LiteForm: Lightweight and Automatic Format Composition for Sparse Matrix-Matrix Multiplication on GPUs

Zhen Peng
zhen.peng@pnnl.gov
Pacific Northwest National Laboratory
Richland, WA, USA

Polykarpos Thomadakis
polykarpos.thomadakis@pnnl.gov
Pacific Northwest National Laboratory
Richland, WA, USA

Jacques Pienaar*
jpienaar@google.com
Google
Mountain View, CA, USA

Gokcen Kestor†
gokcen.kestor@bsc.es
Barcelona Supercomputing Center
Barcelona, Spain

ABSTRACT

Graphics Processing Units (GPUs) have excelled in parallelism and high throughput for dense, regular computations in modern computing. However, sparse computations, such as sparse matrix-matrix multiplication (SpMM), are essential for large-scale, data-intensive applications, where much of the data is inherently sparse. The challenge lies in the sparsity and irregularity of sparse matrices or tensors, which makes achieving high performance on GPU architectures difficult. Consequently, the utilization of suitable sparse data formats is imperative for achieving computational efficiency. Traditional computational libraries often require input in specific formats, which may not accommodate the diversity of matrix characteristics or the varying sparse patterns within a single matrix. While some frameworks support composable formats, they often lack guidance on how to compose these formats effectively or require costly auto-tuning for optimal performance. In this paper, we introduce LiteForm, a novel, lightweight framework designed to automatically compose sparse formats for SpMM computation. We start by presenting CELL, a composable format featuring a three-level blockwise representation that optimizes sparse data for GPUs. LiteForm uses this format and composes it based on the input's characteristics. First, it employs a lightweight model trained to predict whether the CELL format will yield good performance for a given sparse input matrix. Then LiteForm uses a low-overhead predictor and an SpMM cost model to automatically configure the format according to the characteristics of the input matrix. Our experimental evaluation indicates that LiteForm achieves a geometric mean speedup of 2.06×, 1.81×, 1.77×, and 4.18× in comparison to cuSPARSE, Sputnik, dgSPARSE, and TACO, respectively, and demonstrates speedups of 1.26× and 1.52× over state-of-the-art SparseTIR and STile, respectively.

*This research was conducted when the author was a visiting researcher at PNNL.
†This research was conducted when the author was a full-time employee at PNNL.

HPDC '25, July 20–23, 2025, Notre Dame, IN, USA

ACM ISBN 979-8-4007-1869-4/2025/07
https://doi.org/10.1145/3731545.3731574

CCS CONCEPTS

• **Computing methodologies** → **Parallel algorithms**; **Graphics processors**; • **Applied computing** → Mathematics and statistics; • **Software and its engineering** → Compilers.

KEYWORDS

sparse matrix format, format composition, composable format, SpMM, GPU, machine learning, cost model

ACM Reference Format:
Zhen Peng, Polykarpos Thomadakis, Jacques Pienaar, and Gokcen Kestor. 2025. LiteForm: Lightweight and Automatic Format Composition for Sparse Matrix-Matrix Multiplication on GPUs. In *The 34th International Symposium on High-Performance Parallel and Distributed Computing (HPDC '25), July 20–23, 2025, Notre Dame, IN, USA.* ACM, New York, NY, USA, 14 pages. https://doi.org/10.1145/3731545.3731574

1 INTRODUCTION

Graphics Processing Units (GPUs) have become the cornerstone of modern high-performance computing, driving advancements in fields such as deep learning [31], computational physics [7], and bioinformatics [57]. In today's landscape of large language models (LLMs), the number of GPUs an organization possesses is a key indicator of its ability to compete in model training. The efficacy of GPUs is fundamentally attributed to their ability to conduct highly parallel data operations and memory access with efficiency by executing identical operations across multiple data points concurrently, thereby rendering them especially suitable for dense and regular computational tasks.

Conversely, sparse computation emphasizes executing algebraic processes on sparse matrices or tensors—where the majority of components are zeros—by prioritizing the non-zero elements and omitting computations related to zero elements. Among them, sparse matrix-matrix multiplication (SpMM) computes the product of a sparse matrix and a dense matrix by storing only the non-zero elements (as shown in Algorithm 1), thereby avoiding unnecessary computations. Sparse computation is vital in various domains, including machine learning [23], graph analysis [46], and scientific simulations [59], because of its efficiency in both memory usage and computational speed. Its importance also extends to data analytics, allowing scalable analysis of extremely large but sparsely populated datasets, which are common in genomic studies [42], social network analyses [5], and recommendation systems [17].

Algorithm 1: SpMM $C_{ij} = A_{ik} \cdot B_{kj}$, A is in CSR format.

Input: row pointers array *rowPtr*, column indices array *colInd*, values array *val*, number of rows I in A, number of columns J in B, dense matrix B
Output: dense matrix C

```
Function SpMM(rowPtr, colInd, val, I, J, B):
    for i = 0 to I do
        for j = 0 to J do
            for pos = rowPtr[i] to rowPtr[i+1] do
                k = colInd[pos];
                C[i][j] += val[pos] * B[k][j];
    return C;
```

Sparse computation on GPUs presents significant challenges due to their architectural design, which is optimized for dense and regular computations [53]. First, irregular memory access patterns in sparse operations lead to inefficient use of GPU memory bandwidth and cache [23], while GPUs are optimized with coalesced memory access in dense computations. Second, sparsity results in an uneven distribution of workloads across GPU threads, with certain threads handling a greater number of non-zero elements. This imbalance contributes to the overhead of synchronization and the inefficient use of parallel resources [2]. Third, warp divergence arises from the conditional nature of sparse computation, causing threads within a warp to follow different execution paths [66], which reduces the efficiency of Single Instruction, Multiple Threads (SIMT) operations.

Choosing the right sparse format is crucial for optimal GPU performance because each format involves trade-offs in terms of memory usage, computational efficiency, and data access. The appropriate format depends on the matrix's sparsity pattern and the specific operations being performed. Each sparse matrix has unique characteristics, making it challenging to find a single format that performs well across all matrices. Moreover, a single matrix can contain different sections with varying non-zero patterns, further complicating the selection process. The problem of sparse matrix decomposition has been proven to be NP-hard, highlighting the complexity of this task [15]. An ill-suited format can lead to suboptimal performance, underscoring the importance of selecting a format that effectively balances these trade-offs.

Existing efforts to address the challenge of suboptimal performance in sparse computation have focused on either optimizing fixed formats or supporting composable formats. Libraries like NVIDIA cuSPARSE [38]and Triton [51] offer highly optimized, target-specific sparse kernels. Although these kernels can deliver high performance, their efficiency varies depending on inputs and sparse formats, often requiring users to experiment and find the most suitable format. Recent advances in composable formats, such as SparseTIR [63] and STile [15], provide frameworks to create customized formats for specific matrices. However, these approaches often lack clear guidance on how to compose optimal formats or require costly auto-tuning per input, making them increasingly impractical when the number of inputs increases.

In this paper, we introduce LiteForm, a novel, lightweight framework that automatically composes formats for SpMM on GPUs. First, we design the Composable Ellpack (CELL) format, specifically tailored for GPU architectures. CELL uses a three-level blockwise data structure that improves data locality and load balancing. It divides columns into partitions and enhances the traditional Ellpack format by allowing flexible row lengths, grouping rows with similar lengths into buckets, and clustering non-zero elements into blocks. Second, LiteForm incorporates a lightweight machine learning model to predict whether a given sparse matrix should use the CELL format. This model is trained on basic matrix features like dimensions, non-zero counts, and density, avoiding the need for costly preprocessing. Finally, LiteForm uses a low-cost classifier and an SpMM cost model to configure the CELL format for a given matrix without rerunning the kernel at runtime. In summary, LiteForm improves data locality and warp efficiency by grouping non-zero elements in a matrix, and ensures load balancing by keeping the number of non-zeros consistent across the blocks, dedicating itself to enhance GPU performance.

The contributions of this work are as follows.

- We propose the Composable Ellpack (CELL) format, which utilizes a three-level blockwise data layout to improve data locality and load balancing.
- We introduce a lightweight machine learning model trained on basic matrix characteristics to predict whether a given sparse matrix should be represented in the CELL format.
- We utilize a low-overhead classifier and an SpMM cost model to configure the CELL format automatically, eliminating the need for costly runtime auto-tuning.

LiteForm achieves a geometric mean speedup of 2.06×, 1.81×, 1.77×, and 4.18× compared to cuSPARSE, Sputnik, dgSPARSE, and TACO, respectively, along with speedups of 1.26× and 1.52× over state-of-the-art SparseTIR and STile, respectively, while incurring significantly lower overhead by orders of magnitude.

2 BACKGROUND AND MOTIVATION

This section provides an overview of common sparse formats, prior research, and their limitations, which serve as the motivation for this work.

2.1 Sparse Formats

A sparse format determines how the values of a sparse matrix are stored. The primary goals are to 1) minimize space by storing only non-zeros (and zero padding when necessary), thereby eliminating redundant computations involving zeros, and 2) enhance memory access efficiency by organizing elements in contiguous memory. Over time, numerous sparse storage formats have been developed, each offering unique storage patterns and computational properties.

Elementwise Formats. Several elementwise formats have been proposed throughout the years, including COO [43], CSR [52], DCSR [6], Ellpack (ELL) [29], HiCOO [33], CSF [47], DIA [24], etc. Each of these formats offers distinct advantages for different types of computational tasks. For example, the Coordinate (COO) format [43] stores non-zero entries by recording their full coordinates, representing the matrix as a list of row and column index pairs. This leads to redundancy, as the row index is recorded multiple times for entries in the same row. In contrast, the Compressed Sparse Row (CSR) format [52] reduces this redundancy by using

Table 1: Prior work on Sparse Computation on GPUs.

Type	Work	Automatic Format Selection	Sparsity-Pattern Awareness	Format Construction Overhead
Fixed Format	cuSPARSE [38]	✗	✗	Low
	Triton [51]	✗	✗	Low
	TACO [30]	✗	✗	Low
	Sputnik [18]	✗	✗	Low
	dgSPARSE [12]	✗	✗	Low
Automatic Selection	Auto-SpMV [4]	✓	✗	Low
	SpTFS [49]	✓	✗	Low
	IA-SpGEMM [60]	✓	✗	Low
	AlphaSparse [13]	✓	✗	Low
	Seer [50]	✓	✗	Low
Composable Format	SparseTIR [63]	✗	✓	High
	STile [15]	✓	✓	High
	LiteForm (this work)	✓	✓	Low

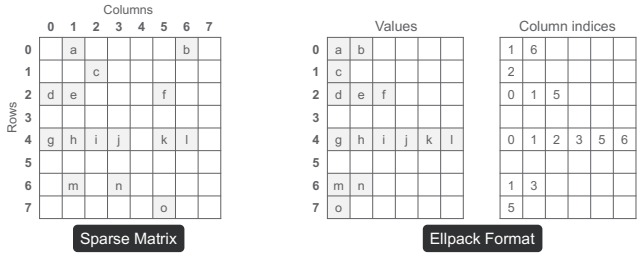

Figure 1: An example of the Ellpack format. Non-zero elements in a row are packed to the left. A long row may result in a high amount of zero padding.

a array *rowPtr* to store the starting position of each row. Figure 1 shows an example of Ellpack format that stores values and column indices in two dense matrices by packing non-zeros to the left.

Blockwise Formats. Blockwise formats divide a matrix into smaller submatrices or blocks. For example, the Block Compressed Sparse Row (BCSR or BSR) format [25] is the blockwise version of CSR. In BCSR, a sparse matrix is divided into equal-sized blocks. Any block that contains at least one non-zero element is treated as a non-zero block, and zero padding is added to construct a full block. Similarly, the Blocked Ellpack (BLOCKED-ELL) format [9] and Sliced Ellpack (SLICED-ELL) format [35] are blockwise variations of the ELL format.

In general, utilizing blocks in sparse matrix formats provides several performance benefits on GPUs. First, values within small blocks can be loaded into shared memory or registers, facilitating data reuse and significantly reducing global memory accesses. Second, the structured and regular nature of blocks enables aligned memory accesses, which optimizes memory bandwidth utilization and minimizes the number of memory transactions. Third, blocks create opportunities for advanced optimizations, such as loop unrolling, which can enhance instruction-level parallelism and boost overall computational efficiency.

However, blockwise formats also present notable challenges. One key issue is that the padding ratio in blockwise formats can be high for very sparse matrices. A high padding ratio increases both the memory footprint and computational overhead. Another challenge arises when dealing with extremely dense rows that the number of non-zero elements is nearly equal to the total number of columns, resembling a dense matrix. Blocks (e.g., in BCSR and ELL) containing these dense rows often have a substantial amount of padding, causing the memory footprint to grow too large for GPUs to efficiently manage. For example, in one of our experiments using BCSR with a block size 8×8, we ended up with an increase in the memory footprint of more than 60×. The padding ratio reached as high as 99%, meaning that only 1% of the elements in a non-zero block were actual non-zero values. This padding significantly restricted the usage of the format to much smaller problem sizes, limiting the performance benefits of the blockwise approach.

2.2 Prior Work on GPU Sparse Computation

Extensive research has been conducted to address the performance challenges in sparse computations such as SpMM. Table 1 evaluates the state-of-the-art approaches according to three metrics:

- **Automatic Format Selection.** This assesses whether the approach automatically selects the most suitable data representation from existing fixed formats for the entire input. Although automatic selection can leverage execution history, a fixed format lacks the flexibility to accommodate varying sparsity patterns across different sections of the input. As a result, it may miss opportunities for further optimization.
- **Sparsity-Pattern Awareness.** In real-world applications, sparse matrices often exhibit a wide variety of sparsity patterns. It is crucial to accommodate different matrix types, ranging from highly irregular matrices to matrices with more predictable sparsity patterns. Meanwhile, it is also critical to detect unique patterns in different parts of a matrix. The ability to adapt to various sparsity patterns ensures consistent and improved performance across diverse scenarios.

- **Format Construction Overhead.** Once the sparsity pattern of the input is identified, determining the appropriate composition configuration to accurately represent these patterns becomes crucial. This process often requires extensive exploration of the search space and may involve repeatedly running the computational kernel or other microbenchmarks. These iterations contribute to the overall format construction overhead.

Fixed Sparse Format. This set of methods focuses on optimizing kernels using fixed sparse formats. Examples include sparse computing libraries such as NVIDIA cuSPARSE [38], Triton [51], Sputnik [18], dgSPARSE [12], Intel MKL [26], and compilers such as TACO [30]. Although the kernels provided by these libraries are highly optimized, their efficiency varies depending on the input and the chosen sparse format. In practice, different formats can lead to significantly different performance outcomes for a given matrix. However, because these frameworks rely on a fixed format to represent an input sparse matrix, they cannot adapt to varying sparsity patterns across different matrices, limiting their ability to fully optimize performance.

Automatic Format Selection. The second category of work leverages machine learning techniques for format selection in sparse computations (e.g. Seer [50], IA-SpGEMM [60], SpTFS [49], Auto-SpMV [4], AlphaSparse [13], Morpheus-Oracle [48]). For example, Auto-SpMV [4] introduces an automated framework for optimizing Sparse Matrix-Vector Multiplication (SpMV) kernels on GPUs. The framework leverages machine learning to explore the optimization space to automatically select the best-performing SpMV kernel for a given sparse matrix. Seer [50] is a predictive runtime framework for sparse format and kernel selection. It uses machine learning to predict the optimal kernel during execution based on features such as input size and structure. Seer dynamically selects the best-performing kernel from a set of candidates, optimizing for irregular computational patterns. For SpMV, Seer leveraged the entire SuiteSparse Matrix Collection to gather kernel runtime data. These approaches typically select a single format that applies uniformly throughout the input matrix, ignoring variations in sparsity patterns. This lack of flexibility prevents an optimal representation of regions with different sparsity characteristics. In addition, they often require large datasets for training to cover a variety of formats, resulting in significant training overhead.

Composable Sparse Formats. The third category of methods supports composable formats that partition sparse matrices and represent each partition using different configurations or formats. Frameworks in this category include SparseTIR [63], STile [15], and TileSpMV [36]. Some of these approaches identify sparsity patterns in the input and adjust their data representation accordingly. However, they do not automate the format composition process. For instance, SparseTIR offers a sparse tensor compilation abstraction that supports composable formats and transformations for deep learning workloads. SparseTIR depends on an exhaustive search in the space to identify the optimal configuration of composable formats, which introduces significant overhead. This requirement adds complexity and cost to overall performance optimization, particularly for large and diverse datasets. STile focuses on searching for hybrid sparse formats tailored to sparse deep learning operations. It supports selecting from three different formats to represent various parts of a matrix and employs a cost model to identify the optimal combination of these formats. The cost model is refined through a microbenchmarking process based on the Roofline model [58]. However, these methods rely heavily on microbenchmarks and require rerunning the kernel multiple times to fine-tune the configuration, leading to high overhead and extended tuning times.

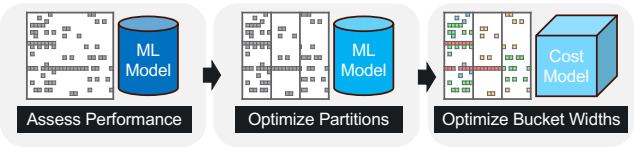

Figure 2: A workflow overview of LiteForm: 1) an ML model assesses CELL performance for a matrix, 2) an ML model determines the number of partitions, and 3) a cost model helps search for the bucket widths.

3 OVERVIEW OF LITEFORM

The LiteForm framework is designed to optimize SpMM on GPUs by automatically composing the Composable Ellpack (CELL) format. The CELL format is a three-level blockwise structure that organizes non-zero elements across both the column and row dimensions. In the column dimension, the matrix is divided into *partitions*. In the row dimension, rows with similar lengths in a partition are grouped into *buckets*. The same number of elements are further grouped as *blocks*, which is regarded as a basic workload unit. By dynamically adjusting partitions and bucket widths, the format CELL improves memory access efficiency and load balancing, particularly for matrices with varying sparsity patterns.

The core of LiteForm is an automated format composition process driven by two machine learning (ML) models and an SpMM cost model. Figure 2 shows the workflow overview of LiteForm. First, an ML model predicts whether the CELL format will outperform fixed sparse formats for a given sparse matrix. Second, for the matrix in CELL, another ML model determines the optimal number of partitions. Third, LiteForm utilizes a cost model along with a search algorithm to compose the buckets and blocks. The search algorithm uses the cost model to estimate the computation cost of the buckets and then explores the search space for the optimal bucket width. Overall, this process allows LiteForm to adapt the CELL format to the characteristics of the input matrix, improving computational performance by minimizing padding, reducing unnecessary computations, and maximizing GPU utilization.

4 COMPOSABLE ELLPACK (CELL) FORMAT

This section introduces the Composable Ellpack (CELL) format, a three-level blockwise sparse format based on Ellpack. As shown in Figure 3, the CELL format first evenly divides the columns of the sparse matrix into multiple *partitions*. Within each column partition, rows are organized into different *buckets* based on the number of non-zero elements in each row (i.e., the length of row) [35]. Specifically, bucket i has a bucket width of 2^i and groups rows with lengths l that satisfy $2^{i-1} < l \leq 2^i$. Each bucket forms a submatrix in the Ellpack format, with rows padded with zeros to match

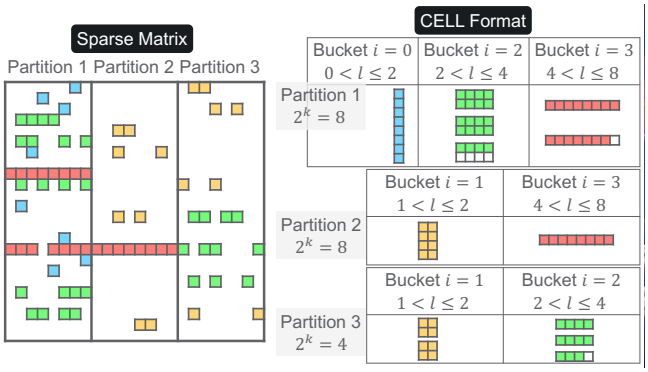

Figure 3: A CELL format example. The matrix is divided into column *partitions*. A partition collects the rows within the same row length (l) range together to form *buckets*. A bucket groups the same number of non-zero elements (2^k) into *blocks*. The number 2^k is set as one or multiple times of the maximum bucket width of the partition.

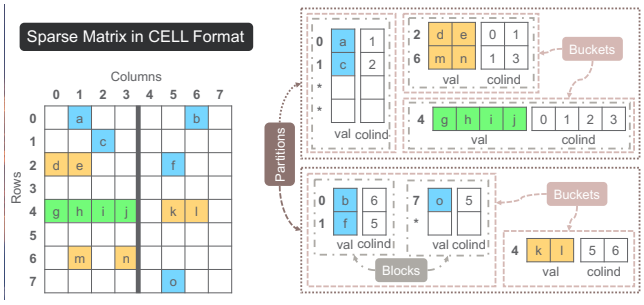

Figure 4: Data structures in the CELL format. A bucket is represented by a row index array, a column index array, and a value array. A row index "$*$" indicates zero padding.

Algorithm 2: SpMM $C_{ij} = A_{ik} \cdot B_{kj}$ within a CELL *block*.

Input: number of rows in the block *numRowsInBlock*, bucket width W, row indices array *rowInd*, column indices array *colInd*, values array *val*, number of columns J in B, dense matrix B

Output: corresponding block in dense matrix C

```
1  Function SpMMInCELLBlock(numRowsInBlock, W,
     rowInd, colInd, val, J, B):
2      for iBlk = 0 to numRowsInBlock do
3          i = rowInd[iBlk];
4          for j = 0 to J do
5              local = 0;
6              for kBlk = 0 to W do
7                  k = colInd[iBlk][kBlk];
8                  local += val[iBlk][kBlk] * B[k][j];
9              if !needAtomic then
10                 C[i][j] += local;
11             else
12                 atomicAdd(C[i][j], local);
```

the bucket width. Additionally, for bucket i in a column partition, every 2^{k-i} rows are grouped into a *block*, and the corresponding computation is assigned to thread blocks on the GPU. Within a block, the number of non-zero elements processed is 2^k, which can be set as either one or multiple times of the maximum bucket width. The computational pattern within a non-zero block is shown in Algorithm 2. This design enhances data locality and load balancing, optimizing performance for sparse matrix computations on GPUs.

In the implementation, each element in the CELL format is associated with its row index, column index, and value. As shown in Figure 4, a bucket is represented by three arrays: a row index array *rowInd*, a column index array *colInd*, and a value array *val*. For an element at position $[i, j]$ (where i and j represent its coordinates in the bucket), the row index from the original matrix is stored in $rowInd[i]$, while the column index and value are stored at positions $[i * W + j]$ in the *colInd* and *val* arrays, respectively, where W is the bucket width. This structure allows extremely long rows to be split into multiple rows in a bucket when the number of non-zero elements in the row exceeds the bucket width. In such cases, the row will appear multiple times in the *rowInd* array. When different threads process rows with the same row index, the atomic operation is added (as shown in line 12). In general, a partition consists of a list of buckets, and a sparse matrix is represented by a list of partitions.

The CELL format is designed to enhance computational efficiency on GPUs. First, using column partitions improves granularity and reduces the padding ratio across the matrix, particularly when dealing with extremely long rows. Second, row buckets enhance the regularity of sub-matrices by aligning rows to the same size, forming blocks that are optimized for GPU processing. Although SparseTIR [63] introduced the *hyb* format, which also uses partitions and buckets, it enforces the same set of bucket widths across all partitions. In contrast, the CELL format allows for different sets of bucket widths in each partition, offering greater flexibility to accommodate varying sparsity in the input data, which significantly improves data locality and memory utilization.

5 AUTOMATIC FORMAT COMPOSITION

This section outlines the necessary steps involved in the automatic composition of the CELL format in LiteForm. First, the process begins by determining whether the CELL format is more advantageous compared to a fixed blockwise format. Second, once this assessment is made, the appropriate column *partitions* are established, Third, LiteForm organizes rows into *buckets* based on the number of non-zero elements and forms *blocks* as the processing unit, which involves selecting the optimal bucket widths for each partition to ensure efficient handling of the matrix. These steps allow LiteForm to dynamically adapt to varying sparsity patterns within input data, thereby optimizing both memory usage and computational performance.

5.1 Assessing the Performance Impact of CELL

The first step in the automatic composition of the CELL format involves determining whether it offers a significant performance advantage over fixed formats CSR and BCSR that are representatives of elementwise formats and blockwise formats, respectively. This decision is driven by a pre-trained machine learning (ML) model designed to predict whether the CELL format will provide better computational performance compared to the fixed blockwise approach.

The ML model is trained using a variety of matrix features and the history of the execution time of different matrices. The chosen features are listed in Table 2, which have been used in prior research to characterize matrices [4, 48, 49, 60]. By analyzing these features, the ML model can assess whether the flexibility of the CELL format—with its ability to adjust bucket widths and dynamically partition the matrix—will lead to improvements in data locality and memory utilization over the fixed formats.

To gather training data for the ML model, LiteForm executes the SpMM kernel on a carefully selected set of matrices using both the CELL and the fixed formats. These matrices are chosen from diverse application domains to cover a broad range of sparsity patterns, ensuring that the model is trained on a representative sample of real-world matrices. For each matrix, the execution times for all formats are recorded. For the CELL format, the kernel is executed multiple times to identify the configuration that delivers the best execution performance. Although this tuning process incurs an initial cost, it is amortized over future uses of the model, ensuring that the performance gains outweigh the upfront overhead in the long run. If the CELL format demonstrates a speedup of more than 1.1× compared to both fixed formats, the matrix is labeled "TRUE." Otherwise, it is labeled "FALSE." These labels, along with the matrix characteristics, form the training data set for supervised learning. This training approach guarantees that the ML model is equipped to predict, based on the characteristics of the incoming matrices, whether the CELL format will yield better performance than the fixed approaches.

Once the model makes a prediction, the composition process proceeds only if the CELL format is expected to offer a tangible performance benefit. This ensures that the overhead associated with configuring the CELL format is justified by the anticipated performance improvements. If the fixed formats is deemed sufficient, the system avoids unnecessary complexity and computation.

Table 2: Sparse matrix features used to predict whether CELL offers a performance advantage.

Feature to Predict Format
number or rows
number of columns
number of non-zero elements
average number of non-zeros per row
minimum number of non-zeros per row
maximum number of non-zeros per row
standard deviation of non-zeros per row

5.2 Optimizing Partitions in CELL Composition

The CELL format, as a three-level blockwise structure, organizes non-zero elements across both column and row dimensions. In the column dimension, the matrix is divided into *partitions*, which is especially beneficial for matrices with extremely long rows, as these partitions break the rows into smaller parts, reducing the overall padding ratio. In the row dimension, rows with similar lengths are grouped into *buckets*, and the same number of elements form *blocks*, enhancing the regularity within each processing unit. The performance of the CELL format for any given matrix depends on the careful selection of partitions and bucket sizes. Therefore, optimizing these parameters is essential to maximize computational efficiency and performance.

Once the CELL format is predicted to outperform the fixed formats, the next step to compose the CELL format is to determine the optimal number of partitions. Since the number of partitions is an integer, this problem can be framed as a typical classification task in machine learning. Using a classification model, LiteForm can predict the appropriate number of partitions based on the input data. The features used for the prediction include the density of the sparse matrix and the size of the dense matrix, as outlined in Table 3. Density is defined as the ratio of non-zero elements to the total size of a dense matrix. Through testing, we found that using density values rather than absolute values significantly improves prediction accuracy. This is because the design of column partitions is closely tied to the matrix's sparsity pattern and non-zero distribution, and density provides more direct information about the input matrix's sparsity than raw absolute values. Additionally, the size of the dense matrix is considered a key feature, as it affects memory layout and, consequently, the kernel's performance.

To generate training data for the machine learning classifier, we run the sparse kernel SpMM on a small set of input matrices, as described in Section 5.1, while varying the size of the dense matrices to candidates of 32, 64, 128, 256, and 512 to cover a range of possible scenarios. The training data is collected by recording the execution time along with the density features of the sparse matrix and the size of the dense matrix.

To select the best model for predicting the number of partitions, we evaluated several machine learning classification models. The evaluation metrics include not only accuracy but also similarity between predicted and actual values. This is important because

Table 3: Sparse matrix features to predict the optimal number of partitions in CELL.

Feature to Predict Number of Partitions
number of rows
number of columns
number of non-zero elements
average density of non-zeros per row
minimum density of non-zeros per row
maximum density of non-zeros per row
standard deviation of non-zero density per row
product of other dimensions in the kernel

partition numbers that are close in value tend to produce similar performance results. For a given sparse matrix and a specific dense matrix size, the similarity between the predicted number of partitions $\hat{p}$ and the actual value p is measured as the relative difference:

$$\text{similarity} = 1 - \frac{|p - \hat{p}|}{max(p, \hat{p})} . \tag{1}$$

For a given sparse matrix across multiple dense matrix sizes (e.g., 32, 64, 128, 256, and 512), the similarity between the predicted partition vector $\hat{P}$ and the actual partition vector P is measured using cosine similarity:

$$\text{similarity} = \frac{P \cdot \hat{P}}{\|P\|\|\hat{P}\|} . \tag{2}$$

The similarity measurement can be used as a loss function to evaluate the machine learning classification models in the sense that similar partition numbers can lead to similar performance. LiteForm utilizes the pre-trained model to predict the optimal number of partitions for an unseen sparse matrix.

5.3 Optimizing Bucket Widths in CELL Composition

After partitioning the sparse matrix into column partitions, the next step in constructing the CELL format is to build the corresponding buckets and blocks. Each column partition generates its own set of buckets based on the distribution of row lengths within that partition. According to the CELL design, the bucket i is assigned a *bucket width* of 2^i, and rows with lengths l (the number of non-zero elements in the row) that satisfy $2^{i-1} < l \leq 2^i$ are placed in this bucket. Furthermore, every 2^{k-i} rows in the bucket i are grouped into a *block*, which is mapped to a thread block on the GPU. Each block contains 2^k non-zero elements, where 2^k is set as a multiple of the maximum bucket width.

The initial buckets in a partition are constructed based on the lengths of all rows within that partition. However, this initial configuration may be suboptimal due to the presence of extremely long rows. The bucket containing these long rows will be assigned the maximum bucket width, and when rows are grouped into blocks, the block size is determined by this maximum width. If the maximum bucket width is excessively large, many blocks will lack sufficient non-zero elements, leading to load imbalance when these blocks are mapped to GPU thread blocks.

To mitigate the impact of long rows, the CELL format allows a single row to be represented as multiple rows within a bucket, all sharing the same row index. Specifically, the row index array *rowInd* stores the index of each row in the bucket. When a very long row is split into multiple rows within the bucket, these are referred to as *folded rows*, and they are assigned the same row index in the corresponding locations of the *rowInd* array.

For partitions containing long rows, the maximum bucket width can be set to be smaller than the length of the longest row (still constrained to powers of 2), allowing these long rows to be placed in relatively smaller buckets, as shown in Figure 5. To optimize this process, we propose a cost model to estimate the computation cost of a given bucket, along with a search algorithm that explores the space for the optimal bucket width based on the cost model.

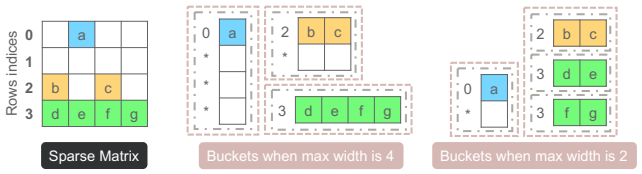

Figure 5: An illustration of how the maximum bucket width influences the distribution of non-zero elements. A larger maximum bucket width could have fewer row index accesses and coarser-grained computation workloads, while a smaller one could have less zero padding.

Cost Model. The proposed cost model estimates the computation cost of a given bucket by considering the volume of global memory loading and storing, as memory access overhead generally dominates overall GPU computation. The configuration of bucket widths plays a critical role in determining how rows are distributed among buckets, which directly affects the number of memory accesses required.

Consider the Sparse Matrix-Matrix Multiplication (SpMM) operation that is expressed as $C_{ij} = A_{ik} \cdot B_{kj}$ where A_{ik} is a sparse matrix and C_{ij}, B_{kj} are dense matrices. The computation is formulated as:

$$C[i, j] = \sum_k A[i, k] \cdot B[k, j] \tag{3}$$

where $0 \leq i < I, 0 \leq j < J, 0 \leq k < K$ with I, J, and K denoting the dimensions of the matrices. When using the CELL format, each element $A[i, k]$ in the matrix A is represented not only by its value but also by its corresponding column index $Ind[i, w]$, where w refers to the position of the index k within the structure, rather than its actual column index. Consequently, the SpMM operation is reformulated as:

$$C[i, j] = \sum_k A[i, k] \cdot B[Ind[i, w], j]. \tag{4}$$

Based on the definition of the computation, the cost model for the SpMM task, in terms of a single bucket x, can be defined as the total cost of performing the matrix multiplication for all non-zero elements within the bucket.

$$cost(x) = cost^{(1)}(x) + cost^{(2)}(x) + cost^{(3)}(x) \tag{5}$$

$$= 2 \cdot I^{(1)} W + |set(Ind[i, w])| J + Atomic \cdot I^{(2)} J. \tag{6}$$

Here, $I^{(1)}$ represents the number of rows in matrix A within this bucket when using the CELL format, and $I^{(2)}$ denotes the number of rows in matrix C that are computed by this bucket. The first part, $cost^{(1)}(x)$, represents the cost of accessing the column indices and values of the non-zero elements in matrix A. It is important to note that $I^{(1)}$ and $I^{(2)}$ are not necessarily equal, since rows with a high number of non-zero elements may be split into multiple rows within the bucket (referred to as *folded rows*), with the same corresponding row indices pointing to the output matrix C. Here, W denotes the bucket width. The second part, $cost^{(2)}(x)$, which is equal to $set(Ind[i, w])$, represents the set of unique column indices of non-zero elements in this bucket. Thus, it reflects the cost of accessing the matrix B when performing SpMM for the given

Algorithm 3: Build buckets for a given partition.

```
   Input: A column partition P of the matrix
   Output: Buckets of the partition P
1  Function BuildBuckets(P):
2      buckets = initialize buckets according to row lengths in
         P;
3      rW = max width in buckets ; // the right boundary
4      lW = 1 ;                    // the left boundary
5      while lW < rW do
6          mW = (lW + rW)/2 ;              // the middle
7          mBuckets = TuneWidth(buckets, mW);
8          mCost = GetAllCost(mBuckets);
9          mBuckets2 = TuneWidth(buckets, 2 × mW);
10         mCost2 = GetAllCost(mBuckets2);
11         if mCost > mCost2 then
               /* If the mW's cost is greater than
                  the next setting 2 × mW's cost, the
                  better setting is to the right    */
12             lW = 2 × mW;
13         else
               /* The better setting is to the left or
                  at rW                             */
14             rW = mW;
15     fBuckets = TuneWidth(buckets, lW);
16     return fBuckets;
```

bucket. The third part, $cost^{(3)}(x)$, represents the cost of writing the computed results to the output matrix C. The *Atomic* represents the weight of the atomic operation, which becomes necessary when multiple GPU threads attempt to update the same memory address simultaneously. This atomic operation ensures the correctness by serializing access to prevent data corruption.

Atomic operations (shown as line 12 in Algorithm 2) are required when there is more than one column partition in the matrix A or when the bucket is the last (maximum) bucket within its partition (shown as the condition in line 9). In cases where multiple partitions exist, non-zero elements from the same row in A may be processed by different GPU threads. Similarly, the last bucket might contain folded rows that are handled by different threads but correspond to the same rows in the output matrix C. In both scenarios, atomic operations ensure the correctness of computation by serializing access to shared memory. However, these operations introduce additional memory access overhead due to the additional memory transactions required [37]. To account for this overhead, we define $Atomic = \frac{I^{(1)}}{I^{(2)}}$, representing the average number of memory accesses compared to a single normal write operation. This results in the following cost model:

$$cost(x) = 2 \cdot I^{(1)} W + |set(Ind[i, w])| J + I^{(1)} J. \tag{7}$$

Search for Optimal Bucket Width. Based on the cost model and the design of the CELL format, the bucket width can be adjusted by either double or half. When the width is doubled, the number of rows in the bucket $I^{(1)}$ decreases, causing $cost^{(2)}$ to increase while $cost^{(3)}$ decreases. In contrast, when the width is halved, these factors respond in the opposite direction. There is supposed to exist an optimal bucket width that is the trade-off between $cost^{(2)}$ and $cost^{(3)}$, where any width smaller than or larger than this will lead to higher overall costs.

For the optimal bucket width for a partition, we propose the Algorithm 3, which leverages the trend of the cost relevant to the bucket width. Initially, the buckets are constructed by dividing the rows according to their length, where the rows with length l satisfying $2^{i-1} < l \leq 2^i$ are placed in a bucket with width 2^i (line 2). The possible maximum bucket width is set as the current maximum width of the initial buckets (line 3), and the lower boundary is set to 1 (line 4). The algorithm then functions like a binary search, calculating the middle width and its associated cost, and updating the lower or upper boundary accordingly (line 6-14). The function `TuneWidth()` adjusts the maximum width of the current buckets; `GetAllCost()` returns the total cost of all current buckets. If the cost of the middle width mW is higher than the cost of $2 \times mW$ (line 11), the optimal width should be to the right, otherwise to the left or at mW. In the end, lW points to the optimal width, and the algorithm returns the buckets built with this width (line 16).

6 IMPLEMENTATION

We built LiteForm on the SparseTIR framework [63] and use TVM [8] to generate kernel code. In the CELL format, the blocks in a bucket are mapped to GPU thread blocks by TVM's `split` and `bind` primitives. SparseTIR would emits multiple CUDA kernels and insert a horizontal fusion [16, 32] pass to the TVM backend to reduce extra kernel-launching overhead on the GPU. The powerful abstractions and tools of TVM facilitate the implementation of new kernels in LiteForm. Additionally, we leveraged scikit-learn [41] to construct the machine learning models. After evaluating various models (details in Section 7), LiteForm adopted Random Forrest model to assess the performance and optimize partitions for CELL format.

7 EVALUATION

This section provides a comprehensive performance evaluation of LiteForm. First, we evaluate the performance benefits of the automatically constructed CELL format in comparison to alternative methods for SpMM. Next, we analyze the overhead associated with the automatic format construction process. Finally, we present the results of the evaluation and selection of the most effective machine learning models to predict both the performance improvements of the CELL format and the optimal configuration parameters, including the number of partitions and bucket widths.

Our experiments were conducted on a system equipped with Intel Xeon E5-2698 20-core processors, an NVIDIA V100-SXM2-16GB GPU, 500 GB of RAM, running Ubuntu 22.04.4, and CUDA 12.2. We compared our method with several baseline approaches, including:

- cuSPARSE [38]: An NVIDIA standard library that provides high-performance implementations of common sparse operations for basic linear algebra subroutines.

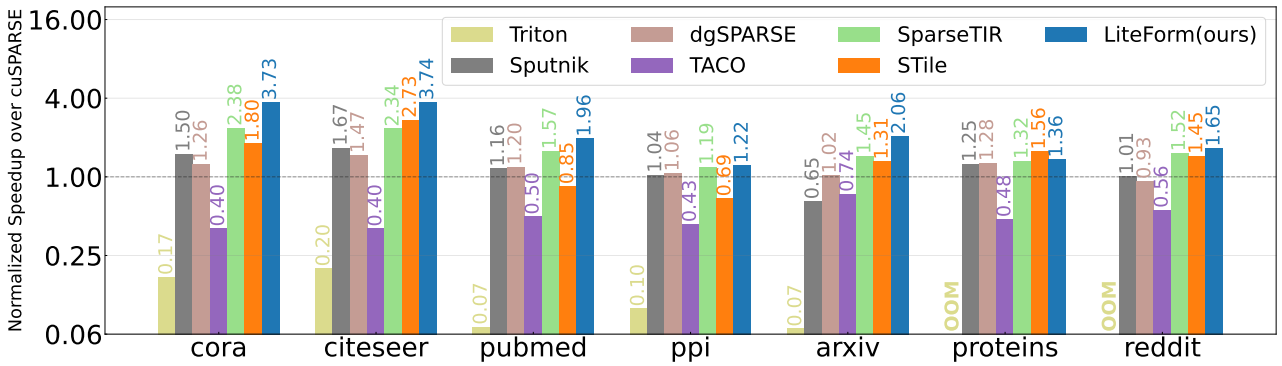

Figure 6: Normalized speedup relative to cuSPARSE for SpMM.

- Triton [51]: A language and compiler for writing highly efficient custom deep learning primitives using a tiling-based intermediate representation (IR). We utilized its block sparse implementation.
- Sputnik [18]: A library of sparse linear algebra kernels and utilities designed for deep learning applications.
- dgSPARSE [12]: A high-performance library optimized for accelerating sparse kernels on GPUs, tailored for machine learning applications such as GNNs.
- TACO (Tensor Algebra Compiler) [30]: A compiler that generates efficient code for computations involving both sparse and dense linear and tensor algebra.
- SparseTIR [63] and STile [15]: State-of-the-art frameworks that support composable formats for sparse computation.

The input sparse matrices used in our experiments are detailed in Table 4. The first seven matrices are widely used for testing operations in graph neural networks (GNNs). The remaining 1,351 matrices, are drawn from the SuiteSparse Matrix Collection [11] that has at least 2,000 nodes and are selected to cover a broad spectrum of sparsity patterns and application scenarios. This diverse selection ensures that our approach is thoroughly tested across various real-world conditions.

7.1 Performance Evaluation

We evaluated the performance of our framework in comparison to alternative methods on SpMM, using representative sparse inputs from the sparse deep learning domain. The performance is measured as the execution time of the kernel, not including the overhead of format construction that will be evaluated in the following subsections. Figure 6 displays the normalized speedup relative to cuSPARSE. Each bar represents the geometric mean of the speedup achieved for SpMM, with the number of columns in the input dense matrix varying over 32, 64, 128, 256, and 512. The label *OOM* indicates the cases in which the experiment encountered an out-of-memory error. LiteForm achieved speedups ranging from 1.22× to 3.73× over cuSPARSE, with a geometric mean speedup of 2.06×. The most closely related works to ours are SparseTIR and STile, which report geometric mean speedups of 1.63× and 1.36×, respectively, over cuSPARSE. In comparison, LiteForm obtained 1.26× and 1.52× geometric mean speedup over SparseTIR and STile, respectively. In addition, Triton, Sputnik, dgSPARSE, and TACO show 0.11×, 1.14×, 1.16×, and 0.49× geometric mean speedups over cuSPARSE, respectively. Meanwhile, LiteForm demonstrated 18.72×, 1.81×, 1.77×, and 4.18× geometric mean speedup in compariton to Triton, Sputnik, dgSPARSE, and TACO, respectively. Here Triton uses the BSR format, while cuSPARSE, Sputnik, dgSPARSE, and TACO use CSR format. For setting its scheduling, TACO took the combination of six different numbers of non-zeros per warp and six different numbers of warps per thread block, i.e. 36 different schedules in total, to run the kernel and used the shortest time as its final performance.

Table 4: Sparse matrices information

Graph	#nodes	#edges	Density
cora [45]	2,708	10,556	1.44E-03
citeseer [45]	3,327	9,228	8.34E-04
pubmed [45]	19,717	88,651	2.28E-04
ppi [20]	44,906	1,271,274	6.30E-04
arxiv [22]	169,343	1,166,243	4.07E-05
proteins [22]	132,534	39,561,252	2.25E-03
reddit [20]	232,965	114,615,892	2.11E-03
SuiteSparse [11]	2.0K–3.8M	3.1K–300.9M	8.7E-07–0.1

The key reason why LiteForm significantly outperforms SparseTIR and STile is its ability to accommodate varying sparsity in input data, which has a substantial impact on GPU performance. LiteForm improves data locality and memory utilization by dynamically adjusting bucket sizes based on the density of different sections of the matrix, assigning different sizes of the buckets to each partition. This approach reduces the amount of zero padding, minimizing both memory waste and computational overhead. Consequently, fewer unnecessary computations are performed on padded zeros, leading to a significant improvement in overall efficiency.

We evaluated LiteForm's performance against SparseTIR using 1,351 matrices with at least 2,000 nodes and varying sparsity patterns from the SuiteSparse matrix collection [11]. Figure 7 illustrates the normalized speedup of LiteForm relative to SparseTIR in which SparseTIR employed an extensive auto-tuning technique to determine the optimal configuration for its composable format. Across these matrices, LiteForm achieved a geometric mean performance

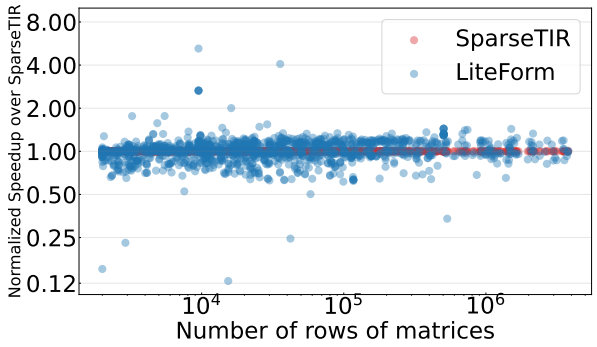

Figure 7: Normalized speedup of LiteForm relative to optimal-tuned SparseTIR using 1,351 matrices with at least 2,000 nodes and varying sparsity patterns from the SuiteSparse matrix collection.

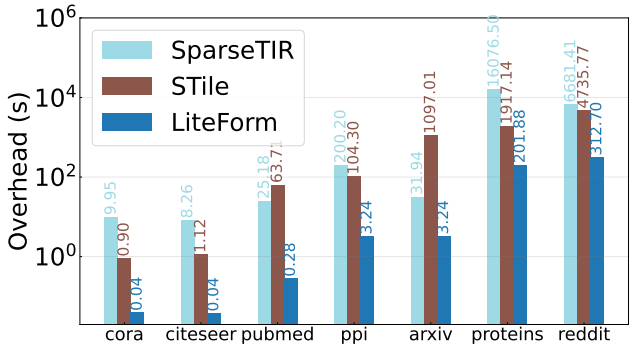

Figure 8: Overhead comparison: SparseTIR's auto-tuning process, STile's format searching mechanism, and LiteForm's format construction process.

of 0.99× that of optimal-tuned SparseTIR, with speedups ranging from 0.19× to 5.21×. The CELL format is designed to handle diverse sparsity patterns across different sections of a matrix, making it particularly useful in domains like machine learning, scientific simulations, and graph processing. However, it may not always be advantageous to use CELL in cases where the input matrix exhibits a uniform or highly regular sparsity pattern, such as densely structured blocks.

7.2 Format Construction Overhead

For sparse inputs in the sparse deep learning domain, we evaluated the overhead of sparse format construction for SparseTIR, STile, and LiteForm. The results are shown in Figure 8. Overall, the overhead introduced by LiteForm is an order of magnitude lower than the other two methods, with SparseTIR and STile experiencing geometric mean overhead 65.5× and 42.3×, respectively. SparseTIR requires an extensive auto-tuning process to find the optimal format composition, while the overhead in STile stems from executing microbenchmarks to search for the best format composition for each matrix. In contrast, LiteForm's overhead primarily comes from three sources: 1) the inference cost of the ML model to determine if the CELL format outperforms a fixed blockwise format, 2) the inference cost for predicting the optimal number of partitions, and 3) the search cost for determining the optimal bucket width. We

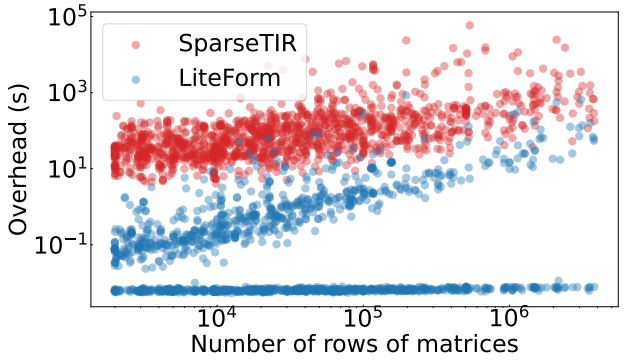

Figure 9: Overhead comparison: SparseTIR's auto-tuning process and LiteForm's format construction process.

argue that the time spent generating training data and training ML models can be amortized over future executions.

Figure 9 shows the overhead comparison using SuiteSparse matrices between SparseTIR and LiteForm. In most cases, SparseTIR's auto-tuning overhead is orders of magnitude larger than LiteForm's inference and searching overhead. Overall, the geometric mean ratio of SparseTIR's overhead to LiteForm's is 1150.2×.

Table 5: Overhead and accuracy of the tested ML models for predicting performance improvement of CELL format.

name	training(s)	inference(s)	accuracy	precision	recall	f1
Random Forest	0.2859	0.0079	88.92%	88.92%	88.92%	88.92%
KNeighbors	0.0024	0.0127	79.31%	79.31%	79.31%	79.31%
Linear SVM	0.0849	0.0098	67.00%	67.00%	67.00%	67.00%
RBF SVM	0.0856	0.0199	73.40%	73.40%	73.40%	73.40%
Gaussian Process	346.2509	0.0697	84.24%	84.24%	84.24%	84.24%
Decision Tree	0.0292	0.0004	85.96%	85.96%	85.96%	85.96%
Neural Net	2.8343	0.0016	66.50%	66.50%	66.50%	66.50%
AdaBoost	0.1828	0.0079	86.45%	86.45%	86.45%	86.45%
Naive Bayes	0.0018	0.0004	63.30%	63.30%	63.30%	63.30%
QDA	0.0022	0.0004	66.75%	66.75%	66.75%	66.75%

Table 6: Overhead and accuracy of the tested ML models for predicting the optimal number of partitions in the CELL format. *cos_sim* stands for cosine similarity.

name	training(s)	inference(s)	accuracy	precision	recall	f1	cos_sim
Random Forest	0.4778	0.0127	87.30%	87.30%	87.30%	87.30%	0.77
KNeighbors	0.0046	0.0321	82.98%	82.98%	82.98%	82.98%	0.23
Linear SVM	0.2273	0.0244	82.45%	82.45%	82.45%	82.45%	0.25
RBF SVM	0.5688	0.0692	82.56%	82.56%	82.56%	82.56%	0.25
Gaussian Process	1481.1395	24.0115	82.56%	82.56%	82.56%	82.56%	0.25
Decision Tree	0.0200	0.0005	85.40%	85.40%	85.40%	85.40%	0.77
Neural Net	3.0432	0.0017	82.45%	82.45%	82.45%	82.45%	0.25
AdaBoost	0.1952	0.0106	82.13%	82.13%	82.13%	82.13%	0.25
Naive Bayes	0.0025	0.0008	56.41%	56.41%	56.41%	56.41%	0.29
QDA	0.0036	0.0011	0.21%	0.21%	0.21%	0.21%	0.25

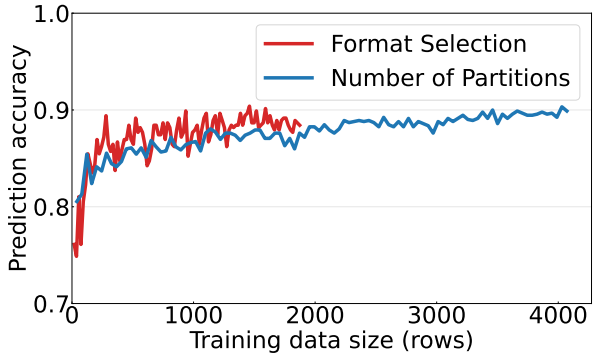

Figure 10: Impact of training set size on prediction accuracy for CELL format selection and optimal partitions.

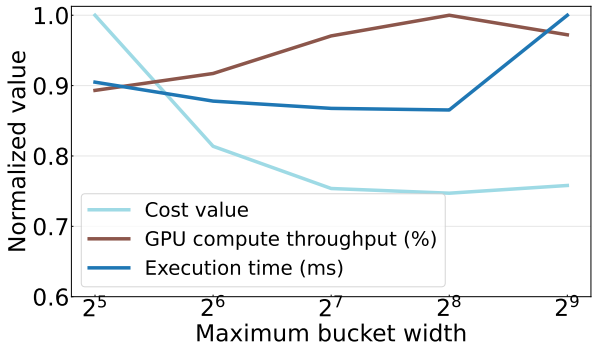

Figure 11: The reflection from the cost model value to the GPU compute throughput and execution time. The bucket width incluences the cost value. When cost value is the lowest, the GPU compute throughtput reaches the highest and the execution time is the shortest. All value are normalized to put on the same y-axis scale.

7.3 Prediction Performance

To identify the most suitable machine learning models to assess the performance benefit of the CELL format and predict the optimal number of partitions, we experimented with 10 commonly used classification models [44]. Table 5 lists the classifiers tested and presents their performance in terms of both overhead and accuracy when predicting whether the CELL format provides a performance improvement for a given input. To test these models, we used 80% of 514 matrices from the SuiteSparse collection as the training set and the remaining 20% as the test set. Among the classifiers tested, Random Forest achieved the best accuracy of 88.92% with a training time of only 0.28 seconds. Consequently, we selected Random Forest as the format selection model due to its ability to provide the highest prediction accuracy while maintaining relatively low training overhead.

We also evaluated the same models to predict the optimal number of partitions in the CELL format. Table 6 presents the overhead and accuracy of these models. During model evaluation, we considered not only accuracy but also similarity metrics to select the most suitable model for our objective. Using a metric like cosine similarity proved particularly advantageous in predicting the optimal number of partitions. Although exact predictions are preferred, it is equally important that incorrect predictions remain close to the groundtruth. By applying cosine similarity, the model prioritized predictions near the correct value, ensuring that even slight deviations resulted in minimal performance loss. Based on these criteria, we selected the Random Forest model for use in LiteForm, as it achieved a high accuracy of 87.30% with a training time of 0.4748 seconds and demonstrated a strong cosine similarity to the groundtruth.

To demonstrate the relationship between prediction accuracy and the training set size, we evaluated the final selected ML models using progressively larger datasets. As shown in Figure 10, the Random Forest model achieved accuracy over 80% even with just a few hundred data points. As the size of the training data increased, the model's accuracy approached 90%, highlighting the significant impact of larger training sets on improving the model's accuracy.

7.4 Cost Model Performance

To evaluate the effectiveness of the cost model in predicting kernel performance, we tested the *reddit* data set with various bucket configurations. Figure 11 illustrates how the maximum bucket width impacts the cost model value, GPU compute throughput [39], and execution time, with all values normalized for comparison on the same y axis. The results show that when the maximum bucket width is set to 2^8, the cost value is minimized, GPU throughput reaches its peak, and execution time is shortest. This alignment between the cost model and the actual performance confirms that the cost model accurately reflects the kernel's efficiency. Consequently, in Algorithm 3, the bucket configuration that produces the lowest cost would deliver optimal performance.

8 LIMITATIONS

LiteForm is designed to provide lightweight, automatic selection and composition of sparse formats with minimal overhead. Currently, the framework has some limitations. First, LiteForm needs historical performance data to configure the CELL format optimally, which may cost hours, and it requires model retraining for new architectures or kernels. Using transfer learning [56] in the future may reduce the necessity of retraining from scratch. Second, LiteForm's performance depends on historical data, which may not cover scenarios like a large number of partitions or extremely wide buckets. Expanding the data set or using generalized prediction techniques could address this problem. Lastly, LiteForm currently does not utilize Tensor Cores on GPUs to further boost performance. Incorporating Tensor Cores support in the future could unlock additional performance improvements, particularly in deep learning workloads.

9 RELATED WORKS

In addition to the ones mentioned in Section 2.2, there are other works on optimizing SpMM and other sparse computations on GPUs, which utilize various optimization techniques.

ML-guided Prediction. There are some works that also use ML models to guide the optimization of sparse computations. DA-SpMM [10] uses the gradient boosting framework LightGBM [28] to select the best loop designs for SpMM. DDB [65] uses an ML model to find optimal SpMM strategies not on GPUs but on hardware with dedicated matrix-multiply units. WISE [64] is an ML framework that predicts the performance of different sparse matrix-vector multiplication (SpMV) methods.

Blocking and Tiling. Blocking and tiling partition the computation and memory access of non-zero elements for better data reuse and load balancing. 1D-VBR [1] uses dynamic row groups in a fixed column grouping (with single columns) to partition the matrix into blocks of one column. It then uses a micro-benchmark to configure the block size. AspT [21] divides a sparse matrix in the CSR format into row panels. Within each row panel, column segments are classified as sufficiently dense or not according to a threshold determined by micro-benchmarking using synthetic matrices. SparseRT [55] aims to accelerate deep learning inference on GPUs. For SpMM, it first treats a sparse matrix as a dense matrix and then explores different tiling strategies. For each of those tiling strategies, SparseRT rebalances and removes computations that involve zeros in the original sparse matrix to obtain an optimized executor code. Yang et al. [62] propose hybrid blocking strategies for SpMV to adapt to various data localities. Anzt et al. [3] uses a modified sliced Ellpack format which blocks a set of vectors and processes them simultaneously. Yang et al. [61] extend two tiling strategies (row-splitting and merge-based) in SpMV for the SpMM.

Workload Balancing. Merrill and Garland [34] adapt the fine-grained MergePath decomposition [40] to merge two sorted lists, allowing them to distribute balanced workloads among the processing elements for SpMV. HP-SpMM [14] ensures that the non-zero elements for each warp belong to the same row as much as possible while achieving load balancing to reduce global memory accesses for GNN training. GNNAdvisor [54] proposes partitioning and thread alignment methods for workload management to reduce inter-node workload imbalance and redundant atomic operations. Anzt et al. [2] compare four sparse representation formats with different work balancing strategies for SpMV. In addition, there are other optimization techniques including reordering [27], thread coarsening [23], and vectorization [19].

10 CONCLUSION

This paper presents LiteForm, a lightweight framework for automatic composition of the CELL format, a flexible three-level blockwise structure to optimize SpMM on GPUs. The CELL format organizes non-zero elements into column partitions, row buckets, and non-zero blocks, adapting to matrix sparsity patterns to enhance memory efficiency and load balancing. The automatic composition of the CELL format is driven by machine learning models and a cost model. Pretrained ML models determine optimal partitions based on matrix density and size, while the cost model adjusts bucket widths by analyzing memory access and overhead, ensuring efficient block distribution across GPU threads. By merging predictive and cost-driven strategies, LiteForm automates the configuration of the CELL format, improving performance across diverse matrix structures and minimizing manual tuning. Thus, LiteForm is an effective solution for optimizing sparse matrix operations in various workloads. Our SpMM experiments show that LiteForm achieves a geometric mean speedup of 2.06×, 1.81×, 1.77×, and 4.18× over cuSPARSE, Sputnik, dgSPARSE, and TACO, respectively, and speedups of 1.26× and 1.52× compared to state-of-the-art SparseTIR and STile, respectively. In the future work, we plan to extend LiteForm's design to various sparse computational kernels and even for multiple GPUs.

ACKNOWLEDGMENTS

The authors thank anonymous reviewers for their constructive comments and informative suggestions. This work is supported by the U.S. Department of Energy (DOE), Office of Science, Advanced Scientific Computing Research (ASCR) program under the project "81461 – Compiler Frameworks and Hardware Generators" at Pacific Northwest National Laboratory (PNNL). PNNL is a multi-program national laboratory operated for the U.S. Department of Energy (DOE) by Battelle Memorial Institute under Contract No. DE-AC05-76RL01830.

REFERENCES

[1] Willow Ahrens and Erik G Boman. 2020. On Optimal Partitioning For Sparse Matrices In Variable Block Row Format. *arXiv preprint arXiv:2005.12414* (2020).
[2] Hartwig Anzt, Terry Cojean, Chen Yen-Chen, Jack Dongarra, Goran Flegar, Pratik Nayak, Stanimire Tomov, Yuhsiang M Tsai, and Weichung Wang. 2020. Load-Balancing Sparse Matrix Vector Product Kernels on GPUs. *ACM Transactions on Parallel Computing (TOPC)* 7, 1 (2020), 1–26.
[3] Hartwig Anzt, Stanimire Tomov, and Jack J Dongarra. 2015. Accelerating the LOBPCG Method on GPUs Using a Blocked Sparse Matrix Vector Product. In *SpringSim (HPS)*. 75–82.
[4] Mina Ashoury, Mohammad Loni, Farshad Khunjush, and Masoud Daneshtalab. 2023. Auto-SpMV: Automated Optimizing SpMV Kernels on GPU. arXiv:2302.05662 [cs.DC]
[5] Per Block, Marion Hoffman, Isabel J Raabe, Jennifer Beam Dowd, Charles Rahal, Ridhi Kashyap, and Melinda C Mills. 2020. Social Network-Based Distancing Strategies to Flatten the COVID-19 Curve in a Post-Lockdown World. *Nature human behaviour* 4, 6 (2020), 588–596.
[6] Aydin Buluc and John R Gilbert. 2008. On the Representation and Multiplication of Hypersparse Matrices. In *2008 IEEE International Symposium on Parallel and Distributed Processing*. IEEE, 1–11.
[7] Stefano Carrazza, Juan Cruz-Martinez, Marco Rossi, and Marco Zaro. 2021. MadFlow: Automating Monte Carlo Simulation on GPU for Particle Physics Processes. *The European Physical Journal C* 81 (2021), 1–7.
[8] Tianqi Chen, Thierry Moreau, Ziheng Jiang, Lianmin Zheng, Eddie Yan, Haichen Shen, Meghan Cowan, Leyuan Wang, Yuwei Hu, Luis Ceze, et al. 2018. {TVM}: An Automated {End-to-End} Optimizing Compiler for Deep Learning. In *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)*. 578–594.
[9] Jee W. Choi, Amik Singh, and Richard W. Vuduc. 2010. Model-Driven Autotuning of Sparse Matrix-Vector Multiply on GPUs. In *Proceedings of the 15th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP 2010)* (Bangalore, India) *(PPoPP '10)*. Association for Computing Machinery, New York, NY, USA, 115–126. https://doi.org/10.1145/1693453.1693471
[10] Guohao Dai, Guyue Huang, Shang Yang, Zhongming Yu, Hengrui Zhang, Yufei Ding, Yuan Xie, Huazhong Yang, and Yu Wang. 2022. Heuristic Adaptability to Input Dynamics for Spmm on GPUs. In *Proceedings of the 59th ACM/IEEE Design Automation Conference (DAC)*. 595–600.
[11] Timothy A Davis and Yifan Hu. 2011. The University of Florida Sparse Matrix Collection. *ACM Transactions on Mathematical Software (TOMS)* 38, 1 (2011), 1–25.
[12] dgSPARSE. 2024. dgSPARSE. https://dgsparse.github.io/. Accessed: September 12, 2024.
[13] Zhen Du, Jiajia Li, Yinshan Wang, Xueqi Li, Guangming Tan, and Ninghui Sun. 2022. AlphaSparse: Generating High Performance Spmv Codes Directly From Sparse Matrices. In *SC22: International Conference for High Performance Computing, Networking, Storage and Analysis*. IEEE, 1–15.
[14] Ruibo Fan, Wei Wang, and Xiaowen Chu. 2023. Fast Sparse Gpu Kernels for Accelerated Training of Graph Neural Networks. In *2023 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*. IEEE, 501–511.

[15] Jingzhi Fang, Yanyan Shen, Yue Wang, and Lei Chen. 2024. STile: Searching Hybrid Sparse Formats for Sparse Deep Learning Operators Automatically. *Proceedings of the ACM on Management of Data* 2, 1, Article 68 (mar 2024), 26 pages. https://doi.org/10.1145/3639323

[16] Pratik Fegade, Tianqi Chen, Phillip Gibbons, and Todd Mowry. 2022. The CoRa Tensor Compiler: Compilation for Ragged Tensors With Minimal Padding. *Proceedings of Machine Learning and Systems* 4 (2022), 721–747.

[17] Chenjiao Feng, Jiye Liang, Peng Song, and Zhiqiang Wang. 2020. A Fusion Collaborative Filtering Method for Sparse Data in Recommender Systems. *Information Sciences* 521 (2020), 365–379.

[18] Trevor Gale, Matei Zaharia, Cliff Young, and Erich Elsen. 2020. Sparse Gpu Kernels for Deep Learning. In *SC20: International Conference for High Performance Computing, Networking, Storage and Analysis*. IEEE, 1–14.

[19] Joseph L Greathouse and Mayank Daga. 2014. Efficient Sparse Matrix-Vector Multiplication on GPUs Using the CSR Storage Format. In *SC'14: Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*. IEEE, 769–780.

[20] Will Hamilton, Zhitao Ying, and Jure Leskovec. 2017. Inductive Representation Learning on Large Graphs. *Advances in Neural Information Processing Systems* 30 (2017).

[21] Changwan Hong, Aravind Sukumaran-Rajam, Israt Nisa, Kunal Singh, and P Sadayappan. 2019. Adaptive Sparse Tiling for Sparse Matrix Multiplication. In *Proceedings of the 24th Symposium on Principles and Practice of Parallel Programming (PPoPP 2019)*. 300–314.

[22] Weihua Hu, Matthias Fey, Marinka Zitnik, Yuxiao Dong, Hongyu Ren, Bowen Liu, Michele Catasta, and Jure Leskovec. 2020. Open Graph Benchmark: Datasets for Machine Learning on Graphs. *Advances in Neural Information Processing Systems* 33 (2020), 22118–22133.

[23] Guyue Huang, Guohao Dai, Yu Wang, and Huazhong Yang. 2020. GE-SpMM: General-Purpose Sparse Matrix-Matrix Multiplication on GPUs for Graph Neural Networks. In *SC20: International Conference for High Performance Computing, Networking, Storage and Analysis*. 1–12. https://doi.org/10.1109/SC41405.2020.00076

[24] Eun-Jin Im. 2000. *Optimizing the Performance of Sparse Matrix-Vector Multiplication*. University of California, Berkeley.

[25] Eun-Jin Im and Katherine Yelick. 1998. Model-Based Memory Hierarchy Optimizations for Sparse Matrices. In *Workshop on Profile and Feedback-Directed Compilation*, Vol. 139. Citeseer.

[26] Intel. 2024. Intel oneAPI Math Kernel Library (oneMKL). https://www.intel.com/content/www/us/en/developer/tools/oneapi/onemkl.html. Accessed: August 29, 2024.

[27] Peng Jiang, Changwan Hong, and Gagan Agrawal. 2020. A Novel Data Transformation and Execution Strategy for Accelerating Sparse Matrix Multiplication on GPUs. In *Proceedings of the 25th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP)*. 376–388.

[28] Guolin Ke, Qi Meng, Thomas Finley, Taifeng Wang, Wei Chen, Weidong Ma, Qiwei Ye, and Tie-Yan Liu. 2017. LightGBM: A Highly Efficient Gradient Boosting Decision Tree. In *Advances in Neural Information Processing Systems (NeurIPS)*, I. Guyon, U. Von Luxburg, S. Bengio, H. Wallach, R. Fergus, S. Vishwanathan, and R. Garnett (Eds.), Vol. 30. Curran Associates, Inc. https://proceedings.neurips.cc/paper_files/paper/2017/file/6449f44a102fde848669bdd9eb6b76fa-Paper.pdf

[29] David R. Kincaid, John R. Respess, David M. Young, and Rober R. Grimes. 1982. Algorithm 586: ITPACK 2C: A FORTRAN Package for Solving Large Sparse Linear Systems by Adaptive Accelerated Iterative Methods. *ACM Trans. Math. Software* 8, 3 (sep 1982), 302–322. https://doi.org/10.1145/356004.356009

[30] Fredrik Kjolstad, Shoaib Kamil, Stephen Chou, David Lugato, and Saman Amarasinghe. 2017. The Tensor Algebra Compiler. *Proceedings of the ACM on Programming Languages* 1, OOPSLA (2017), 1–29.

[31] Yann LeCun, Yoshua Bengio, and Geoffrey Hinton. 2015. Deep learning. *Nature* 521, 7553 (2015), 436–444.

[32] Ao Li, Bojian Zheng, Gennady Pekhimenko, and Fan Long. 2022. Automatic Horizontal Fusion for GPU Kernels. In *2022 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*. IEEE, 14–27.

[33] Jiajia Li, Jimeng Sun, and Richard Vuduc. 2018. HiCOO: Hierarchical Storage of Sparse Tensors. In *SC18: International Conference for High Performance Computing, Networking, Storage and Analysis*. IEEE, 238–252.

[34] Duane Merrill and Michael Garland. 2016. Merge-Based Sparse Matrix-Vector Multiplication (SpMV) Using the CSR Storage Format. *ACM Sigplan Notices* 51, 8 (2016), 1–2.

[35] Alexander Monakov, Anton Lokhmotov, and Arutyun Avetisyan. 2010. Automatically Tuning Sparse Matrix-Vector Multiplication for GPU Architectures. In *High Performance Embedded Architectures and Compilers (HiPEAC 2010)*, Yale N. Patt, Pierfrancesco Foglia, Evelyn Duesterwald, Paolo Faraboschi, and Xavier Martorell (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 111–125.

[36] Yuyao Niu, Zhengyang Lu, Meichen Dong, Zhou Jin, Weifeng Liu, and Guangming Tan. 2021. TileSpMV: A Tiled Algorithm for Sparse Matrix-Vector Multiplication on GPUs. In *2021 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*. IEEE, 68–78.

[37] NVIDIA. 2024. CUDA C++ Programming Guide, 7.14. Atomic Functions. https://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html. Accessed: 2024-05-22.

[38] NVIDIA. 2024. cuSPARSE, The API Reference Guide For cuSPARSE, the CUDA Sparse Matrix Library. https://docs.nvidia.com/cuda/cusparse/. Accessed: June 11, 2024.

[39] NVIDIA. 2024. NSight Kernel Profiling GUide. https://docs.nvidia.com/nsight-compute/ProfilingGuide/index.html. Accessed: September 12, 2024.

[40] Saher Odeh, Oded Green, Zahi Mwassi, Oz Shmueli, and Yitzhak Birk. 2012. Merge Path-Parallel Merging Made Simple. In *2012 IEEE 26th International Parallel and Distributed Processing Symposium Workshops & PhD Forum*. IEEE, 1611–1618.

[41] F. Pedregosa, G. Varoquaux, A. Gramfort, V. Michel, B. Thirion, O. Grisel, M. Blondel, P. Prettenhofer, R. Weiss, V. Dubourg, J. Vanderplas, A. Passos, D. Cournapeau, M. Brucher, M. Perrot, and E. Duchesnay. 2011. Scikit-learn: Machine Learning in Python. *Journal of Machine Learning Research* 12 (2011), 2825–2830.

[42] Junyang Qian, Yosuke Tanigawa, Wenfei Du, Matthew Aguirre, Chris Chang, Robert Tibshirani, Manuel A Rivas, and Trevor Hastie. 2020. A Fast and Scalable Framework for Large-Scale and Ultrahigh-Dimensional Sparse Regression With Application to the UK Biobank. *PLoS Genetics* 16, 10 (2020), e1009141.

[43] Nobou Sato and WF Tinney. 1963. Techniques for Exploiting the Sparsity or the Network Admittance Matrix. *IEEE Transactions on Power Apparatus and Systems* 82, 69 (1963), 944–950.

[44] scikit learn. 2024. Classifier Comparison. https://scikit-learn.org/stable/auto_examples/classification/plot_classifier_comparison.html. Accessed: June 11, 2024.

[45] Prithviraj Sen, Galileo Namata, Mustafa Bilgic, Lise Getoor, Brian Galligher, and Tina Eliassi-Rad. 2008. Collective Classification in Network Data. *AI Magazine* 29, 3 (2008), 93–93.

[46] Julian Shun and Guy E. Blelloch. 2013. Ligra: a lightweight graph processing framework for shared memory. In *Proceedings of the 18th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP '13)* (Shenzhen, China) *(PPoPP '13)*. Association for Computing Machinery, New York, NY, USA, 135–146. https://doi.org/10.1145/2442516.2442530

[47] Shaden Smith and George Karypis. 2015. Tensor-Matrix Products With a Compressed Sparse Tensor. In *Proceedings of the 5th Workshop on Irregular Applications: Architectures and Algorithms*. 1–7.

[48] C. Stylianou and M. Weiland. 2023. Optimizing Sparse Linear Algebra Through Automatic Format Selection and Machine Learning. In *2023 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*. IEEE Computer Society, Los Alamitos, CA, USA, 734–743. https://doi.org/10.1109/IPDPSW59300.2023.00125

[49] Qingxiao Sun, Yi Liu, Ming Dun, Hailong Yang, Zhongzhi Luan, Lin Gan, Guangwen Yang, and Depei Qian. 2020. SpTFS: Sparse Tensor Format Selection for MTTKRP via Deep Learning. In *SC20: International Conference for High Performance Computing, Networking, Storage and Analysis*. 1–14. https://doi.org/10.1109/SC41405.2020.00022

[50] Ryan Swann, Muhammad Osama, Karthik Sangaiah, and Jalal Mahmud. 2024. Seer: Predictive Runtime Kernel Selection for Irregular Problems. In *2024 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*. IEEE Computer Society, Los Alamitos, CA, USA, 133–142. https://doi.org/10.1109/CGO57630.2024.10444812

[51] Philippe Tillet, Hsiang-Tsung Kung, and David Cox. 2019. Triton: An Intermediate Language and Compiler for Tiled Neural Network Computations. In *Proceedings of the 3rd ACM SIGPLAN International Workshop on Machine Learning and Programming Languages (MAPL 2019)*. 10–19.

[52] William F Tinney and John W Walker. 1967. Direct Solutions of Sparse Network Equations by Optimally Ordered Triangular Factorization. *Proc. IEEE* 55, 11 (1967), 1801–1809.

[53] Qihan Wang, Zhen Peng, Bin Ren, Jie Chen, and Robert G Edwards. 2022. MemHC: An Optimized GPU Memory Management Framework for Accelerating Many-Body Correlation. *ACM Transactions on Architecture and Code Optimization (TACO)* 19, 2 (2022), 1–26.

[54] Yuke Wang, Boyuan Feng, Gushu Li, Shuangchen Li, Lei Deng, Yuan Xie, and Yufei Ding. 2021. GNNAdvisor: An Adaptive and Efficient Runtime System for GNN Acceleration on GPUs. In *15th USENIX symposium on operating systems design and implementation (OSDI 21)*. 515–531.

[55] Ziheng Wang. 2020. Sparsert: Accelerating Unstructured Sparsity on Gpus for Deep Learning Inference. In *Proceedings of the ACM International Conference on Parallel Architectures and Compilation Techniques (PACT)*. 31–42.

[56] Karl Weiss, Taghi M Khoshgoftaar, and DingDing Wang. 2016. A Survey of Transfer Learning. *Journal of Big data* 3 (2016), 1–40.

[57] Anuradha Welivita, Indika Perera, Dulani Meedeniya, Anuradha Wickramarachchi, and Vijini Mallawaarachchi. 2018. Managing Complex Workflows in Bioinformatics: An Interactive Toolkit With Gpu Acceleration. *IEEE Transactions on NanoBioscience* 17, 3 (2018), 199–208.

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

[59] Yannan Nellie Wu, Po-An Tsai, Angshuman Parashar, Vivienne Sze, and Joel S Emer. 2022. Sparseloop: An Analytical Approach to Sparse Tensor Accelerator Modeling. In *2022 55th IEEE/ACM International Symposium on Microarchitecture (MICRO)*. IEEE, 1377–1395.

[60] Zhen Xie, Guangming Tan, Weifeng Liu, and Ninghui Sun. 2019. IA-SpGEMM: an Input-Aware Auto-Tuning Framework for Parallel Sparse Matrix-Matrix Multiplication. In *Proceedings of the ACM International Conference on Supercomputing* (Phoenix, Arizona) *(ICS '19)*. Association for Computing Machinery, New York, NY, USA, 94–105. https://doi.org/10.1145/3330345.3330354

[61] Carl Yang, Aydın Buluç, and John D Owens. 2018. Design Principles for Sparse Matrix Multiplication on the Gpu. In *European Conference on Parallel Processing*. Springer, 672–687.

[62] Xintian Yang, Srinivasan Parthasarathy, and Ponnuswamy Sadayappan. 2011. Fast Sparse Matrix-Vector Multiplication on GPUs: Implications for Graph Mining. *Proceedings of the VLDB Endowment* 4, 4 (2011).

[63] Zihao Ye, Ruihang Lai, Junru Shao, Tianqi Chen, and Luis Ceze. 2023. SparseTIR: Composable Abstractions for Sparse Compilation in Deep Learning. In *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 3* (Vancouver, BC, Canada) *(ASPLOS 2023)*. Association for Computing Machinery, New York, NY, USA, 660–678. https://doi.org/10.1145/3582016.3582047

[64] Serif Yesil, Azin Heidarshenas, Adam Morrison, and Josep Torrellas. 2023. Wise: Predicting the Performance of Sparse Matrix Vector Multiplication With Machine Learning. In *Proceedings of the 28th ACM SIGPLAN Annual Symposium on Principles and Practice of Parallel Programming (PPoPP)*. 329–341.

[65] Serif Yesil, José E Moreira, and Josep Torrellas. 2022. Dense Dynamic Blocks: Optimizing SpMM for Processors With Vector and Matrix Units Using Machine Learning Techniques. In *Proceedings of the 36th ACM International Conference on Supercomputing (ICS)*. 1–14.

[66] Tuowen Zhao, Tharindu Rusira, Khalid Ahmad, and Mary Hall. 2016. A Novel Variable-Blocking Representation for Efficient Sparse Matrix-Vector Multiply on GPUs. In *SC '16: Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*.