MergePath-SpMM: Parallel Sparse Matrix-Matrix Algorithm for Graph Neural Network Acceleration

Mohsin Shan, Deniz Gurevin, Jared Nye, Caiwen Ding and Omer Khan
University of Connecticut, Storrs, USA – {mohsin.shan, deniz.gurevin, jared.nye, caiwen.ding, khan}@uconn.edu

***Abstract*—Graph neural networks have seen tremendous adoption to perform complex predictive analytics on massive and unstructured real-world graphs. The trend in hardware accelerator designs has identified significant challenges with harnessing graph locality and workload imbalance due to ultra-sparse and irregular matrix computations at a massively parallel scale. This paper addresses the load imbalance challenge and identifies that state-of-the-art either introduces complex specialized hardware to auto-tune for load-balanced execution at runtime or relies on software-only approaches that exploit parallelism. We propose a novel software-only load-balancing sparse matrix-matrix (SpMM) algorithm that unlocks fine-grain parallelism while maintaining controlled need-based targeted synchronizations to achieve robust performance scaling. The MergePath-SpMM algorithm achieves superior performance using commercial off-the-shelf GPU processors when compared to state-of-the-art hardware accelerators and software-only implementations.**

***Index Terms*—Sparse matrix-matrix, parallel algorithm, merge-path, graph processing, neural networks, GPU, multicore**

I. Introduction

Graph neural networks (GNNs) extend classical deep learning techniques to real-world graphs. GNNs operate on graph structures by aggregating features of connected nodes and edges and then transforming the aggregated embeddings using a neural network model. Recently, many GNN models that combine external features into graph structures have been proposed, such as Graph Convolutional Networks (GCN) [13], GraphSAGE [9], Graph Isomorphism Networks (GIN) [26], and Graph Attention Networks [23]. The efficiency of GCNs has seen rapid adaptation in many real-world graph processing applications. For example, Pinterest's recommender system [28] trained a GCN model with high accuracy for 7.5 billion examples on a graph with 3 billion nodes and 18 billion edges. Despite the success in training massive real-world graphs, performing inference on the GCN models under tight latency constraints has emerged as a hard problem. Since 2020, there has been a resurgent trend in GPU and hardware accelerators for exploiting parallelism in GCNs [2], [6], [7], [10], [12], [14]–[16], [25], [27], [29], [30]. However, due to the highly sparse and irregular nature of the real-world graphs, exploiting data locality and balanced workload execution remains elusive.

Inference in a GCN model requires iterative traversal of the graph nodes and edges using two primary phases: *aggregation* and *combination*. The computation pattern of the combination phase is similar to that of conventional neural networks.

Open-source at https://github.com/cag-uconn/MergePath-SpMM

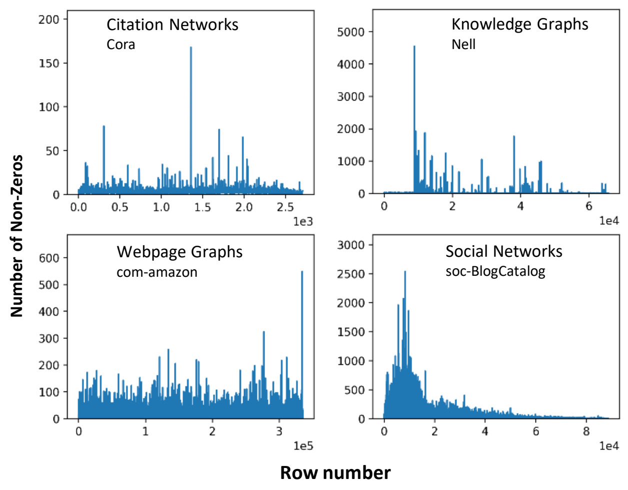


Fig. 1: Graphs from diverse application domains exhibit power law behavior that leads to load imbalance challenges when exploiting parallelism.

However, the aggregation phase relies on the graph structure, which is often sparse and irregular. HyGCN [27] and similar designs [2], [12] adopt the intuitive design strategy where aggregation and combination are realized using separate sparse-sparse (SpGEMM) and sparse-dense (SpMM) matrix multiplication engines. This design strategy suffers from underutilization of either engine due to its graph input dependence, resulting in inter-engine workload imbalance. Hardware accelerators, such as AWB-GCN [6] and GROW [10], as well as GPU accelerators, such as GNNAdvisor [25] unify the hardware design for both phases to mitigate the underutilization problem. These works process moderately sparse feature matrix (X) multiplication with the dense and small weight matrix (W), followed by multiplying the output with the highly sparse and irregular adjacency matrix (A). This results in a workload-efficient computation paradigm that uses a unified SpMM engine.

Power law (or heavy-tail) graphs are ubiquitous and arise in the internet, knowledge, genomics, brain mapping, cybersecurity, social networks, and many other real-world applications [6], [21], [25]. The adjacency matrix of these graphs is proportional to the number of nodes, but the power-law distribution of non-zero elements results in an ultra-sparse and irregular matrix, as shown in Figure 1. These graphs exhibit arbitrarily-long evil rows (non-zero elements usually clustered in just a few rows of A) that lead to workload imbalance when rows or columns are distributed among the processing elements. Moreover, the output of the SpMM kernel operating on the X and W matrices is dense and proportional to the size of the graph nodes. Therefore, when the highly irregular A matrix performs the SpMM kernel with the dense input matrix, the data access locality is challenged. This exacerbates the workload imbalance problem due to irregular memory access patterns.

State-of-the-art hardware accelerators, AWB-GCN and GCoD [6], [29] identify processing evil rows as a major bottleneck for exploiting parallelism in GCNs with thousands of processing elements. They adopt the row-wise parallel SpMM implementation and introduce a dedicated hardware auto-tuner to overcome the evil rows challenge. The auto-tuner identifies the evil rows at runtime and allocates multiple processing elements to each evil row for load-balanced processing of its workload. This specialized hardware support leads to underutilization due to input graph dependence and takes away resources that can otherwise be utilized to exploit parallelism for other rows. We argue that exploiting massive parallelism without specialized dedicated hardware can unlock extreme scalability potential. Is it possible to implement the load-balanced distribution of non-zero elements in the input matrix using a software-only SpMM design?

Recent works have explored GCN acceleration frameworks for GPUs that aim to address the load imbalance challenge using node and edge partitioning at a fine granularity to expose maximum parallelism [5], [25]. GNNAdvisor [25] incorporates an input-driven approach that partitions the nodes among user-parametrizable neighbor partitions of a few non-zeros each. This requires an extension to the compressed sparse row (CSR) format of the adjacency matrix. The neighbor partitions are further distributed among thread chunks along the embedding dimension to maximize parallelism for the node embeddings. At the lowest level, a thread operates on a single embedding dimension of a non-zero. Although GNNAdvisor exposes maximum parallelism among threads, each thread is unaware of the number of sharers for each row in the output matrix. Therefore, each output row update must be performed atomically, leading to fine-grain synchronizations. Such uncontrolled parallelism burdens communication between threads, resulting in lower-than-expected performance scaling at high thread counts.

Research on sparse matrix-vector (SpMV) multiplication aims to balance parallelism and synchronizations in the first matrix since the dimension of the second input is a vector. Merge-path [18] is a state-of-the-art decomposition algorithm that requires no preprocessing overhead, reordering, or extension of the CSR format for the non-zeros in an SpMV kernel setting. It tightly bounds the workload assigned to each thread. It uses a row-wise summation of partial matrix-vector dot products and introduces a novel decomposition to overcome the performance challenges arising from irregular row lengths. The merge-path algorithm exploits the row pointer array of the CSR format, where each thread performs a two-dimensional search to isolate the corresponding region within each list that comprises its share. It produces an equitable partition that ensures that no single thread is overwhelmed by assignment to (a) arbitrarily-long rows or (b) an arbitrarily-large number of zero-length rows. At the end of parallel execution, merge-path SpMV executes a sequential phase where it updates the output values for the rows that are split across multiple threads. This sequential phase is tolerable for SpMV since each accumulation is a single matrix multiplication. However, for SpMM execution, where multiple matrix multiplications are accumulated for each non-zero, the cost of serial execution hampers parallelism. To solve this challenge, we propose a novel MergePath-SpMM algorithm that aims to exploit the parallel execution of partial row accumulations on high core count GPU and CPU processors.

The proposed MergePath-SpMM parallel algorithm does not require hardware support, thus it decouples the GCN accelerator design to focus solely on exploiting massive parallelism. It deploys a novel parallelization strategy that does not incorporate any serial phase. Instead, it explicitly tracks and executes rows that are split across multiple threads using atomic operations. Moreover, it tracks rows that are completely assigned to a single thread and avoid using atomic updates for these rows. The performance scaling potential is demonstrated using a massively multithreaded GPU. The evaluation shows that the MergePath-SpMM unlocks massive fine-grain parallelism in a load-balanced manner to achieve robust performance scaling for both regular and power law graphs. It outperforms the GNNAdvisor baseline for all evaluated dimension sizes of the dense input and output matrices. Furthermore, MergePath-SpMM requires no preprocessing, reordering, or extension of the sparse input matrix.

II. Background and Motivation

Graph neural networks analyze the graph's structure and learn the characteristics of nodes, edges, or even the entire graph [1]. GCNs [13] apply convolutions recursively to extract meaningful information from the graphs. During the aggregation phase, each node iteratively pulls its neighboring nodes' features and updates its feature vector. The new feature vectors are transformed into hidden feature vectors using a neural network in the combination phase. After l layers, each node's output feature vector encapsulates the unique structural information of the node's n-hop neighborhood. $\sigma(\mathrm{A} \times X^{(l)} \times W^{(l)})$ represents the forward propagation of a single convolution layer, l. Consider a graph $G = (n,m)$, and let n denote the number of nodes and m the number of edges. $X^{(l)} \in R^{n\times f}$ denotes the feature matrix of the nodes in layer l, where f is the number of features in each node. $A \in R^{n\times n}$ denotes the graph's adjacency matrix and represents the connectivity information within the graph. $W^{l} \in R^{f\times d}$ is the weight matrix during layer l obtained after training the network. σ is a non-linear activation function such as ReLU or sigmoid.

The adjacency matrix A is ultra-sparse since each node is connected to a few nodes in most real-world graphs. X matrix is moderately sparse since the nodes do not have valid values for all possible features. On the other hand, W contains the model parameters and is a dense, small matrix compared to A and X. State-of-the-art GCN accelerators [6], [7], [10], [25] implement A$\times$(X$\times$W) multiplication order. During this computation, the second SpMM kernel, A$\times$ XW is challenging due to the ultra-sparsity of the adjacency matrix, A. Moreover, due to the unstructured nature of the location of non-zeros in the A matrix, the data accesses of the dense XW matrix are also irregular. All GCN layers require the multiplication of A with their respective XW matrices; therefore, we focus on efficiently performing A $\times$ XW computation. During parallel execution, the two input matrices can be accessed in row or column order, leading to inner, row-wise, column-wise, or outer product combinations of data flow. These strategies affect the data reuse, on-chip memory requirements, and output computation order. The prior GCN accelerators [6], [7], [10], [22], [25] use the row-wise strategy for the A $\times$ XW kernel due to its efficiency and low on-chip memory requirements.

In the row-wise strategy [8], the sparse rows of the A matrix are distributed among the threads. The column index of each non-zero element of a row determines the row index of the XW matrix and computes the partial product for the corresponding output matrix row. The partial products generated by each non-zero element of a row are accumulated in the corresponding output row. The following equation shows the mathematical expression for calculating row i, where the dimensions of matrix A and XW are $n \times n$ and $n \times d$, respectively.

$$C[i,:] = \sum_{j=0}^{n} A[i,j] * XW[j,:]$$

The two popular parallelization strategies for row-wise are row-splitting and nnz-splitting [18], [25]. In the row-splitting strategy, an equal number of rows are split into contiguous chunks and divided among the threads. Since a single thread processes a complete row, the output does not require synchronizations for partial product accumulations. Thus, all GCN hardware accelerators [6], [7], [10], [29] utilize the row-splitting strategy. However, this strategy creates a load imbalance among threads as the number of non-zero elements varies significantly for different rows. Only AWB-GCN and its subsequent variants I-GCN and GCoD explicitly recognize the load imbalance problem due to the power-law distribution of non-zeros. Their solution requires hardware support to detect rows with a disproportional number of non-zeros (evil rows) and assigns multiple processing elements to process the evil rows. This hardware-centric approach is not applicable to a general-purpose processor. Moreover, hardware specialization leads to under-utilization, taking away resources from exploiting parallelism for other rows.

The second parallelization strategy, nnz-splitting aims to divide the number of non-zero elements equally among the threads to solve the load-imbalance problem. Existing GCN frameworks for massively parallel GPU machines implement some form of nnz-splitting [5], [17], [24], [25]. Among these prior works, GNNAdvisor delivers state-of-the-art performance as it aims to harness atomic operations that make better use of the GPU's local private caches and mitigate contention across threads. However, the indiscriminate use of atomic operations for all updates to the output rows imposes an undue burden on communication between threads. Another prior work, merge-path [18] explicitly tracks the non-zeros of rows assigned to a thread as partial or complete rows. For complete rows, the output updates do not require atomic

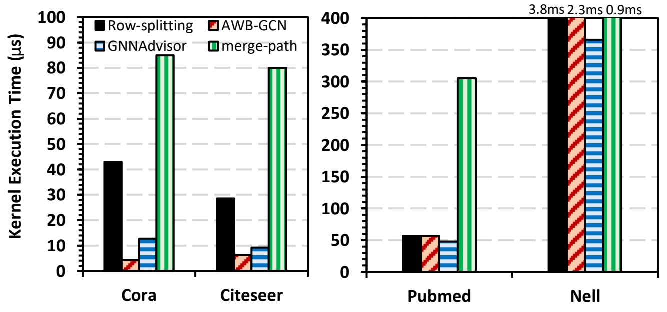


Fig. 2: Performance comparisons of existing hardware accelerator (AWB-GCN) and GPU implementations (GNNAdvisor, merge-path, row-splitting) for representative power-law graphs computing the GCN A$\times$ XW SpMM kernel. Nell uses a hidden dimension size of 64, while others 16.

operations. However, for partial rows, it performs a serial phase to update the output. The serial phase works well for the SpMV kernel since each accumulation is a single matrix multiplication. However, for SpMM execution, where multiple matrix multiplications are accumulated for each non-zero, the cost of serial execution hampers parallelism severely.

To gain insights into the performance implications of row-splitting and nnz-splitting parallelization strategies, Figure 2 shows the kernel completion times for the AWB-GCN [6] hardware accelerator that implements 4096 multiply-accumulate processing elements and executes at 330MHz on an FPGA accelerator setup. For a fair comparison, a GPU with equivalent computational units (NVidia Quadro RTX 6000) is used to evaluate row-splitting, GNNAdvisor [25] and merge-path [18] implementations. The evaluation is done using four representative power-law graphs with their characteristics outlined in Table II (Section IV). The Cora, Citeseer, Pubmed, and Nell graphs are chosen because their A$\times$(XW) execution times for AWB-GCN are reported in Figure 15 of [6].

The Cora and Citeseer are small graphs that do not stress on-chip memory. However, they exhibit irregular memory access with load imbalance stemming from uneven distribution of non-zeros among the rows. AWB-GCN distributes all rows among its processing elements to exploit parallelism. Moreover, it implements a hardware auto-tuner to dedicate a sufficient number of processing elements to sequentially manage each detected evil row. With this setup, AWB-GCN delivers the best-performing execution times (4.3μsec for Cora and 6.3μsec for Citeseer) since it fully exploits the available hardware parallelism for all rows in these small graphs. The GNNAdvisor aims to also exploit the available hardware parallelism by executing a fixed number of neighbor-grouped non-zeros in each GPU warp[1] [4]. However, multiple warps may operate on the same row, thus requiring each output update to be performed using atomic operations. GNNAdvisor

[1]A warp is a set of 32 threads within a thread block. The NVidia GPUs starting with the Volta architecture enable independent thread scheduling among the threads in a warp. This feature allows intra-warp synchronization patterns such that all or a subset of threads in a warp execute independent code.

overcomes the memory stalls induced by the atomic operations by maximizing the number of active warps. However, for these small graphs, the number of warps is constrained to ensure enough work is assigned to each warp (i.e., the number of non-zeros). This results in GNNAdvisor not being able to exploit enough parallelism compared to AWB-GCN, resulting in $\sim 2\times$ lower performance for these graphs. The merge-path [18] implementation aims to eschew the use of atomic operations across all output updates. To accomplish this, it executes all complete rows in parallel but all partial rows are processed in a serial phase. The serial phase severely limits the amount of exploitable parallelism, resulting in the worst performance for these two graphs.

The Pubmed graph is not as irregular as other evaluated graphs, and it has enough rows and non-zeros to expose plentiful parallelism in both AWB-GCN and the GPU implementations. Thus, even row-splitting implementation shows competitive performance against AWB-GCN. The GNNAdvisor also exploits the available parallelism by spawning ~ 4000 warps. Moreover, the parallelism in processing the corresponding atomic operations is better managed as compared to AWB-GCN since the GPU incorporates efficient multi-threading support to hide long latency memory stalls. Consequently, GNNAdvisor outperforms AWB-GCN for this graph. The merge-path implementation favors avoiding atomic operations over exploiting parallelism. Thus, its performance only improves up to a few hundred warps, which is not sufficient parallelism to match the execution times achieved by GNNAdvisor.

The Nell graph has even more rows and non-zeros that lead to much more parallelism in the GPU as compared to the fixed amount of parallelism exposed by AWB-GCN. Moreover, it exhibits extreme power-law behavior that leads to severe load imbalance. The AWB-GCN somewhat mitigates the load imbalance challenge as seen by its improvement over row-splitting. However, due to the lack of exploitable parallelism, its auto-tuner hardware has very limited success. On the other hand, GNNAdvisor spawns sufficient warps on the GPU to exploit parallelism and hide the latency of long-latency atomic updates to deliver $\sim 6\times$ performance gains over AWB-GCN. Even merge-path outperforms AWB-GCN since it optimizes the processing of evil rows. However, the merge path severely limits the amount of parallelism and it is unable to compete with the performance of GNNAdvisor.

Overall, this analysis concludes that GNNAdvisor's approach to maximizing parallelism on the GPU leads to significant performance gains. However, to overcome the load imbalance stemming from the evil rows, the indiscriminate use of atomic operations may not always lead to the best performance. To co-optimize for parallelism and effective utilization of atomic operations, we propose to explicitly track the processing of complete and partial rows. Our approach unlocks massive parallelism for complete rows while only using atomic updates for partial rows to achieve superior performance scaling.

III. MergePath-SpMM Parallel Algorithm

This section first describes merge-path [18] that creates a load-balanced distribution of work among threads for a sparse matrix-vector (SpMV) kernel. At the end of the parallel execution, merge-path SpMV executes a sequential phase where it updates the output values for the rows that are split across multiple threads. This sequential phase is tolerable in SpMV since each thread operates on a single dimension vector. However, for SpMM, each thread operates on multiple dimensions of the dense input and output matrices. Depending on the hardware capabilities to exploit dimension-level parallelism in each thread, any serialization for these computations hampers parallelism. To overcome these performance bottlenecks from the serial phase, a novel parallel SpMM algorithm is proposed that combines the use of atomic operations to unlock parallelism and eschew unnecessary synchronizations by only using atomic updates for partial rows.

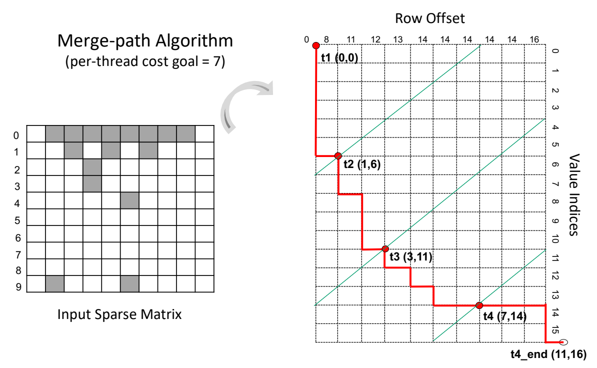
A. Overview of Merge-path for SpMV

Algorithm 1 Merge-path [18] algorithm

```
1: for all (threads) do
2:     $merge\_items = total\_nodes + number\_of\_nonzeros$
3:     $items\_per\_thrd = (merge\_items + num\_threads)/num\_threads$
4:     $cost\_start = min(items\_per\_thrd * core\_id, merge\_items)$
5:     $cost\_end = min(items\_per\_thrd * core\_id + 1, merge\_items)$
6:     $start\_coord$ = BinarySearch(cost_start, RP, nnz, total_nodes)
7:     $end\_coord$ = BinarySearch(cost_end, RP, nnz, total_nodes)
```

Consider the sparse $n \times n$ adjacency matrix A is represented in the CSR format [20], where the *Column Pointer*(CP) array contains the column indices of all non-zeros. The array *Row Pointer*(RP) is of length *n + 1* and encodes the index in *CP* where the given row starts. Also, consider the second $n \times 1$ vector that is represented in coordinate *COO* format, where *RP* array stores the coordinate positions for all non-zeros. The Algorithm 1 outlines the pseudo-code to determine the load-balanced assignment of non-zeros in the sparse input matrix for the merge-path algorithm [18]. It first establishes the total length of the merge path, which is the sum of rows and non-zeros in the adjacency matrix (Line 2). It then computes a per-thread number of merge items by dividing the total length of the merge path by the number of threads (Line 3). This is the *merge-path cost* that the algorithm aims to accomplish per thread. Each thread then computes its lower and upper bound target costs (Lines 4-5). A thread's cost translates to a diagonal in the logical 2D grid, which contains the points having coordinates (i, j) such that $i + j$ = cost. Each coordinate is of the form (i, j), where i is the row number and j is the non-zero number. The goal is to find the first point in the diagonal where RP[i] $\geq j$. This point along the diagonal can be found by performing a constrained 2D binary search along the diagonal (Algorithm 1, lines 6-7). The result of the algorithm is the coordinates of the point that satisfy the constraint. Each thread gets two sets of coordinates, one for the lower and one for the upper bound. All threads process rows and non-zeros in the range of their start coordinate to end coordinate.

To gain a better understanding of the merge-path algorithm, Figure 3 presents a representative example of a sparse matrix with 10 rows and 16 non-zeros. The algorithm creates a logical 2D grid, where the row offset represents the x-axis, and the y-axis is the non-zero value indices in increasing order. Travers-



Thread 1 : start_coord (0,0), end_coord (1,6) → start = 0, end = 0, start_nz = 0, end_nz = 6

Thread 2 : start_coord (1,6), end_coord (3,11) → start = 0, end = 2, start_nz = 6, end_nz = 0

Thread 3 : start_coord (3,11), end_coord (7,14) → start = 2, end = 6, start_nz = 0, end_nz = 0

Thread 4 : start_coord (7,14), end_coord (11,16) → start = 6, end = 10, start_nz = 0, end_nz = 0

Fig. 3: Representative example of distributing adjacency matrix among four threads using Merge-path.

ing along the x-axis in this grid implies processing a row, whereas traversing along the y-axis implies processing each non-zero element. The goal is to find an equitable distribution of non-zeros among threads such that the cost of reading a row and processing non-zeros is taken into consideration. The algorithm computes a per-thread merge-path cost of 7 for four threads since the total length of the merge path is 26. The threads 1 to 4 compute the start costs of 0, 7, 14, and 21, and the end costs of 7, 14, 21, and 27 respectively. The green diagonal line represents the respective diagonal for each thread in Figure 3. In this example, thread 2 searches for a start cost of 7 to find a valid start coordinate. The first coordinate (0, 7) implies that RP[0] results in row offset of 0 that is smaller than the non-zero value index of 7. This breaks the rule and the search continues along the diagonal. The next coordinate (1, 6) satisfies the rule since RP[1] results in a row offset of 8 that is greater than the non-zero value index of 6 for this coordinate. Thread 2 also searches for its end cost of 14 and determines (3, 11) as its end coordinate. Using the start and end costs, thread 2 computes its work assignment as 2 rows (i.e., rows 1 and 2) using the x-coordinates, and 5 non-zeros (i.e., non-zero indices 7 to 11) using the y-coordinates.

After the merge path's start and end coordinates are determined, each thread executes the sparse matrix-vector computations along its merge path. It accumulates the non-zeros for the merge items of the complete rows first and then accumulates any non-zeros for the partial row shared with the next thread. Before ending parallel execution, each thread saves its running total and row ID for subsequent fix-up. Finally, the algorithm executes a serial phase where it updates the output values for all the rows that span multiple threads. As discussed in Section II (cf. Figure 2), this sequential phase severely hampers parallelism in SpMM kernel execution. Next, we present a novel parallel algorithm for SpMM that retains the load-balanced assignment of non-zeros from the merge path algorithm but also exploits parallelism across rows that are shared among multiple threads.

Algorithm 2 Parallel MergePath-SpMM Algorithm

First input is a sparse $n \times n$ adjacency matrix A represented in compressed *CSR* format. The second input is a dense $n \times m$ matrix XW represented in coordinate *COO* format [20].

```
1:  for all (threads) do
2:      if $start\_nz \neq 0$ then
3:          if $start\_row = end\_row$ and $end\_nz \neq 0$ then
4:              $T[0,:] = \sum_{j=start\_nz}^{end\_nz} A[start\_row, CP[j]] * XW[CP[j],:]$
5:              Atomic: $C[start\_row,:] += T[0,:]$
6:              Return
7:          else
8:              $T[0,:] = \sum_{j=start\_nz}^{RP[start\_row+1]} A[start\_row, CP[j]] * XW[CP[j],:]$
9:              Atomic: $C[start\_row,:] += T[0,:]$
10:             $start\_row += 1$
11:     if $end\_nz \neq 0$ then
12:         $T[1,:] = \sum_{j=RP[end\_row]}^{end\_nz} A[end\_row, CP[j]] * XW[CP[j],:]$
13:         Atomic: $C[end\_row,:] += T[1,:]$
14:     for $row \in start\_row$ to $end\_row$ do
15:         $C[row,:] = \sum_{j=RP[row]}^{RP[row+1]} A[row, CP[j]] * XW[CP[j],:]$
```

B. Exploiting Parallelism with MergePath-SpMM

The coordinates obtained from the merge-path algorithm do not specify whether a thread needs to process partial rows. The start coordinate only indicates the row and non-zero IDs a thread must start its processing. Similarly, the end coordinate specifies the row and non-zero IDs for the thread to stop. We introduce two additional variables to track whether the non-zeros assigned for processing start and end are partial rows. If the start coordinate's non-zero ID is equal to the row offset of the start row, then the start row assigned to the thread is a complete row. Otherwise, the start row is a partial row. The variable *start_nz* is set to the non-zero ID for a partial row, and 0 to identify it as a complete row. Similarly, if the end coordinate's non-zero ID is equal to the row offset of the end row, then the end row is a complete row. Otherwise, the end row is a partial row. The variable *end_nz* is set to the non-zero ID for a partial row, and 0 to identify it as a complete row. For example, thread 2 in Figure 3 computes that its start row 1 is a partial row. Therefore, it sets its *start_nz* variable to the non-zero ID of 6. On the other hand, its end row 2 is a complete row that results in setting the *end_nz* flag to 0.

Each thread executes the proposed Algorithm 2 to process the assigned complete and partial rows using the *start_row*, *end_row*, *start_nz* and *end_nz* variables. A thread can have a combination of a partial start row, a partial end row, and complete rows. The algorithm first checks *start_nz*, and its non-zero value indicates that the start row is a partial row (Line 2). A thread may have only one partial row or multiple rows to process. A partial row *only* case is determined by checking if only a single row is assigned to the thread with a valid range of non-zeros to process (Line 3). Each non-zero of the partial *start_row* from matrix A performs the matrix multiplications with all non-zeros (dimensions) of the corresponding row in the dense input matrix XW. The results are accumulated in a thread-local output storage (Line 4). The partial output is then atomically accumulated in the corresponding output for *start_row* (Line 5). Since the thread is assigned to process a single partial row, the thread safely returns (Line 6). However, the thread may be assigned more rows while the first assigned row is a partial row (Line 7). In this case, all non-zeros of the partial *start_row* from matrix A perform matrix multiplications (Lines 8-9). However, the first assigned row is marked processed (Line 10), and the thread continues processing the remaining rows.

A thread may have multiple assigned rows to process, and it is possible to have a partial row at the start and the end of the assignment. Therefore, after processing a potential partial start row, the algorithm checks *end_nz* whose non-zero value indicates that the end row is a partial row (Line 11). Each non-zero of the partial *end_row* from matrix A performs the matrix multiplications with all non-zeros (dimensions) of the corresponding row in the dense input matrix XW. The results are accumulated in a thread-local output storage (Line 12). The partial output is then atomically accumulated in the corresponding output for *end_row* (Line 13). At this point, the thread has completed processing all partial row(s), and must continue processing the remaining complete row(s) in the range [*start_row*, *end_row*). Since this thread is the only one processing complete rows, their outputs are updated without requiring atomic operations (Lines 14-15).

Algorithm 2 creates a load-balanced execution of non-zeros in an SpMM kernel, unlocking the potential for massive thread-level parallelism. However, the atomic operations for output updates may hamper the performance scaling potential. We make a key observation that the atomic operations are limited to start and end partial rows in our algorithm. Moreover, each thread maintains two thread-local arrays, T[0,:] and T[1,:] that accumulate their matrix multiplications locally and only atomically update the corresponding output rows once (cf. Lines 5, 9, 13).

C. Determining Thread Count in MergePath-SpMM

The SpMM kernel operates on dense input and output matrices whose width must match the size of the hidden dimensions for the neural network model. Since the hidden dimensions vary within and across graph neural network models [13], [23], [26], a range of dimension sizes must be considered. A large dimension size implies that multiple multiplications and accumulations are processed for each non-zero assigned to a thread. These operations can be performed serially or in parallel depending on the available hardware-level parallelism. In modern parallel processors, these computations are well suited for single instruction, multiple data (SIMD) execution. Therefore, it is important to map threads to the SIMD units such that the architectural capabilities of the SIMD unit are taken into consideration. When the dimension size is equal to the SIMD unit's capability to process each dimension in parallel, then each thread can be mapped to a SIMD unit. However, when the dimension size is smaller, a one-to-one mapping of thread to SIMD unit leads to under-utilization of SIMD hardware, resulting in diminished parallelism. Similarly, when the dimension size is greater, a one-to-one mapping of thread leads to over-utilization of SIMD hardware, resulting in increased serializations. It is imperative to consider the mapping of threads to SIMD hardware when determining the number of threads for the proposed MergePath-SpMM algorithm.

1) Dimension Size Matches SIMD Unit Lanes: In this scenario, each dimension of a thread is assigned to a lane in the SIMD unit to exploit the available parallelism. In a multi-threaded, throughput-oriented parallel processor (such as a GPU), multiple threads are spawned to exploit thread-level parallelism. However, the thread count must be determined so that each thread performs enough work to amortize the overhead of creating and managing that thread. The number of threads is determined by dividing the total length of merge-path (*merge_items* in Algorithm 1) with the cost of merge-path (*item_per_thrd* in Algorithm 1). Here, the merge-path cost is a programmable parameter that is set to achieve a desirable amount of work per thread. A lower cost implies an increasing number of threads. However, each thread is allocated a smaller number of non-zeros to execute which leads to each thread processing more non-zeros as partial rows (more atomic operations). On the other hand, a higher cost leads to a lower number of threads, but each thread benefits from operating on an increased number of complete rows that do not require atomic operations for output updates. Therefore, tuning for the appropriate merge-path cost optimizes the trade-off between parallelism and synchronizations. The default setting for this parameter is determined empirically using a sensitivity study in the evaluation section.

In graphs with a relatively small number of rows and non-zeros, it is possible that the number of computed threads is too low to exploit the available parallelism. To mitigate this scenario, the merge path cost is decreased to ensure a minimum number of threads are spawned on the system. When the computed threads are below a threshold (e.g., 1024), the total thread count is set to the threshold value.

2) Dimension Size Greater than SIMD Unit Lanes: In this scenario, the number of dimensions to process in each thread exceeds the available hardware lanes in the SIMD unit. This leads to serialization of processing the assigned work, which diminishes parallelism. We propose to break the processing of dimensions across multiple threads such that each thread executes the dimensions that match the number of lanes in the SIMD unit. For example, the NVidia GPU used in this paper supports 32 lanes of SIMD execution in a single warp. If the dimension size is 64, each thread is executed using two warps. Here, the first warp executes the first 32 dimensions while the second warp executes the last 32 dimensions.

The proposed approach leads to an increased number of warps in the GPU since each thread is replicated across warps to ensure a one-to-one mapping of each dimension to a SIMD lane. Therefore, the thread count can be decreased to increase the merge-path cost for each thread, which leads to a decrease in the number of atomic operations. To optimize this trade off between synchronizations and parallelism, the merge-path cost must be appropriately tuned. The default setting for the merge-path cost parameter is determined empirically for a given dimension size when it exceeds the number of lanes in the SIMD unit.

3) Dimension Size Smaller than SIMD Unit Lanes: When the number of dimensions per thread is smaller than the available lanes in the SIMD unit, the hardware is under-utilized leading to diminished parallelism. In modern parallel processors, the SIMD capabilities are increasingly programmable so that the SIMD unit is able to execute multiple independent threads simultaneously. For example, the Volta and later generations of NVidia GPU architectures enable *independent thread scheduling* among the threads in a warp. A set of threads are mapped to a single warp in such a way that a subset of the 32 SIMD lanes operates on the dimensions of each thread. To maximize the use of available SIMD parallelism, we propose to map multiple threads to the same SIMD unit. If the dimension size is 16, two threads execute on a single warp. The first thread occupies the first 16 lanes while the second thread occupies the last 16 lanes of a warp.

The proposed approach leads to a reduced number of warps in the GPU since multiple threads are executed within a warp. The reduced parallelism leads to diminished performance scaling. The total thread count can be increased to increase parallelism. However, this results in a reduced merge-path cost for each thread, which leads to an increase in the number of atomic operations. Therefore, tuning for the appropriate merge-path cost must be done to optimize this trade-off between parallelism and synchronizations. Again, the default merge-path cost parameter is determined empirically for a given dimension size when it is lower than the number of lanes in the SIMD unit.

D. Applicability of MergePath-SpMM

Online versus Offline: The proposed merge path algorithm aims to create a load-balanced distribution of work to achieve scalable parallelism. However, it comes at a nominal scheduling cost that is minimal in static schedulers like the row-splitting technique. For a set of inferences on a given graph, the adjacency matrix may remain stationary or change over time. Therefore, it is possible to consider both online and offline settings. In an online setting, the graph keeps evolving, or a new graph is processed on each inference. Therefore, the MergePath-SpMM schedule needs to be computed for each inference. However, if the adjacency matrix remains stationary across multiple instances, the schedule is computed once and re-used across subsequent inferences. We consider both execution models in the evaluation. However, the offline setting is used as default since the baseline GNNAdvisor framework considers the graph to be pre-processed into user-parameterizable neighbor partitions.

Power law versus Structured Graphs: What happens in graphs that do not follow the power law distribution? For structured graphs, parallelization techniques other than row-splitting may lead to better performance. Therefore, a closed-source NVidia's cuSPARSE parallel implementation library is also used as a baseline SpMM kernel for evaluation.

IV. Methodology

A. GPU, Kernels, and Metrics

The performance evaluation is done using the NVidia Quadro RTX 6000 with 24 *GB* GPU memory, 672 GB/sec memory bandwidth, and 72 symmetric multiprocessors (SMs) with a total of 4608 CUDA cores operating at $1.44GHz$.

Number of Cores	1024 Single-threaded, In-Order @ 1 GHz
Memory Subsystem	
L1–I, LD–D Cache per core	4 KB, 4–way Assoc., 1 cycle
Shared L2 Last-Level Cache	8 KB per-core slice (8*MB* total)
Directory Protocol	Invalidation-based MESI, Limited-4
Num. Memory Controllers	32
DRAM	320 GBps bandwidth, 100ns latency
Electrical 2–D Mesh with XY Routing	
Hop Latency	2 cycles (1–router, 1–link)
Contention Model	Only link contention, 64 bit Flits (Infinite input buffers)

TABLE I: 1000-core simulator parameters for evaluation.

The proposed MergePath-SpMM kernel is compared against the GNNAdvisor [25] and the NVidia's cuSPARSE SpMM kernels. The GNNAdvisor framework is updated with the proposed MergePath-SpMM kernel for its evaluation. The kernel execution time is measured using NVidia's *nvprof* profiling tool.

GNNAdvisor allows the user to set the neighbor-group (NG) partition size, which determines the number of warps to be executed on the GPU. The default setting uses the average graph degree as the NG size in GNNAdvisor. The time required to pre-process the graph to generate partitions for the GNNAdvisor kernel is not included in the reported execution times. The MergePath-SpMM uses the strategy described in Section III-C to set the number of warps for each input graph.

GNNAdvisor supports processing different dimension sizes. In case of dimension size matching the number of SIMD lanes, it utilizes all lanes when executing a warp. When the dimension size is greater than the number of SIMD lanes, it packs all lanes with a subset of the dimensions and serializes the execution of the remaining dimensions in a single warp. However, when the number of dimensions is smaller than the available lanes in a SIMD unit, it under-utilizes the GPU and only uses the lanes that match the dimension size. To better exploit the available hardware parallelism, we propose to extend GNNAdvisor to execute multiple NG partitions per warp when the dimension size is smaller than the number of SIMD lanes. For example, when the dimension size is 16, one NG partition occupies the first 16 lanes while another NG partition occupies the last 16 lanes of a warp. This optimization is included as *GNNAdvisor-opt* in the evaluation.

B. Large Core Count Multicore Simulations

To gain architectural insights into the performance scaling potential in a large multicore setting, we utilize the MIT Graphite multicore simulator [19]. The simulator front-end is updated to support the RISC-V instruction set architecture using the Architecture Description Language (ADL) based functional models [3], [11]. The performance models from Graphite are tailored to support up to 1024 cores. Each core implements a physically distributed private-L1, shared-L2 cache hierarchy, and a 2D mesh on-chip network with X-Y routing to support data accesses and synchronizations among cores. The system implements distributed memory controllers at the chip boundary to exploit parallelism across DRAM accesses. The cache hierarchy size is configured to closely match the total on-chip memory of the GPU used for evaluation. Each core implements a SIMD execution unit capable of executing four 16-bit vector operations. This configuration is used to closely match the $4K$ multiply-accumulate units in the baseline GPU. See Table I for the detailed architectural configurations.

The MergePath-SpMM kernel is compared against GNNAdvisor. Each multi-threaded kernel spawns the number of threads to match the core counts of the processor model. For each matrix kernel, the parallel completion time is measured, i.e., the time spent in the parallel region of the matrix kernel. The completion time is further broken down into compute and memory access latency components to gain deeper insights into performance scaling trends.

C. Input Matrices

All evaluated SpMM kernels use the default dimension size of 16 for the dense input and output matrices. However, a sensitivity study is done for lower and higher dimension sizes of 2 to 128. The MergePath-SpMM utilizes the available SIMD hardware using the thread mapping strategy described in Section III-C.

For the sparse input matrix, a range of real-world graphs is considered, as outlined in Table II along with their relevant parameters. The Type I graphs are all power-law graphs listed with their increasing number of non-zeros in the table. These include all Type I and III graphs from GNNAdvisor [25], as well as Nell from AWB-GCN [6]. Oregon-I, As-caida, Wiki-Vote, email-Enron, email-Euall, soc-SlashDot811, and coAuthorsDBLP are ported from the University of Florida sparse matrix repository[2]. These power-law graphs show a significant variation in the number of non-zeros per node. This is observed in the maximum degree column of the table. For example, Nell graph has 4549 non-zeros in an evil row, whereas the average degree of this graph is 3.9.

Type II are all structured graphs ported from GNNAdvisor. The variability between the average and maximum degrees for these structured graphs is much lower than the counterpart power-law graphs.

Type	Graph Dataset	# Nodes	# Non-zeros	Avg. Deg.	Max. Deg.
I	Cora	2,708	10,556	3.9	168
	Citeseer	3,327	9,228	2.8	99
	Pubmed	19,717	99,203	5.1	171
	Oregon-1	11,492	46,818	4.1	2389
	As-caida	31,379	106,762	3.4	2628
	Wiki-Vote	8,297	103,689	12.5	893
	email-Enron	36,692	367,662	10	1383
	email-Euall	265,214	420,045	1.6	930
	Nell	65,755	251,550	3.8	4549
	PPI	56,944	818,716	14.4	429
	soc-SlashDot811	77,357	905,468	11.7	2508
	artist	50,515	1,638,396	32.4	1469
	com-Amazon	334,863	1,851,744	5.5	549
	coAuthorsDBLP	299,067	1,955,352	6.5	336
	soc-BlogCatalog	88,784	2,093,195	23.6	2538
	amazon0601	410,236	4,878,874	11.9	2760
	amazon0505	403,394	5,478,357	13.6	2760
II	PROTEINS_full	43,466	162,088	3.7	25
	Twitter-partial	580,768	1,435,116	2.5	12
	DD	334,925	1,686,092	5	19
	Yeast	1,710,902	3,636,546	2.1	6
	OVCAR-8H	1,889,542	3,946,402	2.1	5
	SW-620H	1,888,584	3,944,206	2.1	5

TABLE II: Sparse input graphs used for evaluation.

V. Evaluation

Figure 4 shows the speedup of cuSPARSE, GNNAdvisor-opt, and the proposed MergePath-SpMM over the GNNAdvisor baseline using the default dimension size of 16. In this setting, two neighbor groups in GNNAdvisor-opt are mapped in a single warp to fully utilized the available SIMD hardware resources. In the GNNAdvisor baseline, half of the SIMD resources are not utilized since each warp only occupies half the available SIMD lanes. Consequently, GNNAdvisor spawns twice as many warps as GNNAdvisor-opt, but each warp is unable to unlock the available SIMD parallelism. This leads to GNNAdvisor-opt outperforming the GNNAdvisor baseline by a geometric mean of 1.41× for all evaluated power-law and structured sparse input matrices.

Similar to GNNAdvisor-opt, the proposed MergePath-SpMM also maps two threads in a single warp to efficiently exploit the SIMD capabilities. However, it aims to overcome the critical shortcoming of performing all writes to the output matrix using atomic operations in the GNNAdvisor-opt. In MergePath-SpMM, the merge-path cost determines how the work is distributed among threads. A high merge-path cost implies less number of partial rows, which reduces the number of atomic operations on output writes. However, higher cost also implies that the algorithm must spawn a smaller number of total threads to be mapped to warps in the GPU system. This exposes a trade-off between parallelism and synchronizations in MergePath-SpMM. The merge-path cost is a tunable parameter that is determined empirically in Figure 6. The evaluation of sweeping merge-path cost between 2 to 50 shows that for a dimension size of 16, the best-performing merge-path cost is 20. Using this merge-path cost as a fixed parameter, the MergePath-SpMM consistently outperforms GNNAdvisor by a geometric mean of 1.85×, and GNNAdvisor-opt by 31%.

To gain deeper insights, Figure 4 shows the performance results for individual sparse input matrices distributed across Type-I (power-law) and Type-II (structured) graphs. For Type-I graphs that follow a power law distribution, GNNAdvisor, GNNAdvisor-opt, and MergePatth-SpMM all show superior performance against cuSPARSE. These systems aim for a load-balanced execution and explicitly tackle the evil rows challenge, which leads to performance improvements over cuSPARSE. The GNNAdvisor and GNNAdvisor-opt implementations break the evil rows into fine-grain neighbor group partitions but require atomic updates on all writes to the output matrix. The proposed MergePath-SpMM solves the indiscriminate use of atomic updates by selectively determining the per-thread schedule using the merge-path cost as an optimization criterion. The results for Type-I graphs show that MergePath-SpMM outperforms GNNAdvisor-opt in many of the power-law graphs, while the performance advantage is at par for some graphs, such as Wiki-Vote. This behavior stems from the structure of the input graph. When the graph has a relatively small number of input rows and a high average degree, the distribution of non-zeros from different rows in each neigh-

[2]https://www.cise.ufl.edu/research/sparse/matrices/groups.html

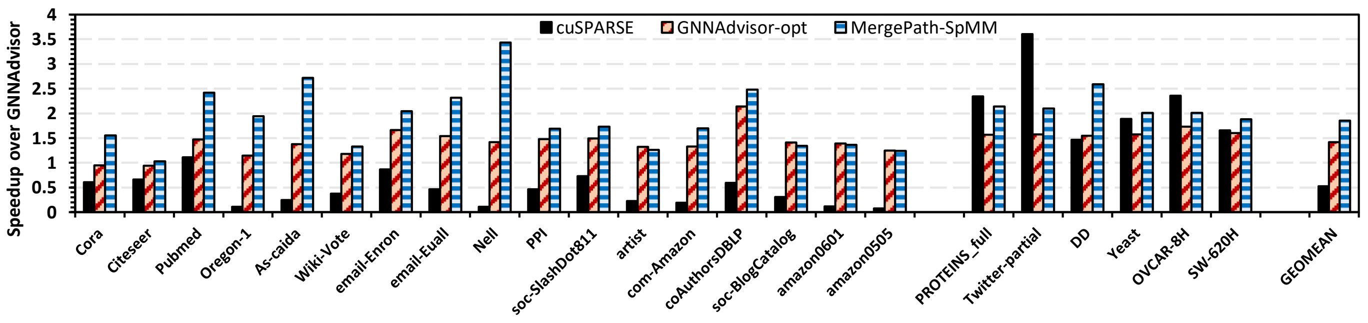


Fig. 4: Speedup of cuSPARSE, GNNAdvisor-opt, and MergePath-SpMM over GNNAdvisor at default dimension size of 16.

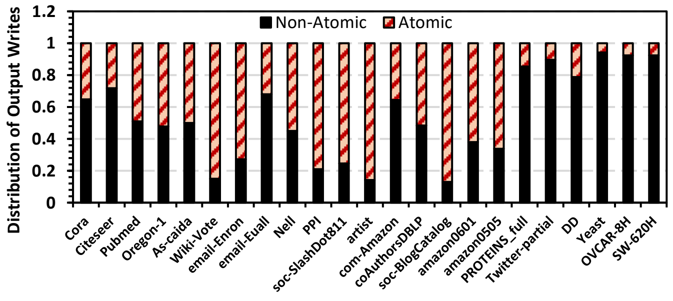


Fig. 5: Distribution of the type of write operations to the output matrix in MergePath-SpMM at dimension size of 16.

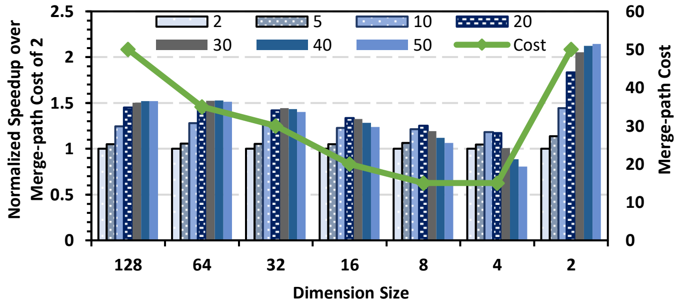


Fig. 6: Normalized performance and the best performing merge-path cost at different dimension sizes.

bor group of GNNAdvisor is similar to MergePath-SpMM using the merge-path cost of 20. Therefore, the number of atomic operations is expected to be higher in both algorithms for such input graphs. Consequently, the performance gap between GNNAdvisor-opt and MergePath-SpMM diminishes. To validate this observation, Figure 5 shows the distribution of atomic and regular output updates for the MergePath-SpMM algorithm. When the input graph has a relatively higher number of atomic updates, the performance improvements of MergePath-SpMM over GNNAdvisor-opt diminish. However, when the number of atomic updates is low, the MergePath-SpMM algorithm shows a significant performance advantage over GNNAdvisor-opt. For example, email-EuAll and email-Enron are similar in terms of the total number of non-zeros, but email-Enron has a much smaller number of rows than email-EuAll. For email-EuAll, the MergePath-SpMM breaks a large number of rows into many complete rows to achieve a load-balanced execution, resulting in fewer atomic updates than email-Enron. The significant reduction in atomic operations in email-EuAll shows an increased performance advantage for MergePath-SpMM as compared to email-Enron.

The Type-II graphs are structured graphs where the number of non-zeros in each row of the sparse input matrix is evenly distributed. Therefore, these graphs do not suffer from the load-imbalance challenge and do not require fine-grain distribution of non-zeros to exploit parallelism. The cuSPARSE closed-source implementation consistently shows superior performance over GNNAdvisor as it has optimized SpMM kernels for these regular input matrices. cuSPARSE is not limited to using row-wise parallelization strategies, and based on the shapes of the input and output matrices, it picks from a slew of available kernels ranging from row-wise, column-wise, inner, and outer product combinations of data flow. The GNNAdvisor-opt improves over GNNAdvisor by increasing the GPU utilization, but it also unnecessarily performs atomic operations for all output updates. This leads to both GNNAdvisor baselines under-performing compared to cuSPARSE for all Type-II structured graphs. The MergePath-SpMM algorithm intelligently detects that breaking rows is not necessary and performs most of its output updates using complete rows. This is evidenced in Figure 5, where almost all of the output updates in MergePath-SpMM are performed using regular write operations for the Type-II graphs. Consequently, MergePath-SpMM eschews unnecessary use of atomic updates and performs comparable to cuSPARSE, as shown in Figure 4. The exception is the Twitter-partial graph where cuSPARSE shows a substantial performance advantage. We evaluated a row-splitting implementation for Twitter-partial (data not shown) that also under-performed by $\sim 2\times$ compared to cuSPARSE. Based on this comparison, we deduce that cuSPARSE is able to utilize a different parallelization kernel for the Twitter-partial input. We make a key observation that the proposed MergePath-SpMM algorithm is a suitable candidate in a diverse kernel selection framework, such as cuSPARSE.

A. Merge-path Cost and Parallelism versus Synchronizations

The amount of parallelism and synchronizations vary at different dimension sizes. In MergePath-SpMM, the tunable merge-path cost parameter determines the tradeoff between the spawned threads and the number of atomic output updates. Therefore, the merge-path cost must be determined independently for each dimension size. Figure 6 shows the

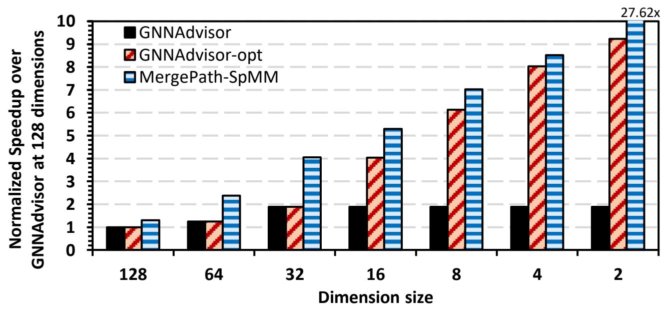


Fig. 7: Speedup at different dimension sizes normalized to GNNAdvisor at the dimension size of 128.

normalized performance for MergePath-SpMM at different dimension sizes. For each dimension size, the merge-path cost is swept from 2 to 50, and the performance is normalized to the merge-path cost of 2. Furthermore, the secondary y-axis shows the best-performing merge-path cost at a given dimension size. When the dimension size is high (128), the MergePath-SpMM shows increasing performance at a higher merge-path cost. In this setting, the resulting threads are replicated across 4× warps to map the 128 dimensions across the SIMD units with 32 lanes each. Therefore, at a higher merge-path cost, the number of spawned threads is afforded to be lower in favor of decreasing the number of non-zeros in partial rows (that require atomic output updates). At 128 dimensions, the merge-path cost is selected as 50. However, when dimensions decrease to 64, the amount of replication of threads across warps also decreases. To compensate for this lack of parallelism, the merge-path cost decreases to 35 for 64 dimensions. At 32 dimensions, each thread is mapped to a warp since it is able to saturate the SIMD unit. Here, the merge-path cost of 30 shows the right tradeoff between parallelism and synchronization.

For the dimension size of 16, two threads are mapped in each warp to efficiently utilize each SIMD unit's hardware parallelism. In this setting, the merge-path cost of 20 is selected since more parallelism is required to support the need for 2× more threads. However, the parallelism is supported by trading off with an increase in the number of atomic operations. At the dimension sizes of 8 and 4, the merge-path cost drops to 15. In these settings, the MergePath-SpMM algorithm selects sufficient parallelism but also keeps the number of atomic operations in check. However, the performance is observed to decrease rapidly for higher merge-path costs as the number of dimensions decreases. At the dimension size of 2, each SIMD unit is mapped with 16 threads to fully utilize the available SIMD hardware parallelism. However, the thread divergence at this extreme setting favors reducing the number of warps in the GPU system. Consequently, the merge-path cost is increased to 50 to optimize for synchronization.

B. *Performance Scalability for different Dimension Sizes*

To gain insights about the performance scaling potential at different dimension sizes, Figure 7 shows the speedup of MergePath-SpMM, GNNAdvisor, and GNNAdvisor-opt at various dimension sizes normalized to GNNAdvisor at the dimension size of 128. As the dimension size decreases, the performance improves for all evaluated kernels. However, the rate of performance improvement is observed to vary dramatically for each kernel. It is observed to saturate for GNNAdvisor at the dimension size of 32. This is due to the inherent limitation of this baseline where the hardware parallelism within the SIMD unit is not fully utilized for dimension sizes below 32. In GNNAdvisor, the number of neighbor groups is determined using the average degree, which does not change for different dimension sizes. At 128 and 64 dimension sizes, more work is performed in each warp relative to the dimension size of 32. Therefore, the performance improves by 2× from the dimension size of 128 to 32. GNNAdvisor-opt matches the performance scaling of GNNAdvisor until 32 dimensions. However, as the threads are mapped more efficiently to utilize the SIMD hardware parallelism at lower dimension sizes, GNNAdvisor-opt shows significant performance gains. At a dimension size of 2, GNNAdvisor-opt achieves ∼9× performance gains over GNNAdvisor at the dimension size of 128.

The MergePath-SpMM shows a significant performance gain of 27.62× over GNNAdvisor as the number of dimensions is reduced from 128 to 2. However, the performance gains relative to GNNAdvisor-opt vary at individual dimension sizes. At the dimension size of 128, MergePath-SpMM spawns threads that are replicated across warps to achieve plentiful parallelism. Since this is done at a high merge-path cost of 50, the number of atomic operations is also kept in check. On the other hand, both GNNAdvisor baselines also have plentiful parallelism, where each warp executes a large number of dimensions to increase the amount of work per warp. Consequently, both GNNAdvisor baselines and the MergePath-SpMM are able to appropriately trade-off between parallelism and synchronizations. This trade off becomes more favorable for MergePath-SpMM at 64 and 32 dimensions since each warp in the GNNAdvisor baselines performs less work and the total amount of parallelism is not adjusted relative to the dimension size of 128. In contrast, the MergePath-SpMM decreases the merge-path cost to compensate for the amount of parallelism to adapt to the number of synchronizations. Therefore, it is able to pull away with a significant performance gain over GNNAdvisor baselines.

The performance advantage of MergePath-SpMM over GNNAdvisor-opt is observed to decrease at the dimension size of 16 as compared to the dimension size of 32. MergePath-SpMM must use two threads to saturate the SIMD hardware in each warp for the dimension size of 16. This comes at the cost of reduced parallelism, which must trade-off with the number of atomic updates by keeping the merge-path cost in check. Therefore, the merge-path cost is decreased from 35 to 20. On the other hand, GNNAdvisor-opt spawns a consistently large number of threads for both dimension sizes since it does not care about the use of atomic operations. Therefore, it is able to keep a high amount of parallelism and utilization of the GPU resources. Consequently, the use of a lower merge-path cost decreases the performance advantage for MergePath-SpMM over GNNAdvisor-opt at the dimension size of 16. This

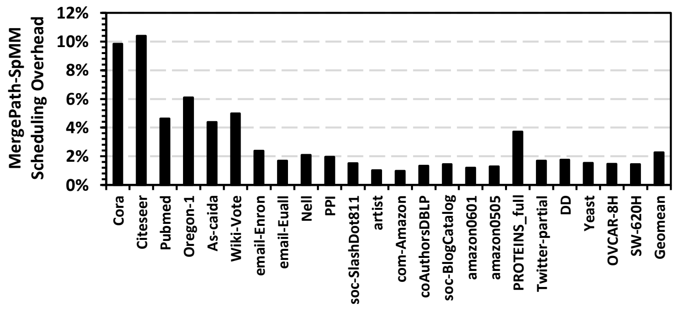


Fig. 8: Scheduling overhead of MergePath-SpMM during online execution in a 2-layer graph neural network setting.

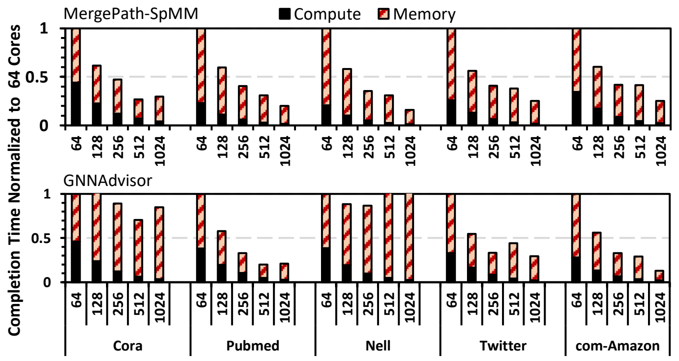


Fig. 9: MergePath-SpMM and GNNAdvisor completion times at increasing core counts normalized to 64 cores.

trend continues for dimension sizes 8 and 4, where MergePath-SpMM lowers the merge-path cost to 15 in order to achieve the right balance between parallelism and synchronizations. At the extreme dimension size of 2, the MergePath-SpMM favors reducing synchronizations over parallelism by selecting the merge-path cost of 50. On the other hand, the GNNAdvisor-opt spawns a consistently large number of warps for this setting. The GPU at this dimension size must process 16 diverging threads per warp and does not perform favorably at a very high number of warps. The MergePath-SpMM is able to adapt to this GPU's architectural dependence and lower the number of warps significantly to achieve superior performance over the GNNAdvisor-opt baseline.

C. *Online versus Offline Execution*

The execution of MergePath-SpMM is evaluated in both online and offline settings. In an offline setting, the algorithm's schedule is processed once for a given input graph and reused as long as the sparse input matrix is not swapped out of the system. On the other hand, the online setting executes the algorithm on each invocation of the graph neural network inference. For example, the GNNAdvisor framework invokes the SpMM kernel twice for a 2-layer GCN model. Figure 8 shows the online scenario where the MergePath-SpMM schedule is computed and stored in global memory before two kernel invocations. The geometric mean scheduling overhead across all input graphs is measured as $\sim$2%. This overhead is generally constant time across different graphs, but its contribution towards the overall execution time increases for small graphs. For example, the scheduling overhead in the smallest Cora graph is highest at 10%, while for large graphs such as com-Amazon, it is under 1%.

D. *Performance Scalability in Large Core Count Multicore*

The RISC-V ecosystem is accelerating the trend towards large core count multicore CPUs. Several processor design companies (MIPS and Esperanto to name a few) have announced single-chip processors with up to 1000 cores on a die. In this massively parallel setup, up to a thousand private caches are kept coherent to enable a scalable distributed cache coherence paradigm. We evaluate the performance scaling potential of the proposed MergePath-SpMM kernel in a 1000-core multicore setting. Figure 9 shows the normalized completion times for the GNNAdvisor and MergePath-SpMM kernels on a simulated multicore processor. The number of cores is varied from 64 to 1024, where a one-to-one thread mapping is used for evaluation. At lower core counts, the total cache capacity is kept consistent by scaling up the size of the per-core cache hierarchy. Moreover, the memory controllers are decreased for smaller core counts, but the total DRAM bandwidth is kept constant. The completion times of the GNNAdvisor and MergePath-SpMM kernels are normalized to their respective 64-core implementation. Due to simulator speed constraints, a set of representative inputs are evaluated using the default dimension size of 16. Cora, Pubmed, Nell, and com-Amazon are picked from the power-law Type-I category to evaluate graphs of different sizes. Moreover, Twitter-partial from the Type-II category is picked to evaluate a structured graph.

GNNAdvisor struggles to demonstrate performance scaling at high core counts for input graphs where evil rows have many non-zero elements compared to the rest of the rows (Cora and Nell). This imbalance results in multiple cores processing the evil row, which leads to costly atomic operations due to an increased number of sharing misses to serialize their execution. This increases the memory stalls that degrade the overall performance. For the remaining graphs where the evil rows do not present a significant challenge, GNNAdvisor demonstrates performance scaling at increasing core counts.

The MergePath-SpMM kernel does not indiscriminately use atomic operations for all output matrix updates. As the number of threads in this setting is fixed to match the core count, the merge-path cost scales up depending on the size of the input graph and the number of cores. A high merge-path cost decreases the number of non-zero elements that are processed using partial rows, which significantly reduces the number of atomic operations. For the evaluated input graphs, only Cora has a small merge-path cost of under 25 at 1024 cores. Therefore, it shows no performance scaling from 512 to 1024 cores. On the other hand, all remaining evaluated graphs use a merge-path cost of greater than 100, and exhibit performance scaling up to 1024 cores. Due to improved performance scaling, MergePath-SpMM improves execution time over GNNAdvisor by 2$\times$ at 1024 cores (data not shown). It is observed that the compute portion of the completion times scale at a high rate with an increase in the core counts. However, the memory stall component shows limited scaling as core counts are increased. In the future, we plan to incorporate efficient data locality and latency-hiding techniques to improve the performance of MergePath-SpMM algorithm for 1000-core processors.

VI. Conclusion

Graph neural networks have shown tremendous potential to unlock predictive analytics on real-world graphs. However, machine learning on graphs involves feature vectors per node that lead to a massive amount of computations and memory accesses in the matrix multiplication kernels. The highly sparse and irregular nature of graphs leads to a sparse matrix-matrix (SpMM) computational paradigm but introduces significant challenges with workload imbalance in a parallel processor setting. This paper makes a key observation that prior work does not take into account the tradeoffs between exploiting parallelism and eschewing unnecessary synchronizations for output matrix updates. A novel merge-based SpMM parallel algorithm is proposed that achieves load-balanced execution on massively parallel processors while co-optimizing parallelism and synchronizations among threads. The evaluation shows robust performance scaling for the proposed MergePath-SpMM algorithm in both GPU and 1000-core multicore processor systems.

VII. Acknowledgments

This work was funded by the Semiconductor Research Corporation (SRC) Artificial Intelligence Hardware program. The authors wish to thank Brian Kahne of Qualcomm and Jose A. Joao of Arm for their continued support and feedback.

References

[1] Sergi Abadal, Akshay Jain, Robert Guirado, Jorge López-Alonso, and Eduard Alarcón. Computing graph neural networks: A survey from algorithms to accelerators. *ACM Comput. Surv.*, 54(9), oct 2021.

[2] Adam Auten, Matthew Tomei, and Rakesh Kumar. Hardware acceleration of graph neural networks. In *2020 57th ACM/IEEE Design Automation Conference (DAC)*, pages 1–6, 2020.

[3] Halit Dogan, Masab Ahmad, Brian Kahne, and Omer Khan. Accelerating synchronization using moving compute to data model at 1,000-core multicore scale. *ACM Trans. Archit. Code Optim.*, 16(1), feb 2019.

[4] Luke Durant, Olivier Giroux, Mark Harris, and Nick Stam. Inside volta: The world's most advanced data center gpu. https://developer.nvidia.com/blog/inside-volta.

[5] Matthias Fey and Jan E. Lenssen. Fast graph representation learning with PyTorch Geometric. In *ICLR Workshop on Representation Learning on Graphs and Manifolds*, 2019.

[6] 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)*, pages 922–936, Los Alamitos, CA, USA, oct 2020. IEEE Computer Society.

[7] Tong Geng, Chunshu Wu, Yongan Zhang, Cheng Tan, Chenhao Xie, Haoran You, Martin Herbordt, Yingyan Lin, and Ang Li. I-gcn: A graph convolutional network accelerator with runtime locality enhancement through islandization. In *MICRO-54: 54th Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO '21, page 1051–1063, New York, NY, USA, 2021. Association for Computing Machinery.

[8] Fred G. Gustavson. Two fast algorithms for sparse matrices: Multiplication and permuted transposition. *ACM Trans. Math. Softw.*, 4(3):250–269, sep 1978.

[9] William L. Hamilton, Rex Ying, and Jure Leskovec. Inductive representation learning on large graphs. In *Proceedings of the 31st International Conference on Neural Information Processing Systems*, NIPS'17, page 1025–1035, Red Hook, NY, USA, 2017. Curran Associates Inc.

[10] 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. In *2023 IEEE International Symposium on High-Performance Computer Architecture (HPCA)*, pages 42–55, Los Alamitos, CA, USA, mar 2023. IEEE Computer Society.

[11] Brian Kahne. FreescaleADL: An Industrial-Strength Architectural Description Language For Programmable Cores. https://source.codeaurora.org/external/adl-tools/adl/tree, 2013.

[12] Kevin Kiningham, Philip Levis, and Christopher Ré. GRIP: A Graph Neural Network Accelerator Architecture. *IEEE Transactions on Computers*, 72(4):914–925, 2023.

[13] Thomas N Kipf and Max Welling. Semi-supervised classification with graph convolutional networks. *arXiv preprint arXiv:1609.02907*, 2016.

[14] Jiajun Li, Ahmed Louri, Avinash Karanth, and Razvan Bunescu. Gcnax: A flexible and energy-efficient accelerator for graph convolutional neural networks. In *2021 IEEE International Symposium on High-Performance Computer Architecture (HPCA)*, pages 775–788, 2021.

[15] Shengwen Liang, Ying Wang, Cheng Liu, Lei He, Huawei LI, Dawen Xu, and Xiaowei Li. Engn: A high-throughput and energy-efficient accelerator for large graph neural networks. *IEEE Transactions on Computers*, 70(9):1511–1525, 2021.

[16] Yixuan Luo, Payman Behnam, Kiran Thorat, Zhuo Liu, Hongwu Peng, Shaoyi Huang, Shu Zhou, Omer Khan, Alexey Tumanov, Caiwen Ding, and Tong Geng. Codg-reram: An algorithm-hardware co-design to accelerate semi-structured gnns on reram. In *2022 IEEE International Conference on Computer Design (ICCD)*, pages 280–289, 2022.

[17] Lingxiao Ma, Zhi Yang, Youshan Miao, Jilong Xue, Ming Wu, Lidong Zhou, and Yafei Dai. Neugraph: Parallel deep neural network computation on large graphs. USENIX ATC '19, page 443–457, USA, 2019.

[18] Duane Merrill and Michael Garland. Merge-based sparse matrix-vector multiplication (spmv) using the csr storage format. In *ACM Symposium on Principles and Practice of Parallel Programming*, PPoPP '16, 2016.

[19] Jason E. Miller, Harshad Kasture, George Kurian, Charles Gruenwald, Nathan Beckmann, Christopher Celio, Jonathan Eastep, and Anant Agarwal. Graphite: A distributed parallel simulator for multicores. In *International Symposium on High-Performance Computer Architecture*, pages 1–12, 2010.

[20] Yousef Saad. *Iterative Methods for Sparse Linear Systems*. Society for Industrial and Applied Mathematics, second edition, 2003.

[21] Alessandra Sala, Haitao Zheng, Ben Y. Zhao, Sabrina Gaito, and Gian Paolo Rossi. Brief announcement: Revisiting the power-law degree distribution for social graph analysis. In *Symposium on Principles of Distributed Computing*, PODC '10, page 400–401, 2010.

[22] Nitish Srivastava, Hanchen Jin, Jie Liu, David Albonesi, and Zhiru Zhang. Matraptor: A sparse-sparse matrix multiplication accelerator based on row-wise product. In *2020 53rd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pages 766–780, 2020.

[23] Petar Veličković, Guillem Cucurull, Arantxa Casanova, Adriana Romero, Pietro Liò, and Yoshua Bengio. Graph attention networks, 2017.

[24] Minjie Wang, Da Zheng, Zihao Ye, Quan Gan, Mufei Li, Xiang Song, Jinjing Zhou, Chao Ma, Lingfan Yu, Yu Gai, Tianjun Xiao, Tong He, George Karypis, Jinyang Li, and Zheng Zhang. Deep graph library: A graph-centric, highly-performant package for graph neural networks. *arXiv preprint arXiv:1909.01315*, 2019.

[25] Yuke Wang, Boyuan Feng, Gushu Li, Shuangchen Li, Lei Deng, Yuan Xie, and Yufei Ding. Gnnadvisor: An adaptive and efficient runtime system for gnn acceleration on gpus. In *USENIX Symposium on Operating Systems Design and Implementation*, 2021.

[26] Keyulu Xu, Weihua Hu, Jure Leskovec, and Stefanie Jegelka. How powerful are graph neural networks? In *International Conference on Learning Representations*, 2019.

[27] Mingyu Yan, Lei Deng, Xing Hu, Ling Liang, Yujing Feng, Xiaochun Ye, Zhimin Zhang, Dongrui Fan, and Yuan Xie. Hygcn: A gcn accelerator with hybrid architecture. In *2020 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, pages 15–29, 2020.

[28] Rex Ying, Ruining He, Kaifeng Chen, Pong Eksombatchai, William L. Hamilton, and Jure Leskovec. Graph convolutional neural networks for web-scale recommender systems. KDD '18, page 974–983, 2018.

[29] Haoran You, Tong Geng, Yongan Zhang, Ang Li, and Yingyan 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.

[30] Bingyi Zhang, Hanqing Zeng, and Viktor Prasanna. Hardware acceleration of large scale gcn inference. In *2020 IEEE 31st International Conference on Application-specific Systems, Architectures and Processors (ASAP)*, pages 61–68, 2020.