SIMT lanes during the accumulation phase does not allow it to scale with increasing sparsity. For example, at 94% pruning, MaxK utilizes only 50% of the lanes in a block, and this utilization decreases to 6.25% at 99% pruning rate. These underutilized lanes also worsen the communication stalls between the two phases as an increasing number of lanes remain idle during the accumulation phase. The atomic operations for the full-dimension updates in MaxK significantly contribute to data dependency stalls, which further diminish its performance scaling. The performance scaling trend in the Decoupled SSD operator is best represented at 16 pruned dimension size (94% pruning rate), where it exploits maximum potential for SIMT lane utilization with two slots. This trend tapers off at lower pruning rates since the constrained slots limit the performance gain. In contrast, MaxK is limited to a single slot execution, which saturates its performance scaling at 32 pruned dimensions.

To gain further insights into the performance scaling trends, the Figures 15a and 15b show the percentage of memory bandwidth and stalls for MaxK and Decoupled SSD oper-

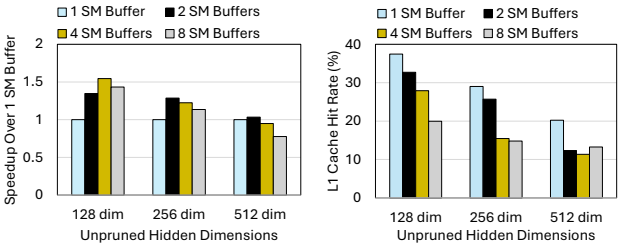
ators at increasing pruning rates. The reduction in memory throughput for both Decoupled and MaxK is due to the reduction in the embedding matrix size from unstructured pruning. However, this reduction is limited since the output matrix is still updated at the full, unpruned dimension size. As a result, a significant portion of the memory traffic remains constant despite the increasing pruning rate. MaxK achieves slightly higher memory throughout compared to Decoupled because it performs the update phase using read-modify-write operations for all EGs. On the other hand, Decoupled uses the atomic-aware updates to only perform blocking updates when needed. This eliminates unnecessary memory traffic. However, Decoupled increases memory traffic by deploying additional EGs to improve parallel throughput for the SSD operator.

The stalls in MaxK increase with increasing pruning rate as fewer warps remain active during the accumulation phase, and others wait on the thread-block-wide barrier synchronization. Moreover, as the pruning rate increases, the update phase dominates among the two phases, and the data-dependent stalls in MaxK become a significant contributor to the increasing trend in stalls. On the other hand, Decoupled replaces the block-wide barrier with warp-level synchronizations to reduce the communication stalls. The update phase is also performed using atomic-aware non-blocking writes to reduce data-dependency stalls. Moreover, as the pruning rate increases, the data-dependency stalls decrease since fewer nonzeros from the sparsified embedding matrix are processed during the accumulation phase. As a result, Decoupled observes significant reductions in stalls compared to MaxK.

C. Number of Slots at Different Hidden Dimension Sizes

To quantify the impact of the number of slots per warp, which also determines the number of shared memory buffers, a sensitivity study is conducted by varying the number of shared memory buffers allocated per warp. In addition, the sensitivity to the dimension size of the embedding matrix is evaluated since it impacts the size of each shared memory buffer. Figures 16a and 16b report the speedup of the Decoupled SSD operator using a single shared memory buffer per warp and the corresponding L1 cache hit rate with an increasing number of shared memory buffers. This evaluation is performed across different unpruned embedding dimension sizes.

As the number of shared memory buffers per warp increases, the L1 cache hit rate consistently decreases due to the reduced cache capacity available for the input and output matrices. Consequently, allocating the maximum number of shared memory buffers per warp is not always optimal. The evaluation further shows that when the unpruned embedding matrix has a small dimension size, allocating multiple shared memory buffers is beneficial. A smaller buffer size places reduced stress on the L1 cache while enabling higher concurrency. In contrast, for larger dimension sizes, additional buffers significantly diminish the effective L1 cache capacity, resulting in increased cache misses. For instance, four shared memory buffers achieve optimal performance at a dimension size of 128, whereas two buffers perform best at the dimension sizes of 256 and 512, respectively.



(a) (b)

Fig. 16: Sensitivity study for an increasing number of shared memory buffers per warp at different full dimension sizes for the proposed Decoupled SSD operator.

VII. Conclusion

Recent efforts have proposed to accelerate the training and latency-critical inference of GNNs and Graph Transformers with unstructured pruning. However, the state-of-the-art matrix operator resulting from unstructured pruning is unable to achieve performance for sparsity. This paper identifies key parallel execution inefficiencies of the sparse-sparse-dense (SSD) operator, including expensive atomic operations for output updates, communication stalls, and suboptimal hardware utilization. To address these challenges, a parallel throughput hardware-efficient implementation of the SSD operator is proposed that decouples the accumulation and update phases of execution, performs coalesced atomic-aware updates, and better utilizes the parallel hardware resources. The Decoupled SSD operator achieves $\sim$50% performance speedups over the MaxK baseline by optimizing the memory and compute throughput in parallel GPU processors.

VIII. Acknowledgement

This material is based upon work supported by the U.S. National Science Foundation under Grant No. 2429516. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of the National Science Foundation.

References

[1] T. Kipf and M. Welling, "Semi-supervised classification with graph convolutional networks," *ArXiv*, vol. abs/1609.02907, 2017.

[2] F. Wu, T. Zhang, A. H. S. Jr., C. Fifty, T. Yu, and K. Q. Weinberger, "Simplifying graph convolutional networks," *CoRR*, vol. abs/1902.07153, 2019. [Online]. Available: http://arxiv.org/abs/1902.07153

[3] J. Du, S. Zhang, G. Wu, J. M. F. Moura, and S. Kar, "Topology adaptive graph convolutional networks," *CoRR*, vol. abs/1710.10370, 2017. [Online]. Available: http://arxiv.org/abs/1710.10370

[4] K. Xu, W. Hu, J. Leskovec, and S. Jegelka, "How powerful are graph neural networks?" in *International Conference on Learning Representations*, 2019. [Online]. Available: https://openreview.net/forum?id=ryGs6iA5Km

[5] W. L. Hamilton, Z. Ying, and J. Leskovec, "Inductive representation learning on large graphs," in *NIPS*, 2017.

[6] P. Velickovic, G. Cucurull, A. Casanova, A. Romero, P. Lio', and Y. Bengio, "Graph attention networks," *ArXiv*, vol. abs/1710.10903, 2017. [Online]. Available: https://api.semanticscholar.org/CorpusID:3292002

[7] S. Brody, U. Alon, and E. Yahav, "How attentive are graph attention networks?" *CoRR*, vol. abs/2105.14491, 2021. [Online]. Available: https://arxiv.org/abs/2105.14491

[8] J. Zhang, X. Shi, J. Xie, H. Ma, I. King, and D. Yeung, "Gaan: Gated attention networks for learning on large and spatiotemporal graphs," *CoRR*, vol. abs/1803.07294, 2018. [Online]. Available: http://arxiv.org/abs/1803.07294

[9] Y. Pei, T. Huang, W. van Ipenburg, and M. Pechenizkiy, "Resgcn: attention-based deep residual modeling for anomaly detection on attributed networks," *Machine Learning*, vol. 111, pp. 519 – 541, 2020.

[10] A. Pareja, G. Domeniconi, J. Chen, T. Ma, T. Suzumura, H. Kanezashi, T. Kaler, and C. E. Leisersen, "Evolvegcn: Evolving graph convolutional networks for dynamic graphs," *ArXiv*, vol. abs/1902.10191, 2019.

[11] F. Manessi, A. Rozza, and M. Manzo, "Dynamic graph convolutional networks," *Pattern Recognition*, vol. 97, p. 107000, 2020. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0031320319303036

[12] P. Jain, Z. Wu, M. Wright, A. Mirhoseini, J. E. Gonzalez, and I. Stoica, "Representing long-range context for graph neural networks with global attention," in *Advances in Neural Information Processing Systems (NeurIPS)*, vol. 34, 2021. [Online]. Available: https://arxiv.org/abs/2106.06428

[13] Y. Rong, Y. Bian, T. Xu, W. Xie, Y. Wei, W. Huang, and J. Huang, "Self-supervised graph transformer on large-scale molecular data," *arXiv preprint arXiv:2007.02835*, 2020. [Online]. Available: https://arxiv.org/abs/2007.02835

[14] G. Mialon, D. Chen, M. Selosse, and J. Mairal, "Graphit: Encoding graph structure in transformers," *arXiv preprint arXiv:2106.05667*, 2021. [Online]. Available: https://arxiv.org/abs/2106.05667

[15] K. Lin, L. Wang, and Z. Liu, "Mesh graphormer," *arXiv preprint arXiv:2104.00272*, 2021. [Online]. Available: https://arxiv.org/abs/2104.00272

[16] J. Zhang and L. Meng, "Gresnet: Graph residual network for reviving deep gnns from suspended animation," *arXiv preprint arXiv:1*.

[17] A. R. Jamasb, R. Viñas, E. J. Ma, C. Harris, K. Huang, D. Hall, P. Lió, and T. L. Blundell, "Graphein - a python library for geometric deep learning and network analysis on biomolecular structures and interaction networks," in *Proceedings of the 36th International Conference on Neural Information Processing Systems*, ser. NIPS '22. Red Hook, NY, USA: Curran Associates Inc., 2022.

[18] J. Liu, Z. Cai, Z. Chen, and M. Wang, "Df-gnn: Dynamic fusion framework for attention graph neural networks on gpus," 2024. [Online]. Available: https://arxiv.org/abs/2411.16127

[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 *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, ser. SC '20. IEEE Press, 2020.

[20] M. Shan, D. Gurevin, J. Nye, C. Ding, and O. Khan, "Mergepath-spmm: Parallel sparse matrix-matrix algorithm for graph neural network acceleration," *2023 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS 2023*, 2023.

[21] D. Gurevin, M. Shan, S. Huang, M. Hasan, C. Ding, and O. Khan, "Prunegnn: An optimized algorithm-hardware framework for graph neural network pruning," *IEEE International Symposium on High-Performance Computer Architecture (HPCA)*, 2024.

[22] H. Zhou, A. Srivastava, H. Zeng, R. Kannan, and V. Prasanna, "Accelerating large scale real-time gnn inference using channel pruning," *Proc. VLDB Endow.*, vol. 14, no. 9, p. 1597–1605, may 2021. [Online]. Available: https://doi.org/10.14778/3461535.3461547

[23] H. Peng, X. Xie, K. Shivdikar, M. A. Hasan, J. Zhao, S. Huang, O. Khan, D. Kaeli, and C. Ding, "Maxk-gnn: Extremely fast gpu kernel design for accelerating graph neural networks training," *ACM/IEEE International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2024. [Online]. Available: https://arxiv.org/abs/2312.08656

[24] T. Chen, Y. Sui, X. Chen, A. Zhang, and Z. Wang, "A unified lottery ticket hypothesis for graph neural networks," *arXiv preprint arXiv:2102.06790*, 2021.

[25] H. Peng, D. Gurevin, S. Huang, T. Geng, W. Jiang, O. Khan, and C. Ding, "Towards sparsification of graph neural networks," *ArXiv*, vol. abs/2209.04766, 2022.

[26] NVIDIA, "Nvidia gh200 grace hopper superchip," 2023. [Online]. Available: https://www.aspsys.com/wp-content/uploads/2023/09/nvidia-grace-hopper-cpu-datasheet.pdf

[27] T. Geng, A. Li, R. Shi, C. Wu, T. Wang, Y. Li, P. Haghi, A. Tumeo, S. Che, S. Reinhardt, and M. C. Herbordt, "Awb-gcn: A graph convolutional network accelerator with runtime workload rebalancing," in *2020 53rd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, 2020, pp. 922–936.

[28] H. You, T. Geng, Y. Zhang, A. Li, and Y. Lin, "Gcod: Graph convolutional network acceleration via dedicated algorithm and accelerator co-design," in *The 28th IEEE International Symposium on High-Performance Computer Architecture (HPCA-28)*, 2022.

[29] R. Hwang, M. Kang, J. Lee, D. Kam, Y. Lee, and M. Rhu, "GROW: A Row-Stationary Sparse-Dense GEMM Accelerator for Memory-Efficient Graph Convolutional Neural Networks," 2022. [Online]. Available: https://arxiv.org/abs/2203.00158

[30] M. Fey and J. E. Lenssen, "Fast graph representation learning with PyTorch Geometric," in *ICLR Workshop on Representation Learning on Graphs and Manifolds*, 2019.

[31] 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 *USENIX Symposium on Operating Systems Design and Implementation*, 2021.

[32] M. Ahmad and O. Khan, "Gpu concurrency choices in graph analytics," in *2016 IEEE International Symposium on Workload Characterization (IISWC)*, 2016, pp. 1–10.

[33] D. Merrill and M. Garland, "Merge-based sparse matrix-vector multiplication (spmv) using the csr storage format," in *Proceedings of the 21st ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, ser. PPoPP '16. New York, NY, USA: Association for Computing Machinery, 2016. [Online]. Available: https://doi.org/10.1145/2851141.2851190

[34] S. Yu, A. Mazaheri, and A. Jannesari, "Auto graph encoder-decoder for neural network pruning," in *Proceedings of the IEEE/CVF International Conference on Computer Vision*, 2021, pp. 6362–6372.

[35] T. Chen, Y. Sui, X. Chen, A. Zhang, and Z. Wang, "A unified lottery ticket hypothesis for graph neural networks," in *Proceedings of the 38th International Conference on Machine Learning*, ser. Proceedings of Machine Learning Research, M. Meila and T. Zhang, Eds., vol. 139. PMLR, 18–24 Jul 2021, pp. 1695–1706. [Online]. Available: https://proceedings.mlr.press/v139/chen21p.html

[36] H. Tessier, V. Gripon, M. Léonardon, M. Arzel, T. Hannagan, and D. Bertrand, "Rethinking weight decay for efficient neural network pruning," *Journal of Imaging*, vol. 8, 2020. [Online]. Available: https://api.semanticscholar.org/CorpusID:247303139

[37] G. Yuan, Z. Liao, X. Ma, Y. Cai, Z. Kong, X. Shen, J. Fu, Z. Li, C. Zhang, H. Peng, N. Liu, A. Ren, J. Wang, and Y. Wang, "Improving dnn fault tolerance using weight pruning and differential crossbar mapping for reram-based edge ai," *2021 22nd International Symposium on Quality Electronic Design (ISQED)*, pp. 135–141, 2021. [Online]. Available: https://api.semanticscholar.org/CorpusID:234479298

[38] H. Zhou, A. Srivastava, H. Zeng, R. Kannan, and V. K. Prasanna, "Accelerating large scale real-time gnn inference using channel pruning," *VLDB Endowment*, vol. 14, no. 9, 2021. [Online]. Available: https://doi.org/10.14778/3461535.3461547

[39] M. Kurtz, J. Kopinsky, R. Gelashvili, A. Matveev, J. Carr, M. Goin, W. M. Leiserson, S. Moore, N. Shavit, and D. Alistarh, "Inducing and exploiting activation sparsity for fast inference on deep neural networks," in *International Conference on Machine Learning*, 2020. [Online]. Available: https://api.semanticscholar.org/CorpusID:225629735

[40] N. Srivastava, G. E. Hinton, A. Krizhevsky, I. Sutskever, and R. Salakhutdinov, "Dropout: a simple way to prevent neural networks from overfitting," *J. Mach. Learn. Res.*, vol. 15, pp. 1929–1958, 2014. [Online]. Available: https://api.semanticscholar.org/CorpusID:6844431

[41] NVIDIA, "Cooperative groups: Flexible cuda thread programming," 2017. [Online]. Available: https://developer.nvidia.com/blog/cooperative-groups/

[42] H. Khan, D. Gurevin, and O. Khan, "Graph input-aware matrix multiplication for pruned graph neural network acceleration," in *2025 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*. IEEE, 2025, pp. 913–925.

Appendix: Artifact Description

Artifact Description (AD)

IX. Overview of Contributions and Artifacts

A. Paper's Main Contributions

The following are the paper's main contributions:

- C_1 An asynchronous execution model for the sparse-sparse-dense (SSD) matrix operator is proposed that decouples its two phases, enabling efficient parallel execution and minimizing communication and data dependency stalls. The operator improves hardware utilization for the accumulation phase under high pruning rates to deliver state-of-the-art computational efficiency on modern GPUs.
- C_2 *Smart Edge Groups (SEGs)* decompose the nonzeros of the graph adjacency matrix to enable the atomic-aware update phase for the SSD operator. SEGs help reduce synchronization stalls and improve the operator's update efficiency on parallel GPU hardware.

B. Computational Artifact

The computational artifact is publicly available on GitHub:

A_1 https://github.com/cag-uconn/ssd-decoupled.git

Artifact ID	Contributions Supported	Related Paper Elements
A_1	C_1, C_2	Figures 8-10

X. Artifact Identification

A. Computational Artifact A_1

Relation to Contributions

Artifact A_1 contains the GPU kernel implementation of the proposed decoupled SSD operator along with the Smart Edge Group (SEG) construction and execution workflow. Specifically:

- For contribution C_1, the artifact implements the asynchronous execution model of the SSD operator. The artifact includes the CUDA kernel and runtime configurations necessary to improve hardware utilization at high pruning rates.
- For contribution C_2, the artifact includes a highly parallel Smart Edge Group (SEG) generation CUDA kernel required to perform the atomic-aware update phase in the SSD operator.

Expected Results

The proposed decoupled SSD operator outperforms MaxK's implementation of the SSD operator across all graphs and pruned embedding dimension sizes from 64 to 2 on the GPU.

Expected Reproduction Time

The artifact can be reproduced in approximately 120 to 150 minutes.

Artifact Setup

1) Hardware: GPU: NVIDIA H100 GPU with compute capability (CC) 9.0. To validate portability across architectures, an additional evaluation is performed on the NVIDIA RTX A6000 GPU (CC 8.6).

2) Software: Docker, NVIDIA Driver (CUDA 12.1 compatible), NVIDIA Container Toolkit.

3) Datasets/Inputs: The graphs can be downloaded, and their metadata can be generated by running the scripts provided in the README.md file on GitHub.

4) Installation and Deployment: All necessary software dependencies are encapsulated within the provided Docker container. Specifically, the container includes Ubuntu 22.04, CUDA 12.1, Python, PyTorch (CUDA 12.1 build), DGL (v24.01), and all required Python libraries. No additional dependencies are required on the host system.

Artifact Execution

The artifact evaluates and compares two GPU kernels for the SSD matrix operator, namely the *Decoupled SSD* kernel and the *MaxK SSD* kernel. The experimental workflow consists of five dependent tasks:

$$T_1 \rightarrow T_2 \rightarrow \{T_{3a}, T_{3b}\} \rightarrow T_4 \rightarrow T_5$$

T_1**: Dataset Acquisition.** Graph datasets are downloaded and extracted inside the container environment. The datasets are stored in CSR-like format under the MaxK-GNN kernel directory and serve as the input to all subsequent stages.

T_2**: Metadata Generation.** Edge Group (EG) metadata is generated using `generate_meta.py`.

T_{3a} **and** T_{3b}**: Kernel Preparation and Build.** Both kernels are compiled independently using CMake in `Release` mode. For the MaxK SSD, the CUDA kernel is isolated via `cleanup.py` before compilation. This produces two standalone executables for benchmarking.

T_4**: Kernel Execution and Profiling.** Each kernel is executed and profiled using `bash run_ssd.sh` and `bash profile_ssd.sh` respectively. Both of these scripts take 5 input parameters that are:

- `graph_name`: input dataset identifier,
- `dim`: unpruned embedding dimension size (e.g., 256),
- `pruned_dim`: pruned dimension size (e.g., 16),
- `repeats`: number of benchmark iterations,
- `print_output`: toggle for matrix output.

T_5**: Automated Full Evaluation.** The script `kernel_driver.py` executes all datasets across both kernels at all pruned dimension sizes and generates a file saved in `logs/summary_experiment.csv` that contains the execution time, the percentage of stalls, and the memory throughput data.

All experiments are executed inside a containerized environment, ensuring controlled compiler versions and reproducible execution.