LO-SpMM: Low-cost Search for High-performance SpMM Kernels on GPUs

JUNQING LIN, Computer Science and Technology, University of Science and Technology of China, Hefei, China
JINGWEI SUN, Computer Science and Technology, University of Science and Technology of China, Hefei, China
XIAOLONG SHI, Computer Science and Technology, University of Science and Technology of China, Hefei, China
HONGHE ZHANG, Computer Science and Technology, University of Science and Technology of China, Hefei, China
XIANZHI YU, Huawei Noah's Ark Lab, Shenzhen, China
XINZHI WANG, Huawei Noah's Ark Lab, Shenzhen, China
JUN YAO, Huawei Noah's Ark Lab, Shenzhen, China
GUANGZHONG SUN, Computer Science and Technology, University of Science and Technology of China, Hefei, China

As deep neural networks (DNNs) become increasingly large and complicated, pruning techniques are proposed for lower memory footprint and more efficient inference. The most critical kernel to execute pruned sparse DNNs on GPUs is Sparse-dense Matrix Multiplication (SpMM). To maximize the performance of SpMM, despite the high-performance implementation generated from advanced tensor compilers, they often take a long time to iteratively search tuning configurations. Such a long time slows down the cycle of exploring better DNN architectures or pruning algorithms. In this article, we propose LO-SpMM to efficiently generate

Extension of Conference Paper. Junqing Lin, Honghe Zhang, Xiaolong Shi, Jingwei Sun, Xianzhi Yu, Jun Yao, and Guangzhong Sun. 2023. EC-SpMM: Efficient Compilation of SpMM Kernel on GPUs. In 52nd International Conference on Parallel Processing (ICPP 2023), August 07-10, 2023, Holladay, UT, USA. Association for Computing Machinery, New York, NY, USA, 21-30. https://doi.org/10.1145/3605573.3605632.
We identified two sources of long time cost for generating SpMM implementations: large search space and long time for evaluating a candidate. The conference paper proposed a solution to constrain the search space. In this extension paper, we proposed a proxy-based method to simplify the evaluation and thereby further reduce time cost.
This study is supported by Youth Innovation Promotion Association CAS and Huawei Noah's Ark Lab. Experiments of this study were conducted on the Supercomputing Center of USTC.
Authors' Contact Information: Junqing Lin, Computer Science and Technology, University of Science and Technology of China, Hefei, Anhui, China; e-mail: linjunqing@mail.ustc.edu.cn; Jingwei Sun (Co-corresponding author), Computer Science and Technology, University of Science and Technology of China, Hefei, Anhui, China; e-mail: sunjw@ustc.edu.cn; Xiaolong Shi, Computer Science and Technology, University of Science and Technology of China, Hefei, Anhui, China; e-mail: shixiaolong@mail.ustc.edu.cn; Honghe Zhang, Computer Science and Technology, University of Science and Technology of China, Hefei, Anhui, China; e-mail: zhanghonghe@mail.ustc.edu.cn; Xianzhi Yu, Huawei Noah's Ark Lab, Shenzhen, Guangdong, China; e-mail: yuxianzhi@huawei.com; Xinzhi Wang, Huawei Noah's Ark Lab, Shenzhen, Guangdong, China; e-mail: wangxinzhi3@huawei.com; Jun Yao, Huawei Noah's Ark Lab, Shenzhen, Guangdong, China; e-mail: yaojun97@huawei.com; Guangzhong Sun (Co-corresponding author), Computer Science and Technology, University of Science and Technology of China, Hefei, Anhui, China; e-mail: gzsun@ustc.edu.cn.

ACM 1544-3973/2024/11-ART73
https://doi.org/10.1145/3685277

high-performance SpMM implementations for sparse DNN inference. Based on the analysis of nonzero elements' layout, the characterization of the GPU architecture, and a rank-based cost model, LO-SpMM can effectively reduce the search space and eliminate possibly low-performance candidates. Besides, rather than generating complete SpMM implementations for evaluation, LO-SpMM constructs simplified proxies to quickly estimate performance, thereby substantially reducing compilation and execution costs. Experimental results show that LO-SpMM can reduce the search time by 281× at most, while the performance of generated SpMM implementations is comparable to or better than the state-of-the-art sparse tensor compiling solutions.

CCS Concepts: • **Computing methodologies** → **Parallel algorithms**; **Neural networks;**

Additional Key Words and Phrases: Sparse-dense matrix multiplication, deep neural network, tensor compiler, GPU

ACM Reference Format:
Junqing Lin, Jingwei Sun, Xiaolong Shi, Honghe Zhang, Xianzhi Yu, Xinzhi Wang, Jun Yao, and Guangzhong Sun. 2024. LO-SpMM: Low-cost Search for High-performance SpMM Kernels on GPUs. *ACM Trans. Arch. Code Optim.* 21, 4, Article 73 (November 2024), 25 pages. https://doi.org/10.1145/3685277

1 Introduction

Along with the great accomplishments of deep learning across various domains, the scale of **deep neural networks** (**DNNs**) has grown significantly. While larger models typically exhibit better performance, they also incur higher computation and storage costs. For example, GPT-3 [23], with its 175 billion parameters, requires 350 GB memory to store (using FP16 precision) and thus requires five A100 (80 GB) GPUs for inference.

Given the over-parameterization of large DNNs [30], neural network pruning methods have been developed to reduce model size by eliminating redundant parameters with minimal impact on model accuracy. Pruning techniques can be categorized as structured pruning and unstructured pruning [8, 28, 39, 51, 59]. Structured pruning can accelerate inference through off-the-shelf GPU tensor computing libraries, but it often results in a significant accuracy drop due to its strict constraints on the position of pruned elements. Conversely, unstructured pruning can preserve higher accuracy, but the resulting sparse DNNs pose challenges in efficiently leveraging current GPU architectures, as they exhibit irregular memory accesses during inference. To execute the inference of sparse DNNs on GPUs, **Sparse-dense Matrix Multiplication** (**SpMM**) is the most critical kernel [24, 36, 63]. It is the multiplication of a sparse matrix and a dense matrix. The sparse matrix is the weights of a pruned linear/convolutional/recurrent/attention layer, while the dense matrix is the input data.

An SpMM kernel can have various implementations based on different tuning configurations. Recent studies of advanced tensor compilers [13, 68, 69] show that automatically generated SpMM implementations can achieve excellent performance and outperform hand-crafted libraries (e.g., cuSPARSE). The typical workflow of tensor compilers consists of defining a large search space for tuning configurations, iteratively selecting candidates from this space by a cost model, generating kernel implementations for evaluation, and adjusting the cost model by the evaluation results. It continues until obtaining a satisfactory kernel implementation. However, this workflow does not effectively leverage prior knowledge of the hardware architecture, SpMM algorithm, and input data layout, resulting in unnecessary search costs. Tensor compilers can take several days or weeks to tune a DNN model [70]. Such a long time slows down the cycle of exploring better DNN architectures or pruning solutions in the AutoML pipeline [10, 37].

The search cost of tensor compilers is primarily caused by two factors. **First**, the implementation of an SpMM kernel constitutes a large search space. For example, an SpMM kernel with shape

(M, N, K) has $M \times N$ possible tiling configurations to explore. **Second**, evaluating each candidate in the space takes a long time. State-of-the-art implementations of SpMM kernels adopt full loop unrolling and constant propagation [60, 69], which write all nonzero elements of the sparse matrix into the code as constants. This optimization can effectively accelerate SpMM, but it also brings huge code files and a long compilation time.

In this article, we propose a method named LO-SpMM to generate high-performance SpMM implementations and minimize the search cost. It optimizes the processes of searching for optimal SpMM implementations from three aspects:

(1) LO-SpMM conducts a few-shot search for optimal implementations. According to our observation, a large portion of candidates in the search space do not properly leverage GPU hardware, leading to suboptimal performance or even execution failure. LO-SpMM automatically identifies and eliminates these ineffective candidates, using several constraints based on the characteristics of the GPU architecture and the SpMM algorithm. In addition, a learning-based rank model is built to further reduce the remaining candidates. Consequently, the search space can be effectively shrunk.

(2) To avoid the costly compilation and execution for searched SpMM implementations, LO-SpMM constructs proxies as their approximations. A proxy is a simplified version of the original SpMM implementations, exhibiting a similar performance rank as the original. Therefore, by compiling and measuring the proxies, LO-SpMM can quickly estimate the performance rank of searched SpMM implementations.

(3) Besides focusing on the issue of long search time, LO-SpMM also tries to improve the performance of generated SpMM implementations. Existing tensor compilers primarily focus on loop reordering/tiling/unrolling optimization techniques, which are critical for GEMM (General Matrix Multiplication). In addition to these loop optimizations, LO-SpMM takes the layout of nonzero elements in sparse matrices into account. It adopts a heuristic method to reorder the sparse matrix in SpMM, enabling better memory access performance and load balancing. It also adopts a prefetching technique to reduce the latency caused by cache misses.

We evaluate LO-SpMM on typical DNN models, including ResNet, ShuffleNet, MobileNet, and BERT, on NVIDIA RTX 2080Ti and Tesla V100. Compared with the state-of-the-art tensor compilers, our approach can reduce the search time by 281× at most. Compared with cuSPARSE, TVM-S [13], Sputnik [24], SparTA [69], and EC-SpMM [38], the performance of SpMM from LO-SpMM achieves 34.70×, 29.32×, 3.00×, 1.10×, and 1.03× average speedup, respectively. We further combine LO-SpMM with SparTA and Rammer [42] to implement end-to-end sparse DNN inference, and achieve $1.01\times - 3.38\times$ speedup over baseline solutions [3, 13, 24, 42, 47, 69].

In summary, the main contributions of our article are as follows:

— We propose a method to automatically prune the search space of tiling for SpMM, based on constraints from prior knowledge of the GPU architecture and programming experience.
— We propose a method to construct proxies to replace the evaluation of candidates. The proxies have a similar performance rank as the original SpMM implementations and enable faster evaluations.
— We propose a matrix reordering method to adjust the layout of nonzero elements of the sparse matrix in SpMM. It can reduce memory access operations while keeping computational load balancing.
— We conduct extensive experiments for sparse DNNs on both consumer-level and high-end GPUs. We validate that our method can achieve comparable or better SpMM kernel performance and significantly reduce the search time, compared with existing tensor compilers.

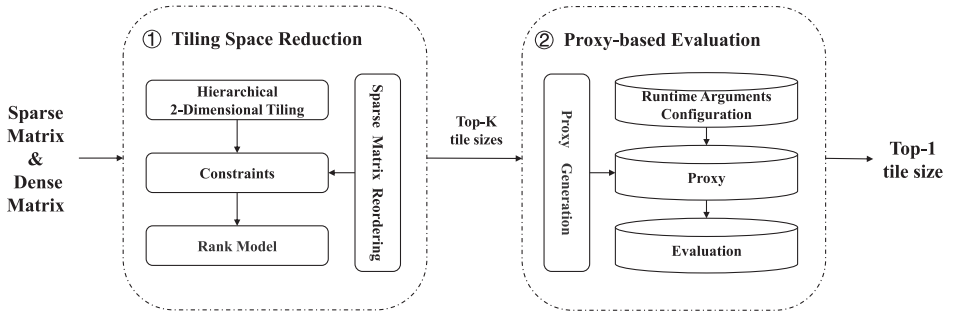


Fig. 1. Overview of LO-SpMM .

2 Overview of LO-SpMM

The overall framework of LO-SpMM is illustrated in Figure 1, which consists of two stages: tiling space reduction, and proxy-based evaluation.

Given a sparse matrix A of size $M \times K$, and a dense matrix B of size $K \times N$, LO-SpMM aims at quickly generating a high-performance SpMM implementation on a GPU for calculating $C = A \times B$.

In the first stage, LO-SpMM utilizes a hierarchical 2-dimensional tiling strategy to tile the computations in the SpMM kernel. Different tile sizes form the search space for constructing SpMM implementations. LO-SpMM prunes search space by several constraints, based on criteria such as register resource, hardware utilization, and load balancing. A sparse matrix reordering operation is performed to reduce the memory load in the SpMM kernel before the constraints. The detailed motivation and design of sparse matrix reordering are introduced in Section 4. In addition to the constraints, LO-SpMM can employ a learning-based rank model to sort the remaining candidates and select the top-K candidates for evaluation, thereby further reducing the number of evaluations.

In the second stage, LO-SpMM generates proxies for the remaining tile sizes, thereafter selecting the corresponding proxy for each tile size and configuring runtime arguments of the proxy for performance evaluation. A proxy is a simplified CUDA code of the original SpMM implementations. It has a lower evaluation cost while keeping a similar relative performance rank as the original SpMM implementations. LO-SpMM selects the Top-1 tile size with optimal proxy performance and subsequently generates code for the SpMM kernel. The details of SpMM implementations are discussed in Section 6.

3 Tile Space Reduction

To take full advantage of GPU parallelism, LO-SpMM uses a hierarchical 2-dimensional tiling strategy to divide an SpMM kernel into tiles for processing. The selection of tile size has a critical impact on the performance of the generated SpMM implementation. Since the search space of tile size is vast, an exhaustive search to determine the optimal implementation among all possible sizes incurs significant overhead.

The implementations of an SpMM kernel with some tile sizes cannot effectively utilize GPUs, leading to suboptimal performance compared to other tile sizes. Therefore, we can reduce the search time by eliminating these underperforming tile sizes. Specifically, LO-SpMM can accurately eliminate low-performance candidates in the search space based on certain constraints, including register resource constraint, hardware utilization constraint, and load balancing constraint. Then, optionally, for the remaining tile sizes, a model is built to estimate their relative performance rank. The estimated top K tile sizes are selected for evaluation. In this way, we can obtain a

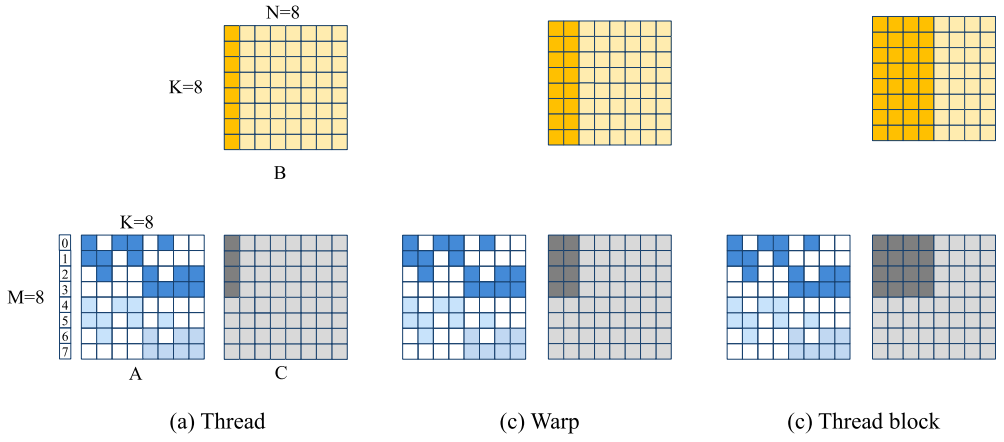


Fig. 2. An example of hierarchical 2-dimensional tiling for A * B = C. A is an 8×8 sparse matrix and the white cells within A are pruned elements. B and C are 8×8 dense matrices. In this example, we assume the device has 2 warps in a thread block, 2 threads in a warp, and tile size (M1, N1) = (4, 4), for simplicity.

high-performance implementation of an SpMM kernel with few actual evaluations, which greatly reduces the search cost.

3.1 Hierarchical 2-Dimensional Tiling

Hierarchical 2-dimensional tiling follows hierarchical 1-dimensional tiling [24] with modifications. Instead of processing only part of a row of the output matrix, each thread block in our tiling strategy processes a submatrix of the output matrix. In the tiling process, first, the sparse matrix is divided into multiple A_{tile}, each of which corresponds to $M1$ rows (could be discontinuous), of the sparse matrix. Then the dense matrix is divided into multiple B_{tile}, each of which corresponds to continuous $N1$ columns of the dense matrix. The output of $A_{tile} \times B_{tile}$ is a submatrix C_{tile} of size $(M1, N1)$. Specifically, on GPUs, an SpMM kernel is divided into three hierarchies: thread, warp, and thread block, as shown in Figure 2. Each thread processes an output submatrix C_{thread} of size $(M1, 1)$, and each warp processes an output submatrix C_{warp} of size $(M1, warp\_size)$. Each thread block processes a tile, namely, an output submatrix C_{block} of size $(M1, N1)$.

The hierarchical 2-dimensional tiling strategy has the following benefits:

— Memory accesses within a warp can be merged to improve the efficiency of SpMM.
— The workload distribution among threads in a thread block is balanced due to processing the same A_{tile}.
— Threads are designed to handle multiple output elements, as opposed to just one, thereby reducing memory access to the dense matrix. This also increases the computational work performed per memory access about the dense matrix, which is advantageous for masking the latency inherent in memory accesses.

3.2 Register Resource Constraint

Due to the constraint of GPU register resource, the register usage for both threads and thread blocks within an SpMM implementation is limited. For GPUs, each variable in a thread typically corresponds to a register. When the number of variables in a thread exceeds the maximum number of registers, the exceeded variables are stored in the local memory. The overhead of accessing the local memory is comparable to that of accessing global memory and much higher than that of

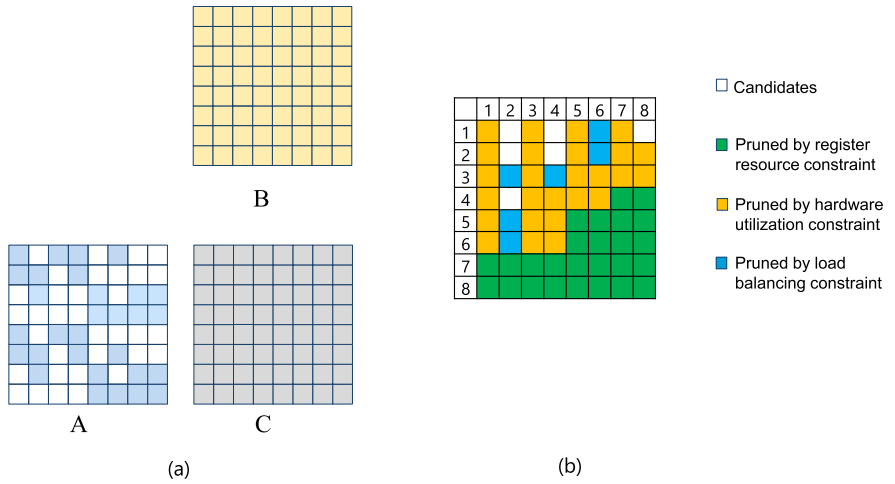


Fig. 3. An example of applying three constraints to an SpMM kernel. **(a)** Illustration of the SpMM kernel $C = A \times B$. **(b)** Tiling search space of the SpMM kernel. The 2D coordinates of each cell represent a candidate 2D tile size $(M1, N1)$. The green, orange, and blue cells are removed by register resource constraint, hardware utilization constraint, and load balancing constraint, respectively. In total, the size of the search space is reduced from $8 \times 8 = 64$ to 6 (white cells).

accessing registers. If a tile size requires exceeding registers and consequently causes local memory access, it is unlikely to exhibit optimal performance. Moreover, the number of thread blocks and threads in a kernel is determined at runtime, thereby the required register resource is also determined at runtime, which is unknown during compilation. As a result, despite successful compilation, an SpMM implementation may fail at runtime with a "too many resources requested for launch" error if the required register resource exceeds the limitation for thread blocks, leading to unnecessary compilation overhead.

Therefore, LO-SpMM counts the register usage of the SpMM kernel across different tile sizes, excluding configurations that necessitate an excessive number of registers. Due to the generated SpMM implementations for each tile size being known, we can get the number of variables used in each thread and forecast the register usage. To avoid over-optimization by the compiler, which may increase the register usage and decrease GPU occupancy, we apply the "--maxrregcount" compiler option in NVCC. It limits the register usage per thread to a threshold that is slightly larger than the variable count, without causing local memory accesses. In other words, we can assess the register usage for different tile sizes without any compilation and execution. The required registers for a thread block can be calculated as the product of the register usage per thread and the total number of threads in a thread block. According to the GPU's register capacity, tile sizes of which register demand exceeds the capacity are excluded from the search space. As shown in Figure 3, assume that the maximum number of registers is 6 in a thread and 24 in a thread block. The search space size is reduced from 64 to 38 after the register resource constraint.

3.3 Hardware Utilization Constraint

The threads within a thread block are organized into warps, which are used as **Single Instruction Multiple Threads (SIMT)** execution units. If the number of threads in a thread block is not a multiple of the warp size, some warps are only partially utilized, leading to inefficiencies. Meanwhile, a GPU consists of an array of stream multiprocessors, where each thread block is allocated to a stream multiprocessor for execution. When the number of thread blocks is less than the number of

stream multiprocessors in a GPU, some stream multiprocessors will be idle, and the hardware resources may not be fully utilized. Therefore, we can further reduce the search space by eliminating tile sizes that would result in insufficient thread blocks.

LO-SpMM sets the number of threads per block ($N1$) to be a multiple of warp size to ensure sufficient warp utilization. Additionally, LO-SpMM calculates the number of thread blocks corresponding to each tile size by Equation (1):

$$block\_num = \lceil M/M1 \rceil \times \lceil N/N1 \rceil, \tag{1}$$

where (M, N) is the shape of the sparse matrix and (M1, N1) is the tile size. Considering the influence of data reuse and cache access patterns in SpMM implementations, LO-SpMM adopts a relaxed constraint on the number of thread blocks and eliminates tile sizes that result in fewer thread blocks than half of the stream multiprocessor count, rather than the full count. As shown in Figure 3, assume the warp size is 2 and the number of stream multiprocessors is 8. The search space size is reduced from 38 to 12 after the hardware utilization constraint.

3.4 Load Balancing Constraint

In hierarchical 2-dimensional tiling, load balancing within a thread block is inherently ensured, but load balancing across thread blocks is not guaranteed. A tile size influences the load balancing among different thread blocks from two aspects: (1) the number of non-zero elements in A_{tile} that each thread block processes, and (2) the number of columns in B_{tile} that each thread block processes.

In the search space, a tile size with the same number of tiles but a more balanced load can be found, which can achieve better performance. Therefore, we should remove tile sizes that do not satisfy a certain load-balancing condition from the search space. For example, in Figure 3, if $M1$ is set to 6, the sparse matrix A is divided into two A_{tile}. The first A_{tile} has 6 rows with 22 nonzero elements, while the second A_{tile} only has 2 rows with 8 nonzero elements. In this case, the load of different tiles is imbalanced. However, setting $M1$ to 4 can keep the same number of tiles and make the load of different A_{tile} more balanced, since each A_{tile} equally has 15 nonzero elements. Likewise, setting $N1$ to 6 results in load imbalance among different tiles, where the first B_{tile} has 6 columns, and the second B_{tile} has 2 columns only. Setting $N1$ to 4 makes a load of different tiles more balanced while maintaining the same number of tiles since each B_{tile} equally has 4 columns.

LO-SpMM measures the A_{tile} load balancing using COV_{row}, calculated by the coefficient of variation of number of non-zero elements across different A_{tile}. The B_{tile} load balancing is measured using $WASTE_{col}$, calculated by the padding ratio of columns in the dense matrix. LO-SpMM sets two thresholds, TH_{row} for COV_{row} and TH_{col} for $WASTE_{col}$, to exclude tile sizes that cause COV_{row} or $WASTE_{col}$ exceeding the thresholds. We set both TH_{row} and TH_{col} to 0.25. The search space in Figure 3 is reduced from 12 to 6 after the load balancing constraint.

3.5 Rank Model

Numerous factors influence the performance of an SpMM implementation, some of which can be explicitly interpreted and analyzed, such as the aforementioned constraints. However, beyond those constraints, a significant number of candidates remain in the search space that cannot be effectively distinguished using explainable rules. Consequently, we propose constructing a model to estimate the performance of the remaining candidates through machine learning techniques. We select the top K candidates based on these estimates for compilation and execution and identify the optimal candidate as the final output.

To search for the optimal tile size, we do not care about the absolute performance of each candidate but only need to know the relative performance order of different candidates. Therefore, we

Table 1. Features of Rank Model

Type	Name	Description
Schedule Information	A_{tile} *num*	number of A_{tile}
	B_{tile} *num*	number of B_tile
	block num	number of block
Sparse Attribute	*sparsity*	sparsity of sparse matrix
	NNC_{sum}	non-empty column sum of all A_{tile}
	NNZ_{max}	max number of nonzero elements of all A_{tile}
Tile Size	$M1$	number of rows in C_{tile}
	$N1$	number of columns in C_{tile}
	K	number of columns in sparse matrix (number of rows in dense matrix)

use a rank model to sort tile sizes. We construct a listwise rank model using LambdaMART [11] as the model. The features used in the rank model are shown in Table 1. An SpMM implementation is represented by schedule information of an SpMM kernel, sparse attribute in the sparse matrix, and tile size. The training dataset is constructed by benchmarking the performance of synthetic SpMM kernels under various tile sizes and sparsity degrees.

4 Sparse Matrix Reordering

Unstructured pruning on a DNN leads to a non-uniform distribution of nonzero elements within its weight matrices. This distribution is ill-suited to GPUs, resulting in imbalanced computational loads across different tiles and increased memory access overhead. Reordering the nonzero elements in the sparse matrix can yield a distribution better optimized for SpMM kernel performance. In this section, we introduce the motivation for our reordering algorithm, along with the problem formulation and the corresponding algorithm for sparse matrix reordering. The effectiveness of the algorithm is demonstrated through an illustrative example.

4.1 Motivation

The memory access overhead has an important impact on the performance of the kernel. For example, assume an SpMM kernel with dimensions $(M, K, N) = (128, 2048, 128)$ executed on an RTX 2080Ti. To avoid the influence of other factors on the performance, this kernel is launched within a single thread block by setting the tile size to $(128, 128)$. We evaluate the performance of SpMM with different numbers of non-empty columns (*nnc*) while fixing the number of non-zero elements (*nnz*) to 2,048, 3,072, and 4,096, respectively. It is noteworthy that there is a positive correlation between *nnc* and memory access load because *nnc* reflects the frequency of accesses to the dense matrix B per thread. As illustrated in Figure 4, the runtime escalates with an increase in memory access overhead for a fixed *nnz*. The analysis reveals a performance divergence of up to 10.99× across different memory access overhead under identical computational demands. Since the SpMM kernel is memory-bound, variations in *nnz* within a tile in this example have negligible impact on the performance of the SpMM kernel.

Row reordering of the sparse matrix can reduce memory access overhead while maintaining load balancing, as illustrated in Figure 5. In this example, each thread computes 4×1 output elements, and each thread block computes 4×4 output elements. As shown in Figure 5(a), the sparse matrix A is partitioned into two A_{tile}, with rows $0-3$ assigned to the first A_{tile} and rows $4-7$ assigned to the second A_{tile}. The *nnc* in each A_{tile} is 8, resulting in 8 memory accesses per thread to fetch the corresponding elements of the dense matrix B into registers. To reduce memory accesses per

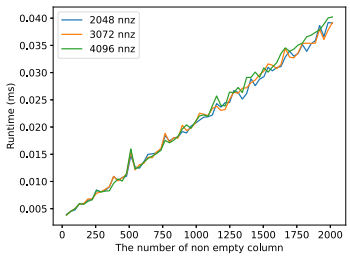


Fig. 4. Runtime under different number of non-empty columns with the same number of nonzero elements.

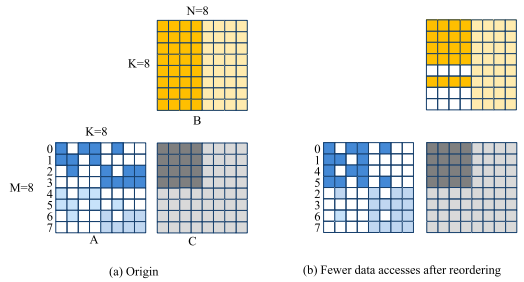


Fig. 5. Reduce memory access by reordering.

tile, the sparse matrix can be reordered by grouping rows with similar distributions of non-empty columns within each A_{tile}. This approach reduces the *nnc* in each A_{tile}, thereby decreasing the number of memory accesses per thread. Figure 5(b) demonstrates this optimization, which assigns rows $0-1$ and $4-5$ to the first A_{tile}, and rows $2-3$ and $6-7$ to the second A_{tile}. It reduces the *nnc* in each A_{tile} to 5. Consequently, each thread performs only 5 memory accesses.

4.2 Problem Formulation and Algorithm

Given a tile size $(M1, N1)$, sparse matrix reordering solves a constrained optimization problem. Its objective is to reduce the maximum number of memory accesses in all A_{tile} while maintaining load balancing among all A_{tile}. This objective can be formulated as follows:

$$\arg\min_{SA_{tile}} max(nnc(SA_{tile})), \tag{2}$$

subject to

$$max(size(SA_{tile})) \leq M1, \tag{3}$$

$$nnz\ imbalance\ of\ SA_{tile} < \alpha, \tag{4}$$

where SA_{tile} is a set of A_{tile}. The size of A_{tile} is limited by $M1$ so that all threads are constrained to have a close number of registers and write operators.

Regarding each row as a node and each column as a hyperedge, this problem can be transformed into a constrained hypergraph partitioning problem. The goal of this problem is to minimize the maximum number of hyperedges across all subgraphs while ensuring that the total number of nodes across all hyperedges is similar in each subgraph. This is an NP-hard problem. Therefore, we propose Algorithm 1 to solve this problem approximately. First, we count the number of nonzero elements in the whole sparse matrix and in each row, denoted by *nnz_matrix*, and *nnzs_row*, respectively (lines 1–2). Then we sort rows in the sparse matrix by *nnzs_row* and remove empty rows (lines 3–4). The sparse matrix rows are processed in ascending *nnzs_row* order. For each row, we merge it with each A_{tile} and sort the TSA_{tile} by the *nnc* in A_{tile} (lines 9–10). We start with the A_{tile} which has the smallest *nnc* and is smaller than $M1$ in size. If the *nnz* in A_{tile} does not exceed the *nnz* limitation (TH_{nnz}), this row is assigned to this A_{tile}. Otherwise, we choose the next A_{tile} to judge. If the *nnz* of all unfilled A_{tile} is larger than the limitation, then we assign this row to the A_{tile} with the smallest *nnc* (lines 12–22). In the reordering process, the *nnz* in each A_{tile} is limited to make *nnz* among different A_{tile} close, so as to achieve computational load balancing.

ALGORITHM 1: Sparse matrix reordering

Input : The sparse matrix, A
The size limitation of A_{tile}, M1
Output : The set of A_{tile}, SA_{tile}

```
nnz_matrix ← number of nonzero elements in sparse matrix;
nnzs_row ← numbers of nonzero elements in sparse rows;
rows ← GetReorderRow(A, nnzs_row);
rows ← RemoveEmptyRow(rows, nnzs_row);
num_A_tile ← ceil(len(rows)/M_1);
TH_nnz ← nnz_matrix/num_A_tile;
SA_tile ← InitialAtile(num_A_tile);
for row ∈ rows do
    TSA_tile ← MergeRow2Atile(SA_tile, row);
    RSA_tile ← GetReorderAtile(A_tile, TSA_tile);
    cid ← NULL;
    for A_tile ∈ RSA_tile do
        if A_tile.attribute(size) < M_1 then
            if cid is NULL then
                cid ← A_tile.attribute(ID);
            end
            if A_tile.attribute(nnz) < TH_nnz then
                cid ← A_tile.attribute(ID);
                break;
            end
        end
    end
    Assign the row to SA_tile[cid];
end
```

4.3 Effectiveness of Sparse Matrix Reordering

To evaluate the effectiveness of Algorithm 1, we constructed a comparative analysis of the performance of SpMM implementations under three scenarios: without sparse matrix reordering, using a hypergraph partitioning method, and using our proposed method. In the hypergraph partitioning method, rows and columns are treated as nodes and hyperedges, respectively, with node weights set to the *nnz* in the row to ensure computation load balancing. Then we use Kahypar [52] to optimize the connectivity objective. Assume an SpMM kernel is of shape (M, N, K) = (1,024, 1,024, 1,024) and sparsity 98%. It is divided into 32 blocks by row, and each block has 32 rows. The similarity R between two blocks is denoted by the ratio of identical nonzero element positions in the two blocks. The R in paired blocks ($block_i$ and $block_{i+16}$, $i \in \{0, 1, \ldots, 15\}$) varies from 0 to 0.9 to show the effect of similarity on these methods. Meanwhile, to show the impact of load balancing, the *nnz* ratio in adjacent blocks ($block_i$ and $block_{i+1}$, $i \in \{0, 2, \ldots, 30\}$) varies from 1 to 3.

As shown in Figure 6, with balanced load, as R increases, the runtime of the SpMM implementation tends to decrease with sparse matrix reordering and remains the same runtime without sparse matrix reordering. As the similarity of the sparse matrix increases, the memory access similarity of each block becomes higher after reordering, enabling fewer memory accesses per block. Moreover, the hypergraph partitioning method produces tiles with variable numbers of non-empty rows and columns when the *nnz* distribution per block is uneven, causing imbalanced memory access loads

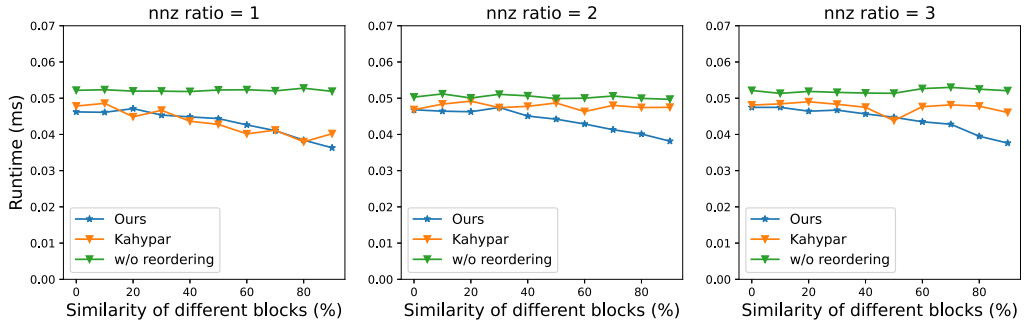


Fig. 6. Runtime of SpMM implementations with varying block similarity.

among thread blocks. In contrast, our method achieves more effective balance in memory access loads.

5 Proxy-based Evaluation

The evaluation of an SpMM implementation mainly consists of two parts: compilation and execution. The usage of full loop unrolling and constant propagation [60, 69] for accelerating an SpMM kernel often results in significantly huge code files and a long compilation time. For example, in the SpMM kernel with the sparsity of 0.7 and shape (3,072, 4,096, 768), with different tile sizes, the compilation time ranges from several hundred to over a thousand seconds, while the execution usually takes a few milliseconds.

To mitigate the issue of long compilation time, we employ a proxy-based method to evaluate SpMM implementations. A proxy is a simplified program that behaves similarly to the original SpMM implementations but has much shorter code. Meanwhile, one proxy can represent a set of SpMM implementations via setting different runtime arguments. It does not output correct computation results and does not have the same absolute execution time as the original program. It only needs to keep a similar relative performance rank as the original program, so that it can be used to estimate which SpMM implementation has higher performance.

In the proxy-based evaluation for an SpMM kernel, the generation of the optimal SpMM implementation consists of three stages. In the first stage, LO-SpMM generates a set of proxies to replace the actual performance evaluation. In the second stage, for each tile size, LO-SpMM selects an appropriate proxy and determines the runtime arguments to evaluate proxy performance on the hardware. These arguments, including grid size, block size, and shared memory usage, are tailored to the specific tile size under evaluation. In the final stage, LO-SpMM selects the tile size with the optimal proxy performance and implements the SpMM kernel based on it. The details of proxy generation and runtime arguments configuration are introduced in the following.

5.1 Proxy Generation

Figure 8(a) and (b) shows the code structure of the SpMM implementation on GPUs. It is a kernel function consisting of a series of thread block functions. Each thread block function includes an entirely unrolled loop for loading data and computing, and a step of result storing. The code length has a positive correlation to the compilation time. We use proxies to simplify original SpMM implementations at three levels: kernel function, thread block function, and loop block. Figure 7 illustrates how proxies are constructed from original implementations. Assume SpMM has multiple possible implementations. Each implementation is a kernel function that contains some thread block functions $\{B_0, B_1, \ldots\}$. Each block function contains many iterations $\{I_0, I_1, \ldots\}$. Multiple

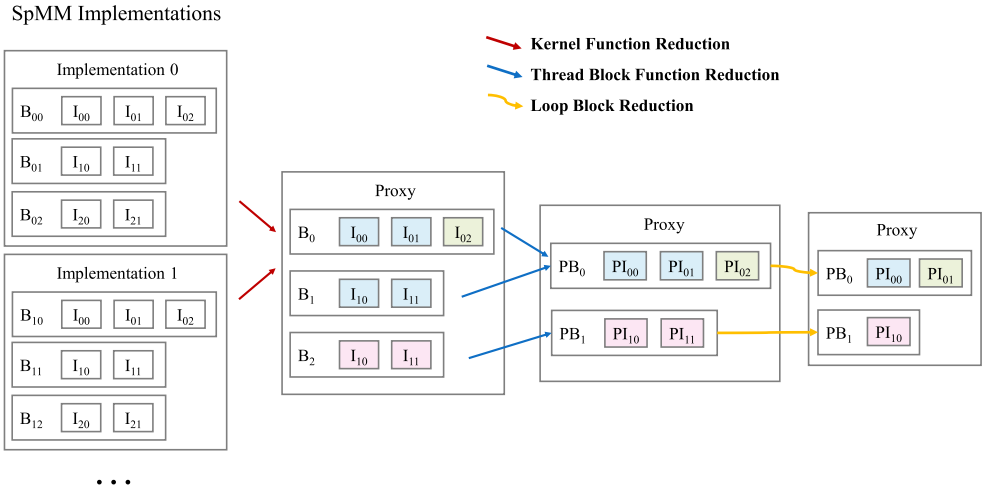


Fig. 7. The process of reducing the original SpMM implementations at three levels, including kernel function reduction, thread block function reduction, and loop block reduction. (1) Kernel function reduction uses one proxy to represent multiple kernel functions. (2) Thread block reduction uses a proxy thread block function to represent multiple thread block functions in each proxy. (3) Loop block reduction reduces the number of iterations in each proxy thread block function.

implementations share one proxy, thus the number of compilations can be reduced. Thread block functions within an implementation are clustered and represented by a smaller number of proxy thread block functions (PBs), thus the code length can be reduced. By loop block reduction, each PB only reserves a few proxy iterations (PIs), so the code length can be further reduced. Detailed steps are introduced in the following sections.

5.1.1 Kernel Function Reduction. In SpMM implementations, hardcoding the tile size as constants in code can reduce redundant computation. However, this approach produces a different code variant for each tile size, requiring repetitive compilations.

Since our purpose is to compare the relative performance rank of different implementations rather than assessing their absolute performance, we can skip the step of hardcoding tile sizes. This allows us to maintain a single proxy, which can accept different tile sizes as input arguments, to account for multiple SpMM implementations.

More specifically, with the same vertical tile size $M1$, SpMM implementations share the same code structure. Hardcoding the horizontal tile size $N1$ will generate an SpMM implementation with a specific number of threads and thread blocks. By defining $N1$ and related constants as variables, we can create a single proxy to represent multiple implementations. We can then estimate the performance of a given tile size by setting tile-related arguments at runtime. This method enables us to evaluate the performance of SpMM implementations with a variety of tile sizes after just one compilation.

5.1.2 Thread Block Function Reduction. An SpMM implementation consists of $\lceil M/M1 \rceil$ thread block functions. Each thread block function is a code snippet for processing a tile of the sparse matrix A. If we can reduce the number of thread block functions to be compiled, the code will be shortened, thereby the compilation cost will also be reduced. We achieve this by partitioning thread block functions into several disjoint clusters. Each cluster contains a set of thread block functions with similar features. According to the feature centroid (namely, the average values of

```
__global__ void SpMM(float *B, float *C){
  if (blockIdx.x == 0)
     block_func_0(B, C);
  ...

  if (blockIdx.x == N-1)
     block_func_n-1(B, C);
}
```

(a)

```
__device__ void block_func_x(float* B, float *C) {
  #pragma unroll
  for(int j = 0; j < NNC; j++){
     load data
     computation
  }
  store result
}
```

(b)

```
__global__ void SpMM_proxy(float *B, float *C){
  if (get_cluster_id(blockIdx.x) == 0)
     block_func_0_proxy(B, C);
  ...

  if (get_cluster_id(blockIdx.x) == Np - 1)
     block_func_np-1_proxy(B, C);
}
```

(c)

```
__device__ void block_func_x_proxy(float* B, float *C) {
  #pragma unroll
  for(int j = 0; j < NNCp; j++){
     proxy iteration

  }
  proxy store result
}
```

(d)

Fig. 8. The code structures of the origin SpMM implementation and proxy. NNC denotes the number of non-empty columns in A_{tile} corresponding to a thread block. **(a)** Illustration of the SpMM kernel function. **(b)** Illustration of the thread block function. **(c)** Illustration of the proxy kernel function. **(d)** Illustration of the proxy thread block function.

features) of each cluster, a proxy thread block function is constructed to represent all thread block functions within the cluster. Figure 8(c) illustrates the usage of proxy thread block functions. In the proxy SpMM implementation, it first gets the cluster id of a block and then calls the corresponding proxy thread block function. Compared with the original implementation in Figure 8(a), the proxy reduces the number of thread block functions from N to N_P.

Since we have conducted sparse matrix reordering, as introduced in Section 4, the operations in different thread block functions are relatively similar. The similarity of two thread block functions can be measured by the Euclidean distance of two features: the number of memory accesses and the number of floating-point operations within a thread block function. These features can be quickly obtained by static code analysis. We then use an agglomerative [45] method to cluster thread block functions and obtain the feature centroid of each cluster. Given the feature centroid, we construct the corresponding proxy thread block function by repeating a certain number of global memory loads and multiply-add operations until achieving close feature values to the feature centroid.

The number of clusters, which equals the number of proxy thread block functions, has a crucial impact on the performance accuracy of proxy. On the one hand, if we set an extremely small number of clusters, the proxy will have a too-short code length. This can lead to discrepant instruction fetch miss rates of proxy thread block functions, compared with the original implementation. On the other hand, the number of clusters should not be greater than the product of the number of thread block functions concurrently executed by a stream multiprocessor and the number of stream multiprocessors in a GPU. Although an SpMM implementation can have many thread block functions, only part of them can be concurrently executed by the active thread blocks in a stream multiprocessor. Since one or multiple active thread blocks execute one thread block function, the maximum number of concurrently executed thread block functions can be equal to or fewer than the number of active thread blocks. Based on our experience, we set the number of clusters to $3 \times max(NUMS_{ATB})$, where $NUMS_{ATB}$ is an array consisting of the number of active thread blocks in each stream multiprocessor of all SpMM implementations corresponding to a proxy.

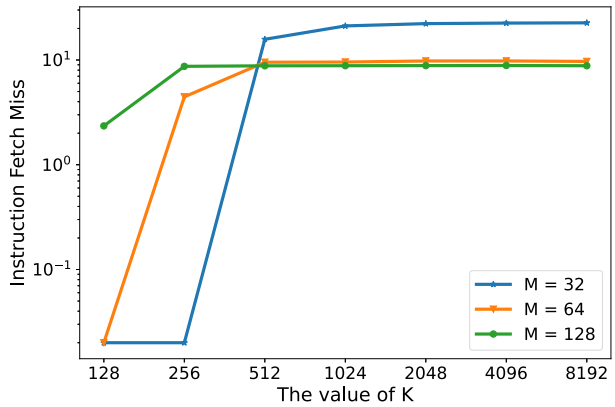


Fig. 9. The instruction fetch miss in the kernel under different iteration count K with the same number of sparse matrix row M

5.1.3 Loop Block Reduction. As shown in Figure 8(b), a thread block function contains an entirely unrolled loop block followed by a result storage step. Since we only need to identify the performance rank across different tile sizes, we do not need to compile and execute all operations in the loop block. Truncating the loop block by the same proportion across different tile sizes has a negligible impact on their performance rank. Therefore, we can replace the original loop block with a truncated version that has shorter code. Specifically, a proxy loop block is constructed by repeating a proxy iteration NNC_P times. The proxy iteration includes one global memory load and multiple multiply-add operations. The number of multiply-add operations is determined by the average number of computations among all global memory loads.

If NNC_P is excessively small, it can cause a large gap in the instruction fetch miss rates between the proxy and original implementations, leading to inaccurate performance rank. Fortunately, within a thread block, the instruction fetch miss rates tend to stabilize after reaching a certain threshold of code length. This fact is depicted in Figure 9 for SpMM kernels with different rows of sparse matrix M and iteration counts NNC_P, where the column of dense matrix N is 256,000, $sparsity$ is 0.5, and tile size is $(M, 256)$. $NNC_P * (1 - sparsity)$ is positively correlated with code length. Based on our experience, we set NNC_P to $min(NNC, \frac{300}{(1-sparsity)})$, where NNC denotes the number of non-empty columns in the A_{tile}.

5.2 Runtime Arguments Configuration

A proxy can accept different runtime arguments, including grid size, block size, and shared memory usage, so that one proxy can flexibly represent the implementations resulted from different tile sizes. The grid size and block size arguments can be directly calculated according to the tile size and SpMM shape. The usage of shared memory needs a careful configuration since it is a key factor that affects the performance similarity between the proxy and the original implementation.

5.2.1 Relation between Occupancy and Shared Memory. To ensure the proxy can keep a similar performance rank as the original SpMM implementation, we try to align their performance in terms of a dimensionless metric (e.g., a ratio between some counts). We select GPU occupancy as the metric since it provides an underlying measurement of GPU hardware utilization [18]. Occupancy refers to the ratio of active thread blocks on a stream multiprocessor (NUM_{ATB} for short) to its maximum thread block capacity. NUM_{ATB} is determined by available hardware resources, including registers, shared memory, and the number of thread blocks. Due to the

loop block reduction, the proxy uses fewer registers than the original implementation, leading to higher occupancy. To align the occupancy, we need to adjust register usage, shared memory, and the number of thread blocks. However, the number of thread blocks is determined by the tiling step and cannot be changed during constructing proxy. Moreover, the usage of registers is difficult to precisely control, as it is affected by a complicated mechanism of compiler and runtime. Therefore, we control the occupancy of the proxy by adjusting the allocation of shared memory.

5.2.2 Shared Memory Usage Configuration. Specifically, the configuration process involves two steps. First, we calculate NUM_{ATB} of the original implementation based on its register usage and the number of thread blocks, which can be formulated as follows:

$$NUM_{ATB} = min(\lceil NUM_{TB}/SMs \rceil, regCapacity/regUsage), \tag{5}$$

where SMs is the number of stream multiprocessors in GPU, NUM_{TB} is the number of thread blocks, $regCapacity$ is the capacity of registers in a stream multiprocessor, and $regUsage$ is the usage of registers in a thread block. Next, we calculate the shared memory allocation needed ($SharedMemorySize$) for each thread block in the proxy to achieve the same occupancy, which is formulated as follows:

$$SharedMemorySize = maxSmemUsage/NUM_{ATB} - Constant, \tag{6}$$

where $maxSmemUsage$ is the max shared memory usage in a stream multiprocessor. Since CUDA runtime allocates some shared memory for each thread block for management and tracking, we subtract the shared memory usage by a constant that is determined by hardware specification and runtime library version [4].

6 Implementation

In this section, we introduce the implementation details of sparse neural networks inference, including SpMM kernel and end-to-end DNN inference.

6.1 SpMM Kernel

The implementation of thread tile in an SpMM kernel is shown as Algorithm 2. During the code generation process, we entirely unroll lines 7–18 and fill in the corresponding sparse matrix elements, $A[a, k]$, which we can know at compile time, according to the constant propagation technique [60]. This approach enables the utilization of the constant cache for accessing A values, thereby minimizing random memory accesses caused by sparse indices. The accumulator, Acc, is stored in registers to enable rapid access. To further improve the performance of an SpMM implementation, we incorporate a prefetching technique for the matrix B data. Specifically, we allocate additional registers, enabling simultaneous data prefetching from matrix B while computing the current column of the sparse matrix A (lines 14–18). Besides, to avoid unnecessary memory access to the dense matrix, when nnz of $A[ROW_{list}, b]$ is empty, we do not cache $B[b, N_{thread}]$ in registers.

Based on the programming model for GPUs, the implementation of thread tile can be naturally extended to warp tile and block tile, respectively.

6.2 End-to-end DNN Inference

We implement end-to-end inference for sparse neural networks by combining LO-SpMM with SparTA and Rammer. First, we use SparTA to propagate sparse properties of sparse neural networks and obtain networks with higher sparsity. For each SpMM kernel, we obtain the Top-1 SpMM implementation using tiling search reduction, sparse matrix reordering, and proxy-based evaluation. After all SpMM implementations are generated, we optimize the neural network using Rammer. Rammer can improve the DNN performance through operator fusion and operator

ALGORITHM 2: Implementation of thread tile in SpMM kernel

Input : sparse matrix A, dense matrix B,SpMM kernel shape (M, N, K), computations per load CPL
Output : output matrix C

$ROW_{list} \leftarrow list\ of\ row\ of\ A\ to\ process\ for\ thread$;
$N_{thread} \leftarrow column\ of\ B\ to\ process\ for\ thread$;
$Acc[M_{list}, N_{thread}] \leftarrow \{0.0\}$;
$b \leftarrow number\ of\ preload\ column$;
$INDICES \leftarrow indices\ of\ nonzero\ elements\ in\ A[ROW_{list}, 0 : b]$;
$Cache\ B[0 : b, N_{thread}]\ in\ registers$;
while $INDICES.size() \geq 0$ **do**
    $num \leftarrow 0$;
    while $num < CPL$ **do**
        $(a, k) \leftarrow INDICES.pop()$;
        $Acc[a, N_{thread}] \leftarrow ACC[a, N_{thread}] + A[a, k] \times B[k, N_{thread}]$;
        $num \leftarrow num + 1$;
    end
    if $b \leq K$ **then**
        $INDICES.push(indices\ of\ nonzero\ elements\ in\ A[ROW_{list}, b])$;
        $Cache\ B[b, N_{thread}]\ in\ registers$;
        $b \leftarrow b + 1$;
    end
end

parallelism. Then, we replace all the GEMM kernels in the neural network with the SpMM kernel implementations from LO-SpMM .

LO-SpMM also supports the implementation of the sparse Conv2D operator by converting it to a combination of im2col and SpMM. For sparse Conv2D, LO-SpMM does not need to explicitly perform the im2col operator. When implementing Conv2D using SpMM, each reference to an element in the B matrix is replaced with the corresponding element in the original input activation tensor.

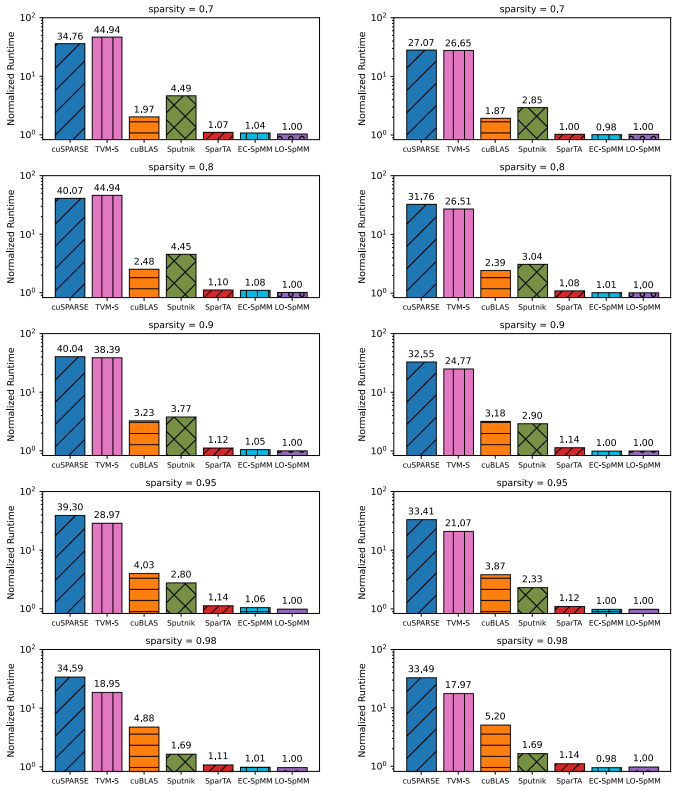
7 Evaluation

In this section, we first compare LO-SpMM with state-of-the-art SpMM implementations, evaluating the performance of generated SpMM kernels and their corresponding generation compilation time cost. Then we also evaluate the effectiveness of the rank model and proxy. Finally, we integrate LO-SpMM into SparTA and Rammer to evaluate the usefulness of LO-SpMM in end-to-end inference tasks.

7.1 Performance of Generated SpMM Kernel

We compare the performance of SpMM implementations in LO-SpMM with existing solutions, including cuSPARSE, TVM-S, Sputnik, SparTA, and EC-SpMM. Besides, we also take cuBLAS for comparison.

— cuSPARSE [2] is a vendor library for sparse computing on GPUs;
— TVM-S [13] is the sparse version of TVM with 2,000 searches for each SpMM kernel;
— Sputnik [24] carries out a lot of optimization techniques for SpMM in sparse neural networks;
— SparTA [69] is the state-of-the-art solution to sparse DNN computing;
— EC-SpMM [38] can rapidly generate a high-performance SpMM kernel;
— cuBLAS [1] is a vendor library for dense linear algebra computing on GPUs.



(a) SpMM kernel performance in 2080Ti (b) SpMM kernel performance in V100

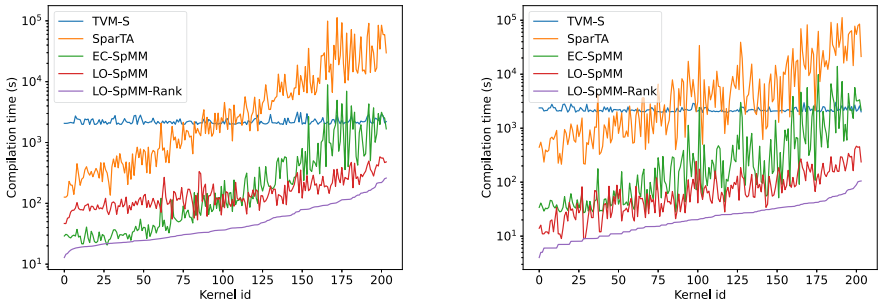
Fig. 10. The average normalized runtime of different SpMM implementations on RTX 2080Ti and V100.

The GPUs for evaluation include NVIDIA GeForce RTX 2080Ti (11 GB, 68 SMs) and NVIDIA Tesla V100 (16 GB, 80 SMs). The CUDA compiler and libraries used in experiments are from CUDA toolkit 11.1. The search process of an SpMM kernel runs on an Intel Xeon 6248 CPU.

We build a benchmark dataset to evaluate the performance of SpMM implementations. We first extract GEMM operators from a variety of neural networks, including BERT [20], ResNet50 [27], ShufflenetV2 [43], and MobilenetV2 [21]. We convert linear layers to GEMM operators and Conv2D layers to combinations of im2col and GEMM, yielding 41 distinct GEMM shapes. The GEMM kernels are randomly pruned with five sparsity degrees, generating a total of 205 SpMM kernels. The shape of an SpMM kernel is denoted as (M, N, K). In our constructed dataset, the values of M range from 64 to 3,072; N ranges from 512 to 50,176; K ranges from 27 to 3,072.

The experimental results on 2080Ti and V100 are shown in Figure 10(a) and (b), respectively. In general, LO-SpMM is 37.75×, 35.24×, 3.32×, 3.43×, 1.10×, 1.06× faster than cuSPARSE, TVM-S, cuBLAS, Sputnik, SparTA, EC-SpMM on 2080Ti, respectively. While on V100, LO-SpMM is 31.65×, 23.40×, 3.30×, 2.56×, 1.09×, 1.00× faster than cuSPARSE, TVM-S, cuBLAS, Sputnik, SparTA, EC-SpMM, respectively.

Compared to SparTA, LO-SpMM achieves at least 95% relative performance in 93.66% of kernels, and performs better in 79.02% of kernels on 2080Ti. On V100, LO-SpMM achieves at least 95% relative performance in 82.43% of kernels and performs better in 69.27% of kernels. These results indicate that, in most cases, LO-SpMM can achieve comparable or better performance than SparTA.



(a) Time cost for SpMM kernel compilation in 2080Ti

(b) Time cost for SpMM kernel compilation in V100

Fig. 11. Time cost for SpMM kernel search. Kernel id is assigned by ascending order of search time of LO-SpMM -Rank

Table 2. Normalized Average Compilation Time of Different Methods on RTX 2080Ti and V100

	TVM-S	Sparta	EC-SpMM	LO-SpMM	LO-SpMM-Rank
2080Ti	60.68	105.05	5.81	3.41	1.00
V100	139.90	281.29	18.19	3.37	1.00

Due to the usage of the prefetching technique and the exploration of more tiling configurations enabled by proxies, the performance of SpMM implementations in LO-SpMM is higher than EC-SpMM on the 2080Ti and comparable to EC-SpMM on the V100. LO-SpMM performs better at higher sparsity in general. This is due to the fact that reordering is more effective in reducing unnecessary memory accesses when the matrix has higher sparsity. Moreover, LO-SpMM outperforms cuBLAS in all cases, indicating that LO-SpMM can bring practical speedups for sparse DNNs.

7.2 Compilation Time Cost

Figure 11 shows the compilation time of four methods. By removing poorly performing candidates by constraints and using the proxies to evaluate the performance of each tile size, LO-SpMM can save a significant portion of the search cost. LO-SpMM -Rank uses the rank model to sort the candidates in the search space and then choose the Top-10 candidates for evaluation, further reducing the search cost.

The average search time normalized by LO-SpMM -Rank is shown in Table 2. Compared with SparTA, LO-SpMM saves search cost by more than an order of magnitude. LO-SpMM also achieves lower cost than EC-SpMM, which is the most effective method for quickly generating SpMM implementations in existing works. LO-SpMM -Rank combines a rank model and only evaluates a subset of candidates, so it can achieve more than 18× improvement on the V100 compared to EC-SpMM.

Because LO-SpMM , SparTA, and EC-SpMM use a full loop unrolling and constant propagation method to generate SpMM implementations, the code size of an SpMM implementation has a positive correlation to the number of nonzero elements. With the increase in the number of nonzero elements, the SpMM implementation generated by EC-SpMM and SparTA needs more compilation time. Due to the usage of the proxies, the increase in search cost for LO-SpMM is less significant. Meanwhile, the search cost reduction of LO-SpMM on V100 is more than that on 2080Ti, because

Table 3. The Data Range for Building Rank Model

Feature		Range
Sparse Matrix	M	$2^6 \sim 2^{11}$
	K	$2^6 \sim 2^{11}$
	nnz	$2^2 \sim 2^{10}$
Dense Matrix	N	$2^{10} \sim 2^{11}$
	K	$2^6 \sim 2^{11}$
Tile Size	M1	$2 \sim 2^7$
	N1	$2^7 \sim 2^{10}$

Table 4. MAPE of Proxy Kernels

	LO-SpMM	LO-SpMM-Rank
2080Ti	1.34%	1.68%
V100	2.07%	3.68%

V100 has more stream multiprocessors than 2080Ti, resulting in fewer active thread blocks per stream multiprocessor, and a lower demand for the number of proxy thread block functions. The cost of TVM-S remains stable on different SpMM kernels since its search cost is less relevant to the number of nonzero elements in a kernel.

7.3 Rank Model and Proxy Performance

We generate synthetic SpMM implementations for benchmarking performance under various tile sizes and sparsity degrees. The value range of shapes and tile sizes is shown in Table 3. Based on the benchmarking results, we can build a rank model to predict the performance of an SpMM implementation, as introduced in Section 3.5.

To evaluate the effectiveness of the rank model and constructed proxies, we measure the performance loss (mean absolute percentage error, MAPE) of the searched optimal SpMM implementation. MAPE is calculated by the relative difference between the performance of the candidate selected by LO-SpMM-Rank/LO-SpMM and the real Top-1 performance. The results are shown in Table 4. The performance of the SpMM implementation selected by the proxy-based method is close to that of the optimal kernel implementation. The performance loss is lower than 2.1% on 2080Ti and V100. That is, the proxy-based method greatly reduces the search cost of SpMM implementations on the premise of acceptable performance loss. LO-SpMM -Rank can further reduce the search cost with a slight increase in performance loss.

7.4 End-to-end Performance

To illustrate the usefulness of LO-SpMM in a complete inference procedure, we also evaluate the inference time cost of different solutions. We compare LO-SpMM with existing solutions, including Pytorch with jit, TensorRT, SparTA, Sputnik, and EC-SpMM. The value of K in EC-SpMM is set to 10 in the end-to-end experiment. We take two neural network models for evaluation: BERT and ResNet50. In the BERT model, we use movement pruning [51] to obtain a sparse model with 90.99% sparsity, and its accuracy on the QQP task is reduced from 89.1% to 86.3%. In ResNet50, we use DLTH pruning [8] without pruning the first Conv2D layer and the last linear layer, resulting in a 90% sparse model with accuracy reduced from 75.7% to 66.0% on ImageNet tasks.

The batch size for end-to-end inference is 32. In ResNet50, we convert the Conv2D layers to SpMM. Going one step further, we also implement an SpMM version of fused Conv2D+relu for better end-to-end performance. W The experimental results are shown in Figure 12. Since most of the kernels in pruned BERT can be directly implemented using SpMM, LO-SpMM can achieve a large improvement on BERT. LO-SpMM has 1.84× speedup over TensorRT, 1.34× speedup over SparTA, and 1.16× speedup over EC-SpMM. For the ResNet50 model, LO-SpMM has 1.88× speedup over TensorRT, and is comparable to SparTA and EC-SpMM. The SpMM kernels in ResNet50 have fewer non-empty rows, which lowers the benefit from sparse matrix reordering.

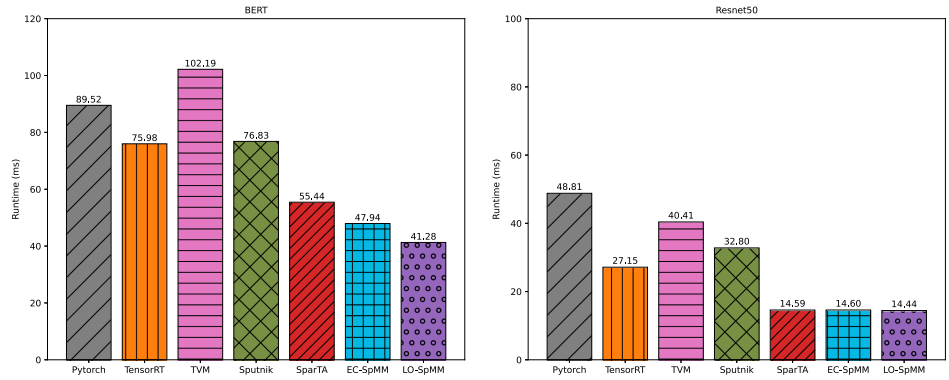


Fig. 12. The end-to-end performance of different solutions on 2080Ti.

It is worth noting that this article mainly focuses on the optimization of compilation time and latency of SpMM kernels. The end-to-end inference experiments validate the feasibility of integrating LO-SpMM into a complete inference procedure. However, the optimization of end-to-end inference involves many other techniques that are beyond the scope of this article. More elaborate optimizations for maximizing the end-to-end performance deserve further investigation in future work.

8 Related Work

This section reviews and discusses related existing studies along three categories: SpMM kernel for DNN inference, auto-tuning for SpMM kernel, and proxy program.

8.1 SpMM Kernel for DNN Inference

The SpMM kernel is crucial for the efficient inference of pruned DNNs, attracting significant research attention. The performance optimizations of SpMM mainly include three aspects. (1) **Tiling**. Tiling partitions the computation and memory access of SpMM for better data reuse and load balancing. Yang et al. [64] extend two primary tiling strategies in SpMV for the SpMM problem, row-splitting [6, 9] and merge-based [17, 44], and implement them on GPUs. AspT [31] uses adaptive sparse tiling to improve memory reuse. Sputnik [24] modifies row-splitting [64] by using multiple thread blocks to process a row of the output matrix for higher parallelism. SparseRT [60] optimizes load balancing by assigning multiple rows per thread block during tiling. (2) **Reordering**. Reordering the sparse matrix improves the data locality and load balancing of SpMM. Bandwidth-reducing reordering algorithms [16, 25, 40] can reduce the matrix bandwidth of symmetric sparse matrices. For asymmetrical sparse matrices, which are common in sparse neural networks, graph partitioning [12, 32, 66] can be used to enhance data locality. (3) **Extracting regular components**. Extracting regular components from a sparse matrix can reduce the storage overhead and enhance data reuse. The polyhedral framework is utilized to mine regular sub-regions, which does not need any indirection array to recover the nonzero coordinates [7, 56, 67]. A sparse matrix can be decomposed into submatrices, which are then categorized into multiple categories, each stored and processed independently [14, 61].

In this study, the proposed LO-SpMM employs a hierarchical 2-dimensional tiling method, which is a modification of Sputnik[24], to obtain better data reuse. Additionally, LO-SpMM integrates a novel row reordering algorithm that minimizes kernel memory access while maintaining load balancing. Given that LO-SpMM enhances data reuse by reordering and eliminates the storage

of sparse matrices through full loop unrolling, LO-SpMM does not involve techniques of extracting regular components.

8.2 Auto-tuning for SpMM Kernel

Algorithms and applications typically have optional configurations that profoundly affect their performance. Auto-tuning aims at automatically searching for the optimal configuration with the best performance, under a certain search budget. General-purpose techniques regard auto-tuning as a global optimization of an expensive black-box function. This type of technique includes OpenTuner [5], KernelTuner [58], KTT [48], ATF [50], GPTune [41], BaCO [29], and so on. These techniques also use hardware information [22, 48, 57] or provide interfaces for user-defined constraints [29, 41, 48, 50, 58] to refine the search space for faster auto-tuning. Meanwhile, performance models [15, 22, 29, 41, 62] are also used to improve the quality of the searched candidates.

In the field of deep learning, tensor compilers have been proposed for auto-tuning kernels in DNN inference. TVM and Ansor [13, 68] abstract dense operators into a tensor IR form and auto-tune their implementations. TACO [35], SparTA [69], and SparseTIR [65] abstract the sparse operator and then use the abstracted representation to construct the search space. However, these tensor compilers fail to effectively leverage prior knowledge of hardware and algorithms, leading to unnecessary search costs. Besides, they require complete code compilation for measuring performance, which takes a long time for optimized SpMM kernels.

In this study, the proposed LO-SpMM utilizes GPU hardware features and SpMM algorithm features to constrain the search space and construct a rank model, thereby reducing the number of evaluations. Additionally, LO-SpMM employs proxies to decrease the cost of measurement. These strategies significantly reduce the search overhead for optimizing SpMM.

8.3 Proxy Program

Evaluating parallel programs can be labor-intensive and time-consuming, due to the prolonged runtime, the complicated software stack, and system dependencies. Proxy programs are proposed to mimic the performance characteristics of the target applications and enable convenient performance evaluation [18, 19, 49, 53]. Low-cost proxy programs can be constructed via partial execution [34], sampled simulation [26, 55], and shrunk datasets [33, 54, 57]. PerfProx [46] generates proxies by leveraging hardware performance counters to monitor and extrapolate database performance.

In this study, we focus on designing specific proxies tailored for quickly evaluating SpMM on GPUs, rather than developing a proxy synthesis method for general parallel programs. By minimizing the code size of proxies based on the SpMM kernel implementation, we significantly enhance the efficiency of the auto-tuning process.

9 Conclusion

In this article, we presented LO-SpMM , a framework designed to efficiently generate high-performance SpMM implementations for sparse DNNs on GPUs. LO-SpMM employs a hierarchical 2-dimensional tiling strategy to define the search space for optimal tile sizes, and utilizes a set of constraints and a rank model to effectively prune the search space. Based on the architecture of GPUs and the structure of SpMM implementations, LO-SpMM creates proxies for efficiently evaluating code variants. It considerably diminishes the evaluation cost, accelerating the overall process of producing the optimal SpMM implementation. Furthermore, by reordering the sparse matrix involved in SpMM, LO-SpMM improves the performance of generated SpMM implementations. Compared with the state-of-the-art tensor compilers, our approach can reduce the search time by 281×

at most. Moreover, SpMM implementations generated by LO-SpMM outperform those compiled by SparTA and EC-SpMM, showing an average speed improvement of 10% and 3%, respectively.

Future work will focus on more types of sparse kernels in deep learning, such as **Sampled Dense Matrix Multiplication** (**SDDMM**) in graph neural networks and **Sparse Matrix-Vector Multiplication** (**SpMV**) in sparse large language models. We aim at developing a unified abstraction for these sparse kernels and investigate accurate performance models to efficiently generate kernel implementations.

References

[1] 2022. Basic Linear Algebra on NVIDIA GPUs. https://docs.nvidia.com/cuda/cublas/index.html. Accessed: February 1, 2022.

[2] 2022. A High-Performance CUDA Library for Sparse Matrix-Matrix Multiplication. https://docs.nvidia.com/cuda/cusparse/index.html. Accessed: February 2, 2022.

[3] 2024. CUDA C++ Programming Guide. https://docs.nvidia.com/cuda/cuda-c-programming-guide/. Accessed: January 1, 2024

[4] 2024. The sdk for high-performance deep learning inference. https://docs.nvidia.com/deeplearning/tensorrt/. Accessed: July 3, 2024.

[5] Jason Ansel, Shoaib Kamil, Kalyan Veeramachaneni, Jonathan Ragan-Kelley, Jeffrey Bosboom, Una-May O'Reilly, and Saman Amarasinghe. 2014. Opentuner: An extensible framework for program autotuning. In *Proceedings of the 23rd International Conference on Parallel Architectures and Compilation*. 303–316.

[6] Arash Ashari, Naser Sedaghati, John Eisenlohr, Srinivasan Parthasarath, and P Sadayappan. 2014. Fast sparse matrix-vector multiplication on GPUs for graph applications. In *SC'14: Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*. IEEE, 781–792.

[7] Travis Augustine, Janarthanan Sarma, Louis-Noël Pouchet, and Gabriel Rodríguez. 2019. Generating piecewise-regular code from irregular structures. In *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation*. 625–639.

[8] Yue Bai, Huan Wang, Zhiqiang Tao, Kunpeng Li, and Yun Fu. 2022. Dual lottery ticket hypothesis. In *Proceedings of the 10th International Conference on Learning Representations, ICLR 2022, Virtual Event, April 25-29, 2022*. OpenReview.net.

[9] Nathan Bell and Michael Garland. 2009. Implementing sparse matrix-vector multiplication on throughput-oriented processors. In *Proceedings of the Conference on High Performance Computing Networking, Storage and Analysis*. 1–11.

[10] Hadjer Benmeziane, Kaoutar El Maghraoui, Hamza Ouarnoughi, Smaïl Niar, Martin Wistuba, and Naigang Wang. 2021. Hardware-aware neural architecture search: Survey and taxonomy. In *Proceedings of the 30th International Joint Conference on Artificial Intelligence, IJCAI 2021, Virtual Event/Montreal, Canada, 19-27 August 2021*. ijcai.org, 4322–4329.

[11] Zhe Cao, Tao Qin, Tie-Yan Liu, Ming-Feng Tsai, and Hang Li. 2007. Learning to rank: from pairwise approach to listwise approach. In *Proceedings of the 24th International Conference on Machine Learning (ICML 2007), Corvallis, Oregon, USA, June 20-24, 2007*. Zoubin Ghahramani (Ed.), ACM International Conference Proceeding Series, Vol. 227, ACM, 129–136. DOI: https://doi.org/10.1145/1273496.1273513

[12] Umit V. Catalyurek and Cevdet Aykanat. 1999. Hypergraph-partitioning-based decomposition for parallel sparse-matrix vector multiplication. *IEEE Transactions on Parallel and Distributed Systems* 10, 7 (1999), 673–693.

[13] Tianqi Chen, Thierry Moreau, Ziheng Jiang, Lianmin Zheng, Eddie Yan, Haichen Shen, Meghan Cowan, Leyuan Wang, Yuwei Hu, Luis Ceze, Carlos Guestrin, and Arvind Krishnamurthy. 2018. TVM: An automated end-to-end optimizing compiler for deep learning. In *Proceedings of the 13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)*.

[14] Kazem Cheshmi, Zachary Cetinic, and Maryam Mehri Dehnavi. 2022. Vectorizing sparse matrix computations with partially-strided codelets. In *Proceedings of the SC22: International Conference for High Performance Computing, Networking, Storage and Analysis*. IEEE, 1–15.

[15] Chris Cummins, Pavlos Petoumenos, Zheng Wang, and Hugh Leather. 2017. End-to-end deep learning of optimization heuristics. In *Proceedings of the 2017 26th International Conference on Parallel Architectures and Compilation Techniques (PACT)*. IEEE, 219–232.

[16] Elizabeth Cuthill and James McKee. 1969. Reducing the bandwidth of sparse symmetric matrices. In *Proceedings of the 1969 24th National Conference*. 157–172.

[17] Steven Dalton, Sean Baxter, Duane Merrill, Luke Olson, and Michael Garland. 2015. Optimizing sparse matrix operations on gpus using merge path. In *Proceedings of the 2015 IEEE International Parallel and Distributed Processing Symposium*. IEEE, 407–416.

[18] Etem Deniz and Alper Sen. 2015. Minime-gpu: Multicore benchmark synthesizer for gpus. *ACM Transactions on Architecture and Code Optimization* 12, 4 (2015), 1–25.
[19] Etem Deniz, Alper Sen, Brian Kahne, and Jim Holt. 2014. Minime: Pattern-aware multicore benchmark synthesizer. *IEEE Transactions on Computers* 64, 8 (2014), 2239–2252.
[20] Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. 2019. BERT: Pre-training of deep bidirectional transformers for language understanding. In *Proceedings of the 2019 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies, NAACL-HLT 2019, Minneapolis, MN, USA, June 2-7, 2019, Volume 1 (Long and Short Papers)*. Association for Computational Linguistics, 4171–4186.
[21] Peiyan Dong, Siyue Wang, Wei Niu, Chengming Zhang, Sheng Lin, Zhengang Li, Yifan Gong, Bin Ren, Xue Lin, and Dingwen Tao. 2020. RTMobile: Beyond real-time mobile acceleration of RNNs for speech recognition. In *Proceedings of the 57th ACM/IEEE Design Automation Conference, DAC 2020, San Francisco, CA, USA, July 20-24, 2020*. IEEE, 1–6.
[22] Jiří Filipovič, Jana Hozzová, Amin Nezarat, Jaroslav Ol'ha, and Filip Petrovič. 2022. Using hardware performance counters to speed up autotuning convergence on GPUs. *Journal of Parallel and Distributed Computing* 160 (2022), 16–35.
[23] Luciano Floridi and Massimo Chiriatti. 2020. GPT-3: Its nature, scope, limits, and consequences. *Minds and Machines* 30, 4 (2020), 681–694.
[24] Trevor Gale, Matei Zaharia, Cliff Young, and Erich Elsen. 2020. Sparse GPU kernels for deep learning. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis, SC 2020, Virtual Event/Atlanta, Georgia, USA, November 9-19, 2020*. IEEE/ACM, 17.
[25] Norman E. Gibbs, William G. Poole, Jr, and Paul K. Stockmeyer. 1976. An algorithm for reducing the bandwidth and profile of a sparse matrix. *SIAM Journal on Numerical Analysis* 13, 2 (1976), 236–250.
[26] Greg Hamerly, Erez Perelman, Jeremy Lau, and Brad Calder. 2005. Simpoint 3.0: Faster and more flexible program phase analysis. *Journal of Instruction Level Parallelism* 7, 4 (2005), 1–28.
[27] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. 2016. Deep residual learning for image recognition. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*. 770–778.
[28] Yang He and Lingao Xiao. 2024. Structured pruning for deep convolutional neural networks: A survey. *IEEE Transactions on Pattern Analysis and Machine Intelligence* 46, 5 (2024), 2900–2919. https://doi.org/10.1109/TPAMI.2023.333461
[29] Erik Orm Hellsten, Artur Souza, Johannes Lenfers, Rubens Lacouture, Olivia Hsu, Adel Ejjeh, Fredrik Kjolstad, Michel Steuwer, Kunle Olukotun, and Luigi Nardi. 2023. Baco: A fast and portable Bayesian compiler optimization framework. In *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 4*. 19–42.
[30] Torsten Hoefler, Dan Alistarh, Tal Ben-Nun, Nikoli Dryden, and Alexandra Peste. 2021. Sparsity in Deep Learning: Pruning and growth for efficient inference and training in neural networks. *Journal of Machine Learning Research* 22, 241 (2021), 1–124.
[31] 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 ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, PPoPP 2019, Washington, DC, USA, February 16-20, 2019*. ACM, 300–314.
[32] George Karypis and Vipin Kumar. 1998. A fast and high quality multilevel scheme for partitioning irregular graphs. *SIAM Journal on Scientific Computing* 20, 1 (1998), 359–392.
[33] Kimberly Keeton and David A Patterson. 2000. Towards a simplified database workload for computer architecture evaluations. *Workload Characterization for Computer System Design* 542 (2000), 49–71.
[34] Keunsoo Kim, Changmin Lee, Jung Ho Jung, and Won Woo Ro. 2014. Workload synthesis: Generating benchmark workloads from statistical execution profile. In *Proceedings of the 2014 IEEE International Symposium on Workload Characterization (IISWC)*. IEEE, 120–129.
[35] Fredrik Kjolstad, Stephen Chou, David Lugato, Shoaib Kamil, and Saman Amarasinghe. 2017. Taco: A tool to generate tensor algebra kernels. In *Proceedings of the 2017 32nd IEEE/ACM International Conference on Automated Software Engineering (ASE)*. 943–948.
[36] Süreyya Emre Kurt, Aravind Sukumaran-Rajam, Fabrice Rastello, and P. Sadayappan. 2020. Efficient tiled sparse matrix multiplication through matrix signatures. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis, SC 2020, Virtual Event/Atlanta, Georgia, USA, November 9-19, 2020*. IEEE/ACM, 87.
[37] Chaojian Li, Zhongzhi Yu, Yonggan Fu, Yongan Zhang, Yang Zhao, Haoran You, Qixuan Yu, Yue Wang, Cong Hao, and Yingyan Lin. 2021. HW-NAS-bench: Hardware-aware neural architecture search benchmark. In *Proceedings of the 9th International Conference on Learning Representations, ICLR 2021, Virtual Event, Austria, May 3-7, 2021*. OpenReview.net.
[38] Junqing Lin, Honghe Zhang, Xiaolong Shi, Jingwei Sun, Xianzhi Yu, Jun Yao, and Guangzhong Sun. 2023. EC-SpMM: Efficient compilation of SpMM kernel on GPUs. In *Proceedings of the 52nd International Conference on Parallel Processing*. 21–30.

[39] Mingbao Lin, Rongrong Ji, Yuxin Zhang, Baochang Zhang, Yongjian Wu, and Yonghong Tian. 2020. Channel pruning via automatic structure search. In *Proceedings of the 29th International Joint Conference on Artificial Intelligence, IJCAI 2020*. Christian Bessiere (Ed.), ijcai.org, 673–679.

[40] Wai-Hung Liu and Andrew H. Sherman. 1976. Comparative analysis of the cuthill–mckee and the reverse cuthill–mckee ordering algorithms for sparse matrices. *SIAM Journal on Numerical Analysis* 13, 2 (1976), 198–213.

[41] Yang Liu, Wissam M. Sid-Lakhdar, Osni Marques, Xinran Zhu, Chang Meng, James W. Demmel, and Xiaoye S. Li. 2021. GPTune: Multitask learning for autotuning exascale applications. In *Proceedings of the 26th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*. 234–246.

[42] Lingxiao Ma, Zhiqiang Xie, Zhi Yang, Jilong Xue, Youshan Miao, Wei Cui, Wenxiang Hu, Fan Yang, Lintao Zhang, and Lidong Zhou. 2020. Rammer: Enabling holistic deep learning compiler optimizations with rTasks. In *Proceedings of the 14th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2020, Virtual Event, November 4-6, 2020*. USENIX Association, 881–897.

[43] Ningning Ma, Xiangyu Zhang, Hai-Tao Zheng, and Jian Sun. 2018. ShuffleNet V2: Practical guidelines for efficient CNN architecture design. In *Proceedings of the 15th European Conference on Computer Vision - ECCV 2018 - Munich, Germany, September 8-14, 2018, Proceedings, Part XIV (Lecture Notes in Computer Science, Vol. 11218)*. Springer, 122–138.

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

[45] Daniel Müllner. 2011. Modern hierarchical, agglomerative clustering algorithms. arXiv preprint arXiv:1109.2378 (2011).

[46] Reena Panda and Lizy Kurian John. 2017. Proxy benchmarks for emerging big-data workloads. In *Proceedings of the 2017 26th International Conference on Parallel Architectures and Compilation Techniques (PACT)*. IEEE, 105–116.

[47] Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, Alban Desmaison, Andreas Köpf, Edward Z. Yang, Zachary DeVito, Martin Raison, Alykhan Tejani, Sasank Chilamkurthy, Benoit Steiner, Lu Fang, Junjie Bai, and Soumith Chintala. 2019. PyTorch: An imperative style, high-performance deep learning library. In *Advances in Neural Information Processing Systems 32: Annual Conference on Neural Information Processing Systems 2019, NeurIPS 2019, December 8-14, 2019, Vancouver, BC, Canada*, Hanna M. Wallach, Hugo Larochelle, Alina Beygelzimer, Florence d'Alché-Buc, Emily B. Fox, and Roman Garnett (Eds.). 8024–8035.

[48] Filip Petrovič, David Střelák, Jana Hozzová, Jaroslav Ol'ha, Richard Trembeckỳ, Siegfried Benkner, and Jiří Filipovič. 2020. A benchmark set of highly-efficient CUDA and OpenCL kernels and its dynamic autotuning with kernel tuning toolkit. *Future Generation Computer Systems* 108 (2020), 161–177.

[49] Pablo Prieto, Pablo Abad, Jose Angel Gregorio, and Valentin Puente. 2021. Fast, accurate processor evaluation through heterogeneous, sample-based benchmarking. *IEEE Transactions on Parallel and Distributed Systems* 32, 12 (2021), 2983–2995.

[50] Ari Rasch, Richard Schulze, Michel Steuwer, and Sergei Gorlatch. 2021. Efficient auto-tuning of parallel programs with interdependent tuning parameters via auto-tuning framework (ATF). *ACM Transactions on Architecture and Code Optimization* 18, 1 (2021), 1–26.

[51] Victor Sanh, Thomas Wolf, and Alexander M. Rush. 2020. Movement pruning: Adaptive sparsity by fine-tuning. In *Proceedings of the Advances in Neural Information Processing Systems 33: Annual Conference on Neural Information Processing Systems 2020, NeurIPS 2020, December 6-12, 2020, Virtual*. Hugo Larochelle, Marc'Aurelio Ranzato, Raia Hadsell, Maria-Florina Balcan, and Hsuan-Tien Lin (Eds.).

[52] Sebastian Schlag, Tobias Heuer, Lars Gottesbüren, Yaroslav Akhremtsev, Christian Schulz, and Peter Sanders. 2023. High-quality hypergraph partitioning. *ACM Journal of Experimental Algorithmics* 27 (2023), 1–39.

[53] Alper Sen, Etem Deniz, and Brian Kahne. 2017. MINIME-validator: Validating hardware with synthetic parallel test-cases. In *Proceedings of the Design, Automation and Test in Europe Conference and Exhibition (DATE), 2017*. IEEE, 386–391.

[54] Minglong Shao, Anastassia Ailamaki, and Babak Falsafi. 2005. DBmbench: Fast and accurate database workload representation on modern microarchitecture. In *Proceedings of the 2005 Conference of the Centre for Advanced Studies on Collaborative Research*. 254–267.

[55] Timothy Sherwood, Erez Perelman, Greg Hamerly, and Brad Calder. 2002. Automatically characterizing large scale program behavior. *ACM SIGPLAN Notices* 37, 10 (2002), 45–57.

[56] Michelle Mills Strout, Mary Hall, and Catherine Olschanowsky. 2018. The sparse polyhedral framework: Composing compiler-generated inspector-executor code. *Proceedings of the IEEE* 106, 11 (2018), 1921–1934.

[57] Luk Van Ertvelde and Lieven Eeckhout. 2010. Benchmark synthesis for architecture and compiler exploration. In *Proceedings of the IEEE International Symposium on Workload Characterization (IISWC'10)*. IEEE, 1–11.

[58] Ben van Werkhoven. 2019. Kernel tuner: A search-optimizing GPU code auto-tuner. *Future Generation Computer Systems* 90 (2019), 347–358.
[59] Huan Wang, Can Qin, Yue Bai, Yulun Zhang, and Yun Fu. 2022. Recent advances on neural network pruning at initialization. In *Proceedings of the International Joint Conference on Artificial Intelligence, IJCAI, Vienna, Austria.* 23–29.
[60] Ziheng Wang. 2020. SparseRT: Accelerating unstructured sparsity on GPUs for deep learning inference. In *Proceedings of the PACT'20: International Conference on Parallel Architectures and Compilation Techniques, Virtual Event, GA, USA, October 3-7, 2020.* Vivek Sarkar and Hyesoon Kim (Eds.), ACM, 31–42.
[61] Lucas Wilkinson, Kazem Cheshmi, and Maryam Mehri Dehnavi. 2023. Register tiling for unstructured sparsity in neural network inference. *Proceedings of the ACM on Programming Languages* 7, PLDI (2023), 1995–2020.
[62] Floris-Jan Willemsen, Rob van Nieuwpoort, and Ben van Werkhoven. 2021. Bayesian optimization for auto-tuning GPU kernels. In *Proceedings of the 2021 International Workshop on Performance Modeling, Benchmarking and Simulation of High Performance Computer Systems (PMBS).* IEEE, 106–117.
[63] Jie Xin, Xianqi Ye, Long Zheng, Qinggang Wang, Yu Huang, Pengcheng Yao, Linchen Yu, Xiaofei Liao, and Hai Jin. 2021. Fast sparse deep neural network inference with flexible SpMM optimization space exploration. In *Proceedings of the 2021 IEEE High Performance Extreme Computing Conference, HPEC 2021, Waltham, MA, USA, September 20-24, 2021.* IEEE, 1–7.
[64] Carl Yang, Aydın Buluç, and John D. Owens. 2018. Design principles for sparse matrix multiplication on the gpu. In *Proceedings of the European Conference on Parallel Processing.* Springer, 672–687.
[65] Zihao Ye, Ruihang Lai, Junru Shao, Tianqi Chen, and Luis Ceze. 2023. SparseTIR: Composable abstractions for sparse compilation in deep learning. In *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 3.* 660–678.
[66] A. N. Yzelman and Rob H. Bisseling. 2009. Cache-oblivious sparse matrix–vector multiplication by using sparse matrix partitioning methods. *SIAM Journal on Scientific Computing* 31, 4 (2009), 3128–3154.
[67] Tuowen Zhao, Tobi Popoola, Mary Hall, Catherine Olschanowsky, and Michelle Strout. 2022. Polyhedral specification and code generation of sparse tensor contraction with co-iteration. *ACM Transactions on Architecture and Code Optimization* 20, 1 (2022), 1–26.
[68] Lianmin Zheng, Chengfan Jia, Minmin Sun, Zhao Wu, Cody Hao Yu, Ameer Haj-Ali, Yida Wang, Jun Yang, Danyang Zhuo, Koushik Sen, Joseph E. Gonzalez, and Ion Stoica. 2020. Ansor: Generating high-performance tensor programs for deep learning. In *Proceedings of the 14th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2020, Virtual Event, November 4-6, 2020.* USENIX Association, 863–879.
[69] Ningxin Zheng, Bin Lin, Quanlu Zhang, Lingxiao Ma, Yuqing Yang, Fan Yang, Yang Wang, Mao Yang, and Lidong Zhou. 2022. SparTA: Deep-learning model sparsity via tensor-with-sparsity-attribute. In *Proceedings of the 16th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2022, Carlsbad, CA, USA, July 11-13, 2022.* USENIX Association, 213–232.
[70] Hongyu Zhu, Ruofan Wu, Yijia Diao, Shanbin Ke, Haoyu Li, Chen Zhang, Jilong Xue, Lingxiao Ma, Yuqing Xia, Wei Cui, Fan Yang, Mao Yang, Lidong Zhou, Asaf Cidon, and Gennady Pekhimenko. 2022. ROLLER: Fast and efficient tensor compilation for deep learning. In *Proceedings of the 16th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2022, Carlsbad, CA, USA, July 11-13, 2022.* USENIX Association, 233–248.

Received 22 January 2024; revised 19 June 2024; accepted 17 July 2024