WACO: Learning Workload-Aware Co-optimization of the Format and Schedule of a Sparse Tensor Program

Jaeyeon Won
MIT CSAIL
Cambridge, MA, USA
jaeyeon@mit.edu

Charith Mendis
UIUC
Urbana and Champaign, IL, USA
charithm@illinois.edu

Joel S. Emer
MIT CSAIL / NVIDIA
Cambridge, MA, USA
emer@csail.mit.edu

Saman Amarasinghe
MIT CSAIL
Cambridge, MA, USA
saman@csail.mit.edu

Abstract

In this paper, we present WACO, a novel method of co-optimizing the format and the schedule of a given sparsity pattern in a sparse tensor program. A core challenge in this paper is the design of a lightweight cost model that accurately predicts the runtime of a sparse tensor program by considering the sparsity pattern, the format, and the schedule. The key idea in addressing this is exploiting a sparse convolutional network to learn meaningful features of the sparsity pattern and embedding a coupled behavior between the format and the schedule using a specially designed schedule template. In addition, within the enormous search space of co-optimization, our novel search strategy, an approximate nearest neighbor search, efficiently and accurately retrieves the best format and schedule for a given sparsity pattern. We evaluated WACO for four different algorithms (SpMV, SpMM, SDDMM, and MTTKRP) on a CPU using 726 different sparsity patterns. Our experimental results showed that WACO outperformed four state-of-the-art baselines, Intel MKL, BestFormat, TACO with a default schedule, and ASpT. Compared to the best of four baselines, WACO achieved 1.43×, 1.18×, 1.14×, and 1.27× average speedups on SpMV, SpMM, SDDMM, and MTTKRP, respectively.

CCS Concepts

• **Software and its engineering** → **Compilers**; **Domain specific languages**.

Keywords

Sparse Tensor, Auto-Scheduling, Tensor Compiler

ACM Reference Format:
Jaeyeon Won, Charith Mendis, Joel S. Emer, and Saman Amarasinghe. 2023. WACO: Learning Workload-Aware Co-optimization of the Format and Schedule of a Sparse Tensor Program. In *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2 (ASPLOS '23), March 25–29, 2023, Vancouver, BC, Canada.* ACM, New York, NY, USA, 15 pages. https://doi.org/10.1145/3575693.3575742

ASPLOS '23, March 25–29, 2023, Vancouver, BC, Canada

ACM ISBN 978-1-4503-9916-6/23/03.
https://doi.org/10.1145/3575693.3575742

1 Introduction

Sparse tensor algebra is an indispensable tool in many domains, such as graph analytics [23], scientific computing [2], and deep learning [20]. Unlike in dense tensor algebra, where only the shape of the tensor matters, the performance of sparse tensor algebra depends heavily on the sophisticated sparsity pattern of the tensor. Over the last several decades, many sparse formats have been proposed, but none of them was universally optimal across all sparsity patterns. A different schedule that transforms the traversal order of the iteration space can lead to significant performance changes depending on the sparsity pattern. For example, a sparse matrix with a skewed distribution of non-zeros must exploit fine-grained load balancing, whereas coarse-grained load-balancing must be applied to a sparse matrix with uniformly distributed non-zeros.

Recently, Kjolstad et al. presented TACO [25], a compiler for sparse tensor algebra, which generalizes many proposed sparse formats by introducing a format abstraction [12]. In addition, a sparse iteration space transformation framework was implemented on top of TACO [41]. This framework allows the compiler to generate a code with schedules that perform loop splitting, reordering, parallelizing, and other tasks to explore different traversal orders of iteration space. Although prior studies had built the *mechanism* of the compiler that enables the code generation supporting many different formats and schedules, the *policy* of the compiler that decides the best format and the best schedule for a given sparsity pattern, has not yet been designed. Unfortunately, a single format or fixed implementation cannot be globally optimal for all sparsity patterns. Thus, designing this policy is closely related to a program auto-tuning problem.

Program auto-tuning has been heavily used to optimize dense tensor programs the performance of which depends on the input size. It started with traditional high-performance scientific libraries such as ATLAS [47] and FFTW [15]. They self-optimize their interest routines by empirically transforming the program for the given input shape. Recently, languages such as Halide [39], Tiramisu [5], and TVM [9] decouple algorithms from schedule primitives to transform the structure of the loop in dense tensor programs. Such scheduling languages allow the expression of a broader range of

algorithms (compared to the limited BLAS routines in ATLAS) and the introduction of a huge search space due to schedules.

Auto-tuning sparse computation is not new [29, 32, 34, 42, 46]; even production systems are introducing auto-tuning workflows for sparse computations. For example, Intel MKL uses an inspector-executor model to auto-tune a few popular sparse computations [34]. However, the current production as well as the state-of-the-art research systems have the following limitations.

Limitations in capturing the sparsity pattern. For a dense tensor program, an auto-tuner only needs the tensor's shape. However, a shape alone fails to capture the sophisticated sparsity pattern. Capturing the sparsity pattern with the entire sparse matrix is costly because the number of non-zeros can reach billions. To summarize the sparsity pattern, much more information is required, such as the density, the size of dense blocks, and the existence of symmetry. Designing features that accurately summarize the sparsity pattern is critical for optimal decision-making in auto-tuning. Existing approaches fall short of fully capturing the pattern because they rely either on manually crafted features [27, 40] or a convolutional neural network with a downsampled matrix [42, 48], both of which result in significant information loss of the sparsity pattern.

Absence of co-optimization. The joint optimization of the data layout and the schedule is critical even in dense tensor programs, which are simpler than sparse tensor programs [22]. Nevertheless, prior auto-tuning studies on sparse tensor programs mainly tackled only one of two problems: choosing the best schedule or the best format. For instance, Intel MKL supports the inspector-executor sparse BLAS routines [34] that the executor calls the routine tuned by an inspector. However, MKL inspector misses optimization opportunities because it limits the tuning space by fixing the format. It is necessary to consider a coupled behavior between the format and the schedule to get the good performance.

Our approach. This paper presents the **W**orkload-**A**ware **Co**-**O**ptimization (WACO), a framework for automatically and jointly optimizing the format and the schedule of a given sparsity pattern. WACO uses a deep-learning based cost model that accurately and efficiently predicts the performance of the sparse tensor program. The cost model uses a novel sparse convolutional network, *WACONet*, to extract rich features of a sparsity pattern and uses a unified schedule template, *SuperSchedule*, to understand both the format and the traversal order of the iteration space. WACO further utilizes an approximate nearest neighbor search to quickly search for the optimal format and schedule over the huge search space.

Overall, our main contributions are as follows:

- To the best of our knowledge, WACO is the first auto-tuner that co-optimizes the format and the schedule in a workload-aware manner for a sparse tensor program.
- WACO is the first autotuner with a cost model that considers the coupled behavior of the sparsity pattern, the format, and the schedule.
- WACO introduces a sparsity pattern feature extractor *WACONet*, a novel sparse convolutional network architecture to effectively learn meaningful features to represent a sparsity pattern.

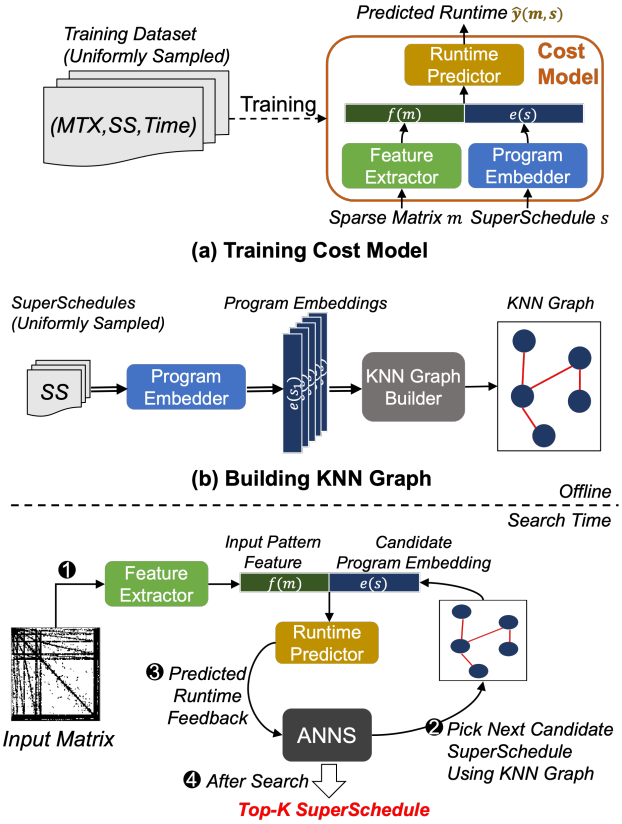


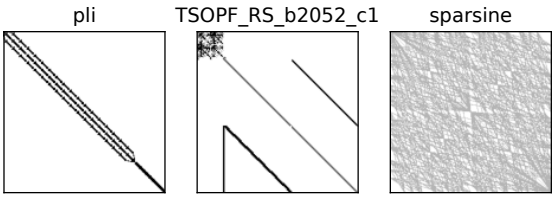
Figure 1: Overview of WACO. In the training dataset in (a), *(MTX, SS, TIME)* is the abbreviation for *(Sparse Matrix, SuperSchedule, Ground Truth Runtime)*. SuperSchedule defines the format and the schedule together.

- WACO uses an extremely fast search strategy, an Approximate Nearest Neighbor Search (ANNS), to retrieve the near-optimal format and schedule.
- We compared WACO against four state-of-the-art baselines, Intel MKL, BestFormat, TACO with a fixed format and schedule, and ASpT. WACO outperformed the best of four baselines by achieving 1.43×, 1.18×, 1.14×, and 1.27× average speedups on SpMV, SpMM, SDDMM, and MTTKRP, respectively.

1.1 Overview of WACO

Figure 1 shows an overview of WACO. We designed our cost model to predict the runtime of the program. The cost model takes a sparse matrix and a SuperSchedule, a unified template that defines the format and the schedule together, as inputs (Figure 1-(a), details in Section 4.1).

After training the cost model, WACO builds a KNN graph that helps with the search later. The KNN graph is built on program embeddings of uniformly sampled SuperSchedules (Figure 1-(b), details in Section 4.2).



Name	Rows	Cols	Nonzeros	Density
pli	22,695	22,695	1.35M	0.26%
TSOPF	25,626	25,626	6.76M	1.03%
sparsine	50,000	50,000	1.55M	0.06%

Figure 2: Sparse matrices used for the motivation.

Finally, when the input matrix comes in, WACO uses a novel search strategy, an approximate nearest neighbor search (ANNS), to search for the optimal format and schedule for a given input sparse matrix. ANNS repeats the picking of the next candidate SuperSchedule using a KNN graph and receives the candidate's predicted runtime as feedback until it converges to a locally optimal SuperSchedule (Figure 1-(c), details in Section 4.2).

2 Motivating Example

In this section, we will describe how the co-optimization can impact the performance of a sparse tensor program. In addition, we will show that the performance of a sparse tensor program strongly depends on the sparsity pattern. This demonstrates the strong need for an auto-tuning framework for sparse tensor programs.

2.1 Impact of the Co-optimization

Table 1 shows the impact of co-optimization in a sparse tensor program by comparing results of auto-tuning on three different tuning spaces: the format, the schedule, and both the format and the schedule. For the baseline, we used CSR, one of the most popular sparse matrix formats, with the default schedule generated by TACO. For the F., we only tuned the format while keeping the iteration order identical to the baseline, except that we made the traversing order to be concordant [43] with how the tuned format is aligned. For the S., we only tuned the schedule to transform the iteration order while keeping the format identical to the baseline (CSR). For the F.+S., we co-optimized both the format and the schedule.

Jointly optimizing the format and the schedule yields the most significant speedup for all the matrices in Figure 2, in contrast to the restricted search space. Restricting the tuning space to either choose the optimal format or the optimal schedule can miss optimization opportunities. Especially for TSOPF, co-optimization boosts the performance (2.02×), whereas considering only the format or the schedule yields a slight performance improvement (~1.1×).

2.2 Sparsity Pattern-Dependent Nature

The performance of sparse tensor programs is very sensitive to the sparsity pattern of the input matrix. No single format or implementation can show the optimal performance for all sparsity patterns, even for highly optimized handwritten libraries of experts. Table 2 demonstrates this nature. We ran a sparse matrix - dense matrix multiplication (SpMM) with the format and schedule optimized for different sparse matrices. As expected, the diagonal of the table shows the best performance because it is a result of the co-optimization that corresponds to the input matrix. A significant performance drop often occurs when other optimizations are applied.

Name	Base	F.	S.	F.+S.
pli	1×	1.03×	1.03×	**1.21**×
TSOPF	1×	1.11×	1.12×	**2.02**×
sparsine	1×	2.4×	1.02×	**2.5**×

Table 1: SpMM speedup over the base implementation after auto-tuning. The three rightmost columns represents different tuning spaces of a sparse tensor program. F., S., F.+S. mean format-only tuning, schedule-only tuning, and co-optimization.

Name	opt-pli	opt-TSOPF	opt-sparsine
pli	**1.21**×	0.82×	0.98×
TSOPF	1.14×	**2.02**×	0.96×
sparsine	0.81×	0.37×	**2.5**×

Table 2: SpMM speedup over the base implementation for different optimization methods. opt-*X* indicates the format and the schedule that are optimized for matrix *X* (as a result of F.+S. in Table 1).

These examples strongly indicate the need to co-optimize the format and schedules according to the input sparsity pattern. From the perspective of auto-tuning, three challenges stand out compared to dense applications. ❶ While considering the sparsity pattern, our framework should automatically decide ❷ which format to store the tensor in and ❸ which schedules should be applied to transform the iteration order. To address these challenges, the auto-tuner should understand the complex interactions among the sparsity pattern, the format, and the schedule.

3 Background

In this section, we describe how TACO generates codes that support various formats and iteration space transformations. Then, we describe existing sparsity pattern-aware cost models for sparse tensor program auto-tuning.

3.1 Tensor Algebra Compiler

TACO is a sparse tensor algebra Domain Specific Language (DSL) with an accompanying compiler that decouples the algorithm from the data representation and schedule [12, 25, 41]. Its algorithm is specified by an Einsum notation, for example, C[i,j] = A[i,k] * B[k,j] represents a matrix multiplication. Chou et al. introduced a format abstraction that describes how a sparse tensor is stored in different formats with coordinate hierarchies and level formats [12]. A sparse tensor can be viewed as a hierarchy of coordinates where each level is stored in one of the *level formats*. Chou et al. presented six level

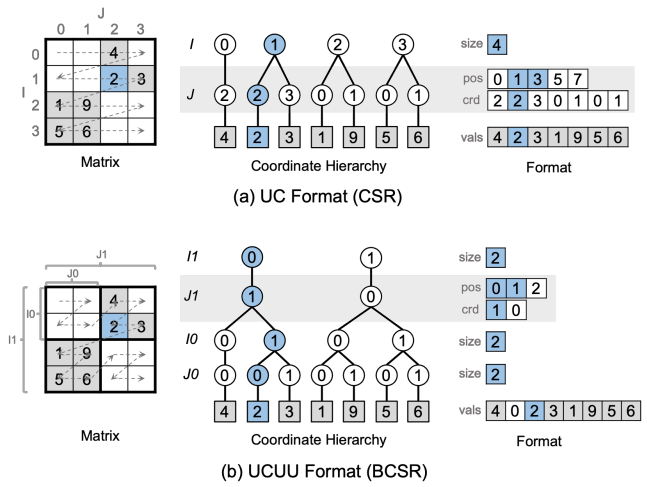


Figure 3: Identical tensors stored in two different formats, UC and UCUU. U and C mean Uncompressed and Compressed level format, respectively. The shaded level indicates the Compressed level format.

```
1:  for (int i = 0; i < 1024; i++)
2:   for (int kA = A2_pos[i]; kA < A2_pos[i+1]; kA++)
3:    int k = A2_crd[kA];
4:    for (int j = 0; j < 1024; j++) {
5:     int jC = i*1024 + j;
6:     int jB = k*1024 + j;
7:     C_vals[jC] += A_vals[kA] * B_vals[jB];
8:    }}}
```

Transforming loop via compute schedule

```
.split(j,j1,j0,64)
.reorder(i,j1,k,j0)
.parallelize(i,24,8)
```

```
1:  #pragma omp parallel for num_threads(24) schedule(dynamic,8)
2:  for (int i = 0; i < 1024; i++) {
3:   for (int j1 = 0; j1 < 16; j1++) {
4:    for (int kA = A2_pos[i]; kA < A2_pos[i+1]; kA++) {
5:     int k = A2_crd[kA];
6:     for (int j0 = 0; j0 < 64; j0++) {
7:      int jC = i*1024 + j1*64 + j0;
8:      int jB = k*1024 + j1*64 + j0;
9:      C_vals[jC] += A_vals[kA] * B_vals[jB];
10:    }}}}
```

Figure 4: Loop transformation of C[i,j]=A[i,k]*B[k,j] by schedules. It changes the traversal order of the iteration space. A[i,k] is stored in the UC format.

formats to represent various formats, but we will mainly focus on two level formats, *Uncompressed* and *Compressed*.

Format abstraction. Figure 3 illustrates how the CSR and BCSR formats are represented in the format abstraction. Any sparse tensor can be viewed as a coordinate hierarchy, which is a tree where each node contains nonzero coordinates at the level. The order of the levels ($I \rightarrow J$ or $I1 \rightarrow J1 \rightarrow I0 \rightarrow J0$) indicates the order in which tensor is stored (e.g., row-major or column-major). By specifying all level formats in the hierarchy, we can define a complete format. A level format determines what physical storage is used to store the coordinates of that level. An Uncompressed(U) level format stores the dimension (N) of the level and encodes a dense coordinate interval $[0, N)$. A Compressed(C) level format stores only the coordinates that have non-zeros by explicitly storing coordinates that appear in the hierarchy. A combination of level splitting, level reordering, and level format selection can create tens of thousands of data representations. For instance, the coordinate hierarchy in Figure 3-(b) can have a total of $4! * 2^4$ representations, where $4!$ indicates the number of possible level orders and 2^4 indicates the number of level format choices. More formats can be formulated depending on the number of levels in the hierarchy.

Iteraion space transformation via schedules. In addition to format abstraction, schedules decide how to traverse the tensor stored in a particular format by transforming the iteration space. For example, as shown in Figure 4, the `split` schedule splits a specified loop level into two nested levels, `reorder` specifies the order of the nested loops, and `parallelize` controls the load-balancing across multiple threads. A good choice of transformations enables parallelism and/or better data locality (e.g., register/cache blocking). In sparse computation, however, such loop transformation must be chosen deliberately while considering the format. For instance, if a loop order is discordant [43] with how the format is ordered, its generated code may involve an inefficient traversal routine such as a binary search over the Compressed level format. Thus, an auto-tuner must understand the coupled behavior between the format and the traversal order of the iteration space.

3.2 Cost Model for Auto-Scheduling

In scheduling languages [39], auto-scheduling is the task that finds the best schedule for a given input [10, 35, 51]. Auto-scheduling mainly has two parts. The first part is a cost model that quickly predicts the performance of the program, and the second is a search strategy that finds the best schedule according to the cost model. Although the actual hardware measurement can be used as a cost, it is very time-consuming, so designing an efficient and accurate cost model is crucial. For the sparse tensor program, understanding the sparsity pattern is the most critical design consideration of the cost model.

3.2.1 *Sparse Tensor Feature Extraction*

When designing the cost model of a sparse tensor program, two methods of extracting the features of a sparse tensor have been commonly used: ❶ human-crafted features [27, 40] and ❷ a convolutional network over downsampled tensors [42, 48].

Human-crafted features. A feature vector is designed manually by considering the statistical properties of tensors. Typical features are the total number of non-zeros, the mean or variance of the number of non-zeros per row, and format-specific features such as the average distance from the diagonal for DIA format. Nevertheless, the usefulness of human-crafted features for determining the accuracy is unknown. The features also have to be manually redesigned whenever a new format to be considered.

Convolutional neural network (CNN). Another approach uses a CNN to extract the features by viewing a sparse tensor as an image. A sparse tensor can have many different shapes, but since the CNN is limited to taking a fixed-size shape as an input, the

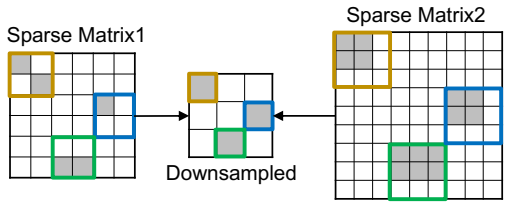


Figure 5: Different sparse tensors downsampled into the same 3×3 tensor.

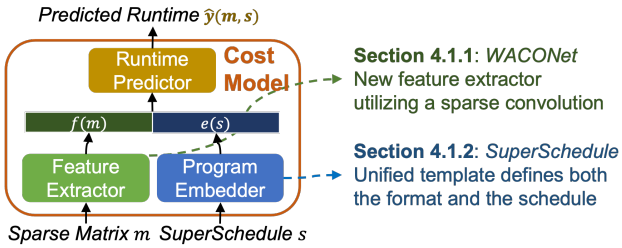


Figure 6: Overview of WACO's cost model. It predicts the program's runtime by taking a sparse matrix and SuperSchedule as inputs. SuperSchedule contains information on both the format and the schedule.

sparse tensor is downsampled into a fixed shape. Figure 5 illustrates how tensor downsampling works for arbitrary sparse tensors. In practice, the sparse tensors are usually downsampled to 128x128. To provide additional information to the CNN, a non-zero location in a downsampled tensor may contain a corresponding number of non-zeros in the original sparse tensors. However, as the shape of the sparse tensor increases, downsampling leads to a significant loss of the information on the local pattern. For example, while the Sparse Matrix2 in Figure 5 only has dense blocks, both matrices are downsampled into the same matrix. In addition, there are real-world sparse tensors with shapes in the millions scale, which cannot be helped by downsampling.

The aforementioned methods have deficiencies in accurately extracting the features of a sparsity pattern. In the case of human-crafted features, it is impossible to manually design all the format-specific features in TACO's format abstraction. In the case of downsampling, it only works for small sparse tensors or it will lose significant information, which often leads to sub-optimal decisions.

4 Workload-Aware Co-Optimization

In this section, we introduce WACO, an auto-tuning framework for sparse tensor programs. WACO automatically searches for the best format and schedule for a given sparse matrix from among what TACO can generate.

First, we will describe how WACO uses a novel cost model that understands a complex interaction of the sparsity pattern, format, and schedule (Figure 1-(a)). Then, we will explain how WACO efficiently searches over the large search space using a novel search strategy, ANNS (Figure 1-(b,c)).

4.1 Cost Model Design

Our cost model has three parts (Figure 6). The first part, the feature extractor, captures the sparsity pattern of the input matrix. The second part, the program embedder, understands the coupled behavior of the format and the schedule. Finally, the runtime predictor predicts the runtime through multiple linear-ReLU layers by concatenating the results of the previous parts.

4.1.1 *Feature Extractor: WACONet*

Challenges. As described in 3.2.1, extracting features of a sparsity pattern is non-trivial. The core idea of our approach is to use a sparse CNN to learn good features. We propose a novel feature extractor, *WACONet*, based on a sparse CNN with a novel network architecture. In Section 5.3, our evaluation shows that WACONet improves the training and validation loss by roughly 50% when compared to a conventional CNN feature extractor.

Exploring different architectures. We first tried a conventional CNN that treated a sparse matrix as a dense matrix, where all levels were stored in the Uncompressed format. However, as the shape of the matrix grew, it ran out of computational resources very quickly. For example, if there is a sparse matrix of shape $10^5 * 10^5$, it will need a total of $4 * 10^{10}$ bytes (assuming 4 bytes single-precision) regardless of the number of non-zeros. Another approach that we tried is using a recurrent neural network by viewing a sparse tensor as a sequence of coordinates. However, since the sequence length (a number of non-zeros) is in the millions scale, the recurrent neural network cannot remember everything and easily forgets the early sequences. It is also difficult to decide in which order to put the coordinate sequences such as the row-major or the column-major. We ended up using CNN for our feature extractor, but instead of a dense convolution, we used a sparse convolution on the raw sparse matrix itself.

Sparse convolutional layer. A sparse convolutional layer [17] (often called as submanifold sparse convolution) performs a convolution operation over a sparse input. There is a marked difference between the sparse convolution and conventional convolution. While conventional convolution operates over all the input activations, a sparse convolution operates only when the filter's center is located on a non-zero input activation (Figure 7). This peculiar behavior prevents the activations from becoming dense as the layers are stacked, thus keeping the computation relatively cheap. However, this behavior also has an issue when the non-zeros are distributed far apart. As shown in Figure 8-(a), this behavior can only capture the local pattern but not the global pattern because the non-zeros are not close enough to propagate information. Sparse convolution has shown a powerful ability to understand 3D point clouds when their non-zeros are close enough [13]. However, real-world sparse matrices often have a distant non-zero distribution, so we need to design a network architecture that addresses this issue while utilizing the advantage of sparse convolution.

WACONet. We propose *WACONet*, a novel sparse CNN architecture that learns the rich features of a sparsity pattern effectively (Figure 9). Except for the first layer, we used a strided convolution with a filter size of 3x3 for every sparse convolutional layer. Multiple stacks of strided convolution help distant non-zeros because a strided behavior forces the receptive field to increase (Figure 8-(b)).

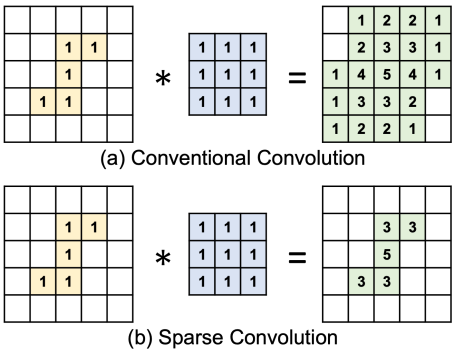


Figure 7: Difference between conventional convolution and sparse convolution.

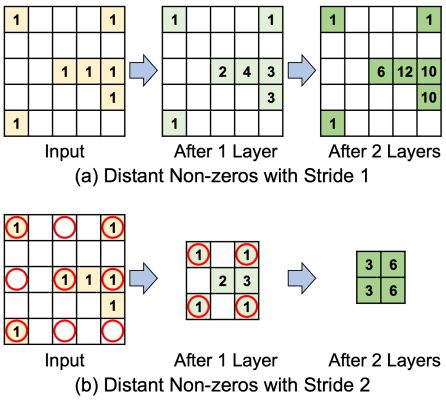


Figure 8: When non-zeros are distributed far apart, the receptive field does not increase even with multiple layers if the stride of sparse convolution is 1. Filter's center is located at red circles in the stride of 2 as used in WACONet.

Due to the limited memory size of the GPU, the number of channels in the sparse convolutional layer is small (32) to fit a sparse matrix with a large number of non-zeros up to 10 million, unlike in a typical vision CNN model (e.g., 256 and 512). To compensate for decreased network capacity due to a limited number of channels, WACONet concatenates all 14 intermediate results after the global average pooling rather than using a result of the final layer.

WACONet minimizes the loss of information of the sparsity pattern because it takes a raw sparse matrix as an input without any downsampling. Due to the nature of the convolution operation, a small filter (3x3) recognizes the local pattern, and a global pattern is captured while passing through multiple strided layers. In addition, WACONet can be easily extended for high-dimensional sparse tensors by simply changing the dimension of the filter. In section 5, we demonstrate that WACONet extracts the rich features for both 2D and 3D sparse tensors.

4.1.2 *Program Embedder : SuperSchedule*

We will now consider the second part of the cost model, a program embedder. In a dense tensor program, a program embedder only needs to encode a traversal order of the iteration space reflected by the low-level loop abstract syntax tree [4, 10, 35]. In a

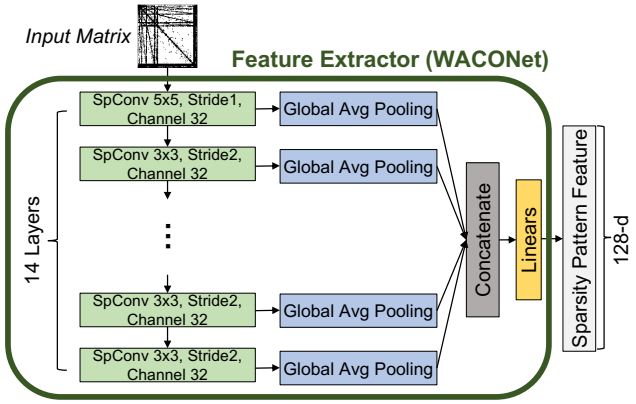


Figure 9: Network architecture of WACONet. We omitted non-linear activation layers in the figure.

sparse tensor program, however, a program embedder must encode both the traversal order and the format to accurately understand the coupled behavior for the joint optimization.

Challenges. Encoding a loop order is non-trivial because the number of levels in a nested loop varies due to the schedule `split`. The search space expands as well after splitting, as seen in Figure 4, where the number of loop reorderings increased to 4! from 3!. To deal with the variableness due to the `split`, we adopted a template-guided auto-scheduling [10]. In addition, our template specifies both the format and the schedule and creates the program embedding directly on top of that template.

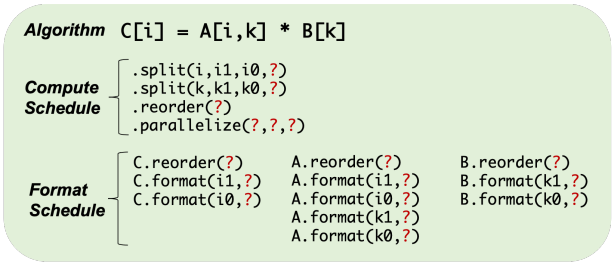
SuperSchedule. The unified schedule template, which we call *SuperSchedule*, defines the format and the schedule at the same time. Figure 10-(a) shows how the SuperSchedule template is defined in a matrix-vector multiplication (MV). The SuperSchedule consists of a compute schedule and a format schedule. A compute schedule defines the traversal order of the iteration space and a format schedule that defines how tensors will be stored. While `reorder` in the format schedule determines the level order of the tensor (e.g., the row-major or the column-major), `reorder` in the compute schedule decides the traversal order of the tensors.

One observation is that a schedule template that already has multiple `splits` can be reduced into a schedule that has fewer `splits`. This reduction can be done by specifying the split size as 1. To support this, the compute schedule splits each index(i and k) once, making the MV algorithm (C[i] = A[i,k] * B[k]) a split MV algorithm (C[i1,i0] = A[i1,i0,k1,k0] * B[k1,k0]). Within this SuperSchedule, we can sample all the schedules from

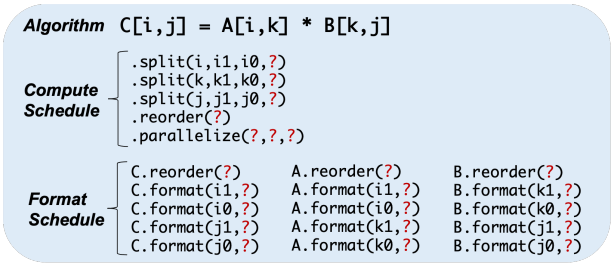
(1) C[i1] = A[i1,k1] * B[k1]
(2) C[i1,i0] = A[i1,i0,k1] * B[k1]
(3) C[i1] = A[i1,k1,k0] * B[k1,k0]
(4) C[i1,i0] = A[i1,i0,k1,k0] * B[k1,k0]

by appropriately choosing the `split` size as 1.

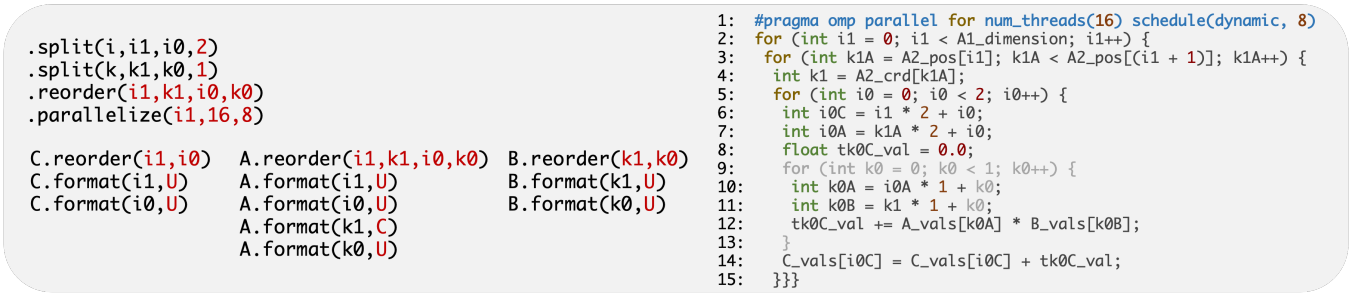
From these split algorithms, SuperSchedule can also derive various formats. For instance, the UC format in Figure 3 can be derived by choosing both `split` sizes as 1 and specifying level formats as UC according to the level order of i1 and k1. Similarly, the UCUU format can be derived by choosing both `split` sizes greater than



(a) Matrix-Vector Multiply(MV) SuperSchedule



(b) Matrix-Matrix Multiply(MM) SuperSchedule



(c) Sampled Schedule from MV SuperSchedule and its generated code.

Figure 10: (a,b) MV/MM SuperSchedules. (c) The sampled schedule showed how C[i1,i0] = A[i1,k1,i0] * B[k1] with a BCSR format can be sampled from the SuperSchedule by choosing the k split size as 1. The shaded lines in the generated code indicate those can be ignored due to the split size 1. The SuperSchedule for SDDMM(D[i,j] = A[i,j] * B[i,k] * C[k,j]) can also be defined similarly.

Schedule	Parameters	Description
`split`	[1, 2, ..., 32768]	Split Size
`reorder`	$P(i1, i0, k1, k0)$	Loop Order
`parallelize`	$[i1, i0]$	Parallelized Index
	[24, 48]	# Threads
	[1, 2, ..., 256]	OMP Chunksize
`C.reorder`	$P(i1, i0)$	Level Order of C
`A.reorder`	$P(i1, i0, k1, k0)$	Level Order of A
`B.reorder`	$P(k1, k0)$	Level Order of B
`format`	[U, C]	Level Format

Table 3: MV SuperSchedule parameters. $P()$ indicates a permutation of indices. For `parallelize`, we used the OpenMP work-sharing policy (`#pragma omp parallel for schedule(dynamic, chunksize)`).

1 and specifying the level format as UCUU according to the level order.

SuperSchedule is a superset of all possible schedules under a fully split algorithm. For example, the MV and MM SuperSchedule in the Figure 10 can represent a total of 4 and 8 split algorithms, respectively, but SuperSchedule can represent more algorithms depending on how many `splits` are defined. We chose a maximum of one `split` per dimension since we have found out that more than one `split` yields diminishing returns.

Network architecture. Such a template-based schedule allowed us to embed the program more easily. Rather than extracting each loop's features from the low-level loop abstract syntax tree, SuperSchedule allowed us to embed the format schedule and the compute schedule directly from the parameters of the template. Table 3 describes each schedule and its possible parameter choices used in our evaluation. All the parameters are categorical except for the `reorder`, which take a permutation of indices. Our program embedder (Figure 11) takes parameters of a SuperSchedule and outputs the program embedding. It first calculates the embeddings of each parameters. Each categorical parameter passes a learnable lookup table (green box) that maps the one-hot categorical parameter to a high-dimensional real-valued vector. Each permutation parameter is converted into a corresponding permutation matrix and passes multiple linear-ReLU layers (orange box). When the embeddings for the all schedule parameters are calculated, they are concatenated and pass multiple linear-ReLU layers into the final program embedding.

4.1.3 *Training Cost Model*

Data generation. Our training dataset was a set of tuples *(Sparse Matrix, SuperSchedule, Ground Truth Runtime)*. We designed our dataset to include various sparsity patterns. We augmented the 2,893 real-world sparse matrices in the SuiteSparse matrix collection [14] by arbitrarily resizing them into 21,400 sparse matrices while restricting the number of rows to less than 131,072 and the number of non-zeros to less than 10 million.

For each matrix, we randomly sampled 100 formats and schedules from the SuperSchedule. Then, we generated a corresponding code using TACO for each sample, repeated the program for 50 rounds, and reported the median time. We excluded formats and schedules

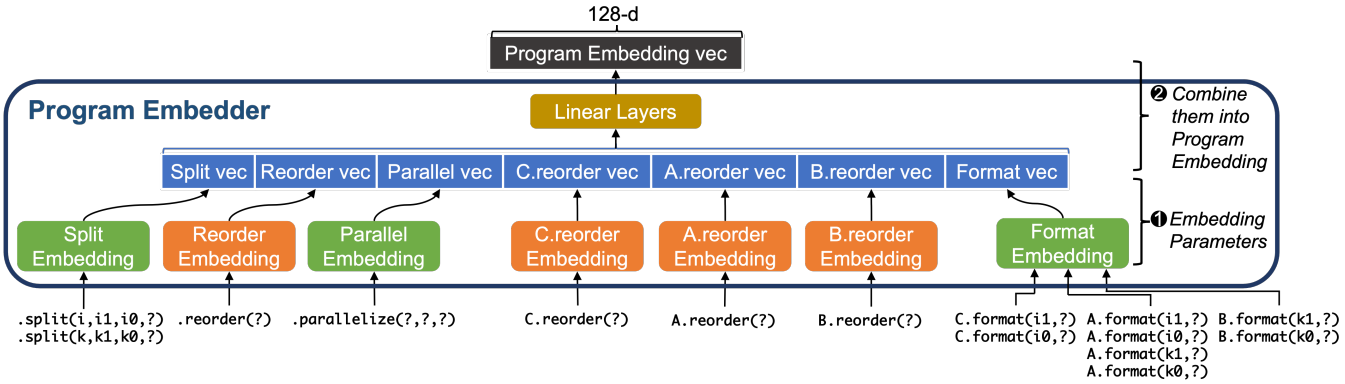


Figure 11: Network architecture of the program embedder. The parameters of the SuperSchedule are used as the inputs. The green embedder takes a categorical parameter, while the orange embedder takes a permutational parameter.

that take more than a minute. We repeated this process for the three algorithms used in evaluations (SpMV, SpMM, and SDDMM), so we collected about 6 million tuples (2 million tuples for each algorithm). Collecting the dataset took two weeks with 10 computing nodes. During the training, we divided the total dataset into the training dataset and the validation dataset at an 80:20 ratio.

Training objective. For a given input sparse matrix m_i and SuperSchedule s_j, the goal of our cost model $\hat{y}(m_i, s_j)$ is not to accurately predict the ground truth runtime y_{ij}. Instead, we want our cost model to learn the ranking of different SuperSchedules. Therefore, we used a pairwise ranking loss [8] to reflect the relative order of performance of the schedules instead of using the L1 or L2 loss.

$$L = \sum_{m_i} \sum_{(s_j, s_k)} sign(y_{ij} - y_{ik}) * \phi(\hat{y}(m_i, s_j) - \hat{y}(m_i, s_k))$$

where $sign(x)$ is 1 if $x > 0$, or 0 otherwise. $\phi(x)$ can be defined as various functions, such as the hinge function $max(0, 1 - x)$ or the logistic function $log(1 + e^{-x})$. We adopted the hinge function for our model. We used a SuperSchedule (s_j, s_k) batch size of 32 for each sparse matrix m_i and an Adam optimizer [24] with a learning rate of 0.0001.

4.2 Efficient Schedule Search via Nearest Neighbor Search

Besides the design of the cost model, a search strategy is also an essential component of auto-scheduling. Many auto-schedulers or tuners rely on black-box optimization algorithms to find the best parameters in the schedule template [3, 18, 50].

Challenges. Traditional black-box optimization algorithms are often slow because besides evaluating black-box (cost model), they must manage the metadata required for optimization. For example, Bayesian optimization trains a surrogate model internally that facilitates the procedure during the search. To speed up the search, we cast the auto-scheduling problem as a Nearest Neighbor Search (NNS) [36]. We then exploit an existing high-performance NNS library to search for the optimal parameters of the SuperSchedule. In our experiment, the proportion of evaluating costs in the whole search was only 3.9% and 8.1% on two famous black-box optimizers, HyperOpt [6] and OpenTuner [3], while our search strategy improved the proportion to 93.9%.

4.2.1 *Relationship between auto-scheduling and NNS*

Here, we will show that auto-scheduling can be reduced into the NNS. The definition of the NNS is as follows :

Definition 4.1 (Nearest Neighbor Search). *Suppose we have a dataset $S = \{x_1, x_2, ..., x_n\}$ where $x_i \in \mathbb{R}^{d_s}$. Nearest Neighbor Search retrieves a point $p \in S$ which is nearest to a given query $q \in \mathbb{R}^{d_q}$.*

Here, *nearest* can be defined with various metrics such as the Euclidean distance or cosine similarity. Then NNS will retrieve the point that *minimizes* a distance metric for a given query. In terms of auto-scheduling, the main objective is to find a schedule s that *minimizes* the predicted runtime $\hat{y}(m, s)$ for a given input matrix m. Therefore, we can cast an auto-scheduling as an NNS by setting the dataset S to be all the formats and the schedules in SuperSchedule template, and the **query q** to be the **input matrix m**. If we define a **distance metric** as a **cost** $\hat{\mathbf{y}}(\mathbf{m}, \mathbf{s})$, NNS will retrieve the best SuperSchedule s for a given input matrix m that minimizes the $\hat{y}(m, s)$.

Approximate Nearest Neighbor Search. Retrieving an exact nearest neighbor requires exhaustive distance calculations all over the points in S, which is intractable. In practice, Approximate Nearest Neighbor Search (ANNS) [28] has been widely used instead of an exact NNS. For a given query, ANNS cleverly searches the subset of S that speeds up the search while guaranteeing high recall. While there are several approaches to achieve ANNS, we used a graph-based algorithm.

4.2.2 *Graph-based ANNS*

Graph-based ANNS for auto-scheduling has two phases: building a KNN graph and searching on the KNN graph, as shown in Figure 1-(b,c) and Figure 12. The first phase builds a KNN graph whose vertex is the SuperSchedule, and the edge between two vertices is connected only if two program embeddings of the vertices are top-K closest to each other in the l_2 distance. The second phase starts once the query (the input matrix m) comes in. In the second phase, ANNS starts retrieving the schedule s in the graph that minimizes

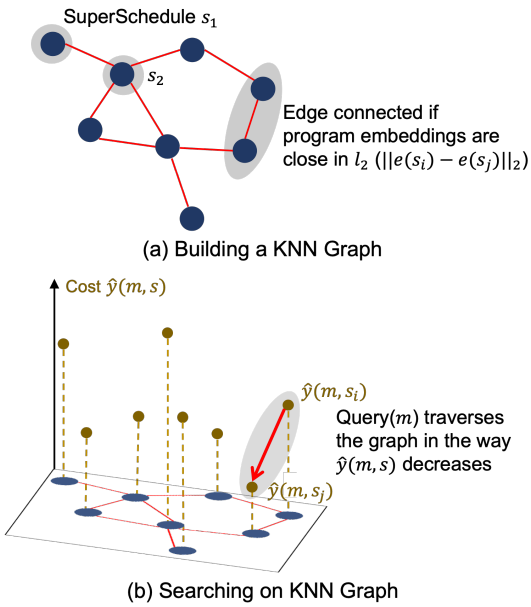


Figure 12: Our search strategy via ANNS. In the stage of building the KNN graph, the graph is built by connecting the edge between the schedules with close embeddings in the Euclidean distance. During searching, a query (input matrix m) traverses the graph in the direction predicted runtime $\hat{y}(m, s)$ minimizes.

the cost $\hat{y}(m, s)$ by traversing the KNN graph in the direction at which the cost decreases. ANNS can search efficiently because it merely traverses the pre-built KNN graph that guides the direction, whereas other black-box optimizations require expensive metadata updates.

Discussions. The distance metric used in each phase is different. The l_2 distance between two program embeddings of SuperSchedules is used in the building phase, and the cost itself is used in the searching phase ($\|e(s_i) - e(s_j)\|_2$ vs. $\hat{y}(m, s)$). The reason for using completely different metrics is that the KNN graph built upon an l_2 distance has a property that guarantees retrieval of top-K candidates at any generic distance ($\hat{y}(m, s)$ in our case) in the searching phase, while promising high recall [44]. In Section 5.4, we empirically show that graph-based ANNS efficiently and accurately retrieves the optimal SuperSchedule for a given input matrix.

One intuitive explanation of graph-based ANNS is a gradient-based search over discretized space. In fact, our cost model based on a neural network is differentiable, which means we can calculate the gradient for it. Therefore, when searching for the best SuperSchedule, we can actually find the local optima SuperSchedule using the first-order iterative optimization algorithm such as gradient descent. However, there is no guarantee that the local optima we found is a valid encoding of parameters. Specifically, most gradient-based searches will end up with invalidly encoded parameters because we encoded the categorical parameter of SuperSchedule as a one-hot vector and the permutation parameter as a permutation matrix. On the other hand, if we build a KNN graph with valid SuperSchedules, we can think of the ANNS on the KNN graph as a gradient-based search over valid encodings. Projected gradient descent [18] is another way to resolve this, but it was not able to find a good local minimum in our experiments.

Implementation details. In practice, ANNS libraries build a variant of the KNN graph and traverse it with complicated heuristics to improve the search efficiency. We implemented our search strategy using a state-of-the-art graph ANNS algorithm, HNSW [31]. Although HNSW can support up to a billion-scale graph, it is intractable to build a graph with all formats and schedules in the SuperSchedule as it contains an astronomical number of parameter choices. Therefore, we built the graph with the SuperSchedules which appeared in our training dataset.

5 Evaluation

5.1 Experimental Setup

Algorithms. We choose four sparse tensor algebra algorithms for our evaluation. All the algorithms were performed with single-precision data.

- **SpMV(C[i] = A[i,k] * B[k]):** This multiplies sparse matrix(A) by dense vector(B) and stores the product in dense vector(C).
- **SpMM(C[i,j] = A[i,k] * B[k,j]):** This multiplies sparse matrix(A) by dense matrix(B) and stores the product in dense matrix(C). We set the number of columns of dense matrices (|j| in B, C) at 256, and forced both dense matrices' level order to be row-major.
- **SDDMM(D[i,j] = A[i,j] * B[i,k] * C[k,j]):** This performs a sampled matrix multiplication of two dense matrices(B and C). The output matrix D and the input matrix A are sparse matrices. We set the dimension |k| in B, C at 256. We fixed B's level order to be row-major and C's level order to be column-major.
- **MTTKRP(D[i,j] = A[i,k,l]*B[k,j]*C[l,j]):** This performs a matricized tensor times Khatri-Rao product between a 3D sparse tensor(A) and two dense matrices(B and C). We set |j| at 16 and both dense matrices' level order to be row-major. We followed a prior work's approach [42] to generate the training dataset for 3D sparse tensors.

Baselines. We compare WACO with the following four state-of-the-art baselines. MKL and BestFormat are auto-tuning-based baselines. FixedCSR and ASpT are baselines with a fixed format and schedule.

- **MKL:** Intel MKL sparse BLAS routines [34] utilize an inspector-executor model that auto-tunes a computation on a fixed format. Because it does not support SDDMM and MTTKRP, we only compare it with SpMV and SpMM using the CSR format.
- **BestFormat:** BestFormat automatically selects the appropriate format among a handful candidates for a given sparsity pattern. The candidates were chosen by the five most frequently appearing formats among WACO's search results in the test matrices. We've used prior works' artifacts to predict the best format for a 2D sparse matrix [48] or a 3D sparse tensor [42].
- **Fixed CSR:** Fixed CSR is a code with a fixed format and schedule generated by TACO. We used the CCC format(CSF)

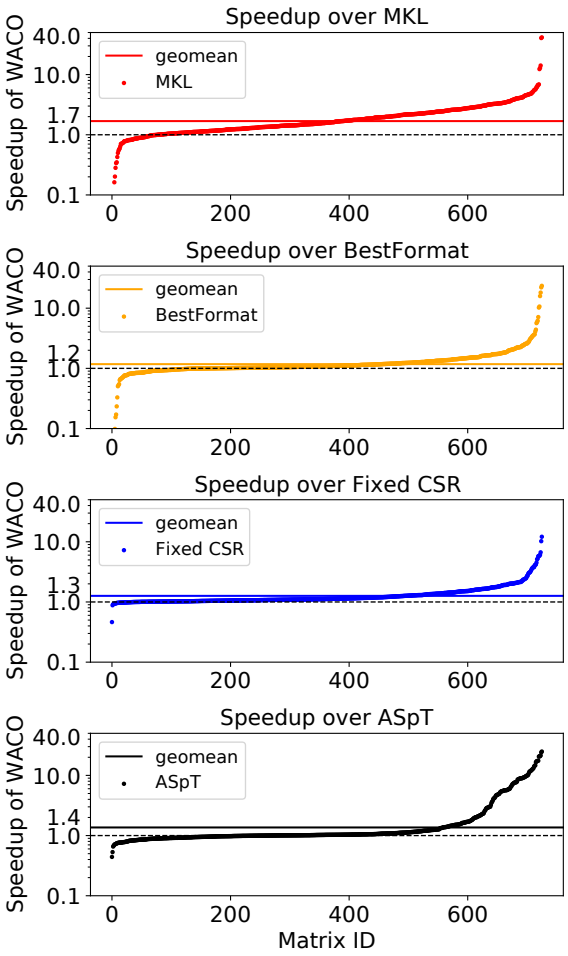


Figure 13: Performance comparison on SpMM.

for MTTKRP and UC format(CSR) for the rest. We set the OpenMP chunk size at 128, 32, 32, and 32 for SpMV, SpMM, SDDMM, and MTTKRP.

- **ASpT:** ASpT [19] is the state-of-the-art sparse format that directly reorders the sparse matrix to make dense regions. While ASpT is not limited to a specific algorithm, we only compare it only with SpMM and SDDMM because these are the only formats publicly released by the authors.

Implementations. We used TACO to generate the code for the best format and schedule that WACO has found. Then we compiled the generated code with `icc-2021.3.0` with `-march=native -mtune=native -O3 -qopenmp` options. All the experiments were conducted on a dual-socket, 24-core with 48 threads, 2.5 GHz Intel Xeon E5-2680 v3 machine with 30 MB of L3 cache per socket and 128 GB of main memory with Ubuntu 18.04.3 LTS. We used `numactl -interleave=all` to control the NUMA policy.

We implemented our cost model architecture using PyTorch and MinkowskiEngine for sparse convolution. We trained four separate models for each algorithm, and it took four days to train up to 70 epochs for each model on a single GPU, NVIDIA GeForce RTX 3090 24GB.

	Auto-tuning based baselines	
	vs. Format-only	**vs. Schedule-only**
SpMV	1.43x	2.32x
SpMM	1.18x	1.68x
SDDMM	Not Impl.	Not Impl.
MTTKRP	1.27x	Not Impl.

Table 4: Geomean speedup of WACO over other auto-tuners. Format-only and Schedule-only auto-tuner correspond to the BestFormat and MKL, respectively.

	Fixed Implementations	
	vs. Fixed CSR	**vs. ASpT**
SpMV	1.54x	Not Impl.
SpMM	1.26x	1.36x
SDDMM	1.29x	1.14x
MTTKRP	1.35x	Not Impl.

Table 5: Geomean speedup of WACO over other state-of-the-art implementations with a fixed format and schedule.

5.2 Performance Results

We first evaluated the performance of the format and the schedule that WACO has found. Experiments were conducted on 726 real-world sparse matrices from SuiteSparse that were not included in the training dataset. We picked matrices that had less than 10 million non-zeros and less than 100,000 rows. Among the top-10 SuperSchedules selected by WACO according to the cost model, we report the fastest after we measured them on the hardware. Before we explain the results in detail, the geomean of the speedups on each algorithm are shown in Table 4 (vs. auto-tuners) and Table 5 (vs. fixed implementations). Overall, WACO performed better than the baselines as it successfully found a specialized format and schedule together for each sparse matrix. This improvement is not limited to a specific algorithm; WACO can find a better format and schedule for all four algorithms.

Figure 13 show the speedups of WACO over four baselines across the test matrices on SpMM. The y axis indicates the speedup of WACO against baselines. All x axes of figures are sorted according to the speedup. The dots below the $y = 1.0$ show matrices in which the baseline performed better than WACO. For MKL and BestFormat, there are more matrices below this line than compared to other baselines because they are able to adopt a larger portion of the space though still not as much as of WACO. Thus, auto-tuning based baselines perform better at a few patterns when they find a better format or schedule than WACO.

5.2.1 *Discussion on Speedup*

We further analyze the source of the speedups on SpMV, SpMM, and SDDMM. We picked the matrices with a speedup > 1.5× than the Fixed CSR and classified the speedup factors into five categories. Table 6 shows these categories and their proportions.

SpMV. First, half of the matrices benefitted from choosing the appropriate OpenMP chunk size, which controls the load balancing across multiple processors. Another half benefits from storing the

Factor	SpMV	SpMM	SDDMM
OpenMP Chunk Size	51%	66%	47%
Dense Block >50% Filled	30%	26%	15%
Dense Block <50% Filled	19%	-	-
Sparse Block	-	8%	-
Parallelize over Column	-	-	38%

Table 6: Speedup analysis of WACO. The number shows the corresponding factor's percentile among matrices that had a speedup of over 1.5× than the Fixed CSR.

```
1:  const int b = ?
2:  for (int i1 = 0; i1 < A1_dimension; i1++) {
3:   for (int kA = A2_pos[i1]; kA < A2_pos[i1+1];kA++) {
4:    int k = A2_crd[kA];
5:    for (int i0 = 0; i0 < b; i0++){
6:     int i0A = kA * b + i0;
7:     int i0C = i1 * b + i0;
8:     C_vals[i0C] += A_vals[i0A] * B_vals[k];}}}
```

b=8

```
vfmadd231ss xmm0,xmm8,[r8+rcx*4]
vfmadd231ss xmm1,xmm8,[4+r8+rcx*4]
vfmadd231ss xmm2,xmm8,[8+r8+rcx*4]
vfmadd231ss xmm3,xmm8,[12+r8+rcx*4]
vfmadd231ss xmm4,xmm8,[16+r8+rcx*4]
vfmadd231ss xmm5,xmm8,[20+r8+rcx*4]
vfmadd231ss xmm6,xmm8,[24+r8+rcx*4]
vfmadd231ss xmm7,xmm8,[28+r8+rcx*4]
```

b=16

```
vfmadd213ps ymm0,ymm2,[r8+rdx*4]
vfmadd213ps ymm2,ymm1,[32+r8+rdx*4]
```

Figure 14: icc generated assembly for SpMV with the UCU format. b decides the size of the one-dimensional dense block. icc starts to use the AVX instructions(vfmadd213ps) from b=16.

matrix into a dense blocked format which exploits the register reuse in the dense block. A dense blocked format can be represented UCU or UCUU in a format abstraction like Figure 3-(b). One counter-intuitive factor is the speedup of a matrix with the non-zeros filling less than 50% of the dense block. Storing such matrices into a dense blocked format usually results in memory increase due to unnecessary zeros. Nevertheless, a speedup occurs because of the heuristic decision in the Intel `icc` compiler regarding utilizing SIMD instructions. As shown in the Figure 14, we found out that `icc` starts to exploit the SIMD instructions when the block size is larger than 16. It is surprising to see that WACO learned the compiler's heuristics and intentionally chose the larger block size to utilize the vector registers despite the memory increase.

SpMM. Like SpMV, most matrices benefit from better load balancing by choosing an appropriate chunk size. Other than that, some matrices benefit from a unique format, which we call a *sparse block format*. Compared to the dense block format where the level format of the inner split level is Uncompressed (e.g., UCU or UCUU), a sparse block format stores the inner level into the Compressed format (e.g., UUC). Splitting the level into the Compressed level format with a large split size helps improve the cache locality in SpMM. For instance, the LLC miss rate was reduced to 7% from 36% and the performance improved about 2.5× when we stored sparsine (Figure 2) into the $k1(U) \rightarrow i(U) \rightarrow k0(C)$ format by splitting k by 16,384.

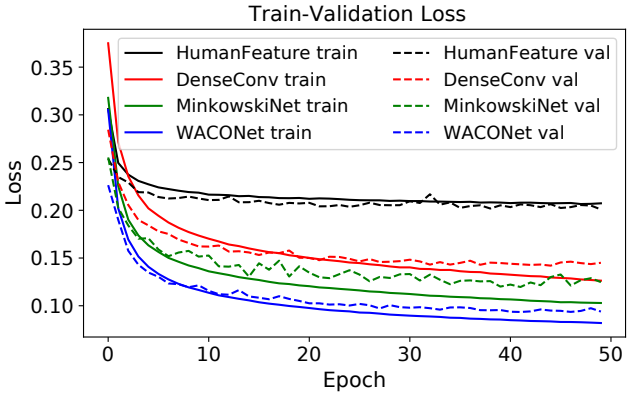


Figure 15: Train-validation losses of the SpMM cost models using four different feature extractors.

SDDMM. Other than better load balancing and a dense block format, SDDMM can take an advantage of the use of a column-major format. One difference between SDDMM and other algorithms is that it is safe to parallelize both rows and columns of the sparse matrix in SDDMM. For SpMV(C[i] = A[i,k] * B[k])) or SpMM(C[i,j] = A[i,k] * B[k,j]), it is inefficient to parallelize over the column of the sparse matrix(k in A[i,k]) because the reduction occurs along that dimension. Therefore, WACO flexibly chose the row-major or column-major format without any restriction in the parallelizing dimension on SDDMM.

5.3 Cost Model Exploration

We conducted the experiment to test how effectively our feature extractor learns meaningful features of the sparsity pattern. The train-validation losses of four alternative cost models, each of which uses a different feature extractor, are shown in Figure 15. HumanFeature uses three simple statistics of the sparsity pattern, (# rows, # cols, and # non-zeros). DenseConv [48] uses a conventional CNN after downsampling an input matrix into 256×256. MinkowskiNet [13] is a popular deep learning model based on sparse convolution layers for 3D point clouds. Due to the limited size of GPU memory, we reduced the number of channels in MinkowskiNet to support a matrix with 10 million non-zeros. WACONet is our feature extractor that described in Section 4.1.1. There is a marked difference between the HumanFeature and the remaining three networks using convolution. WACONet and MinkowskiNet, networks that use a sparse convolutional layer, learn better than DenseConv because DenseConv causes the loss of pattern information during downsampling as explained in Section 3.2.1. Finally, as the strided convolution accommodates distant non-zeros, WACONet retrieved more meaningful features than MinkowksiNet.

5.4 Search Strategy Exploration

Different search strategies. We compared ANNS with two other black-box optimization search strategies, HyperOpt [6] and OpenTuner [3]. HyperOpt utilizes Bayesian optimization, and OpenTuner utilizes an ensemble of search techniques that use multi-armed bandit. For each search strategy, we ran 3,000 trials to search for the

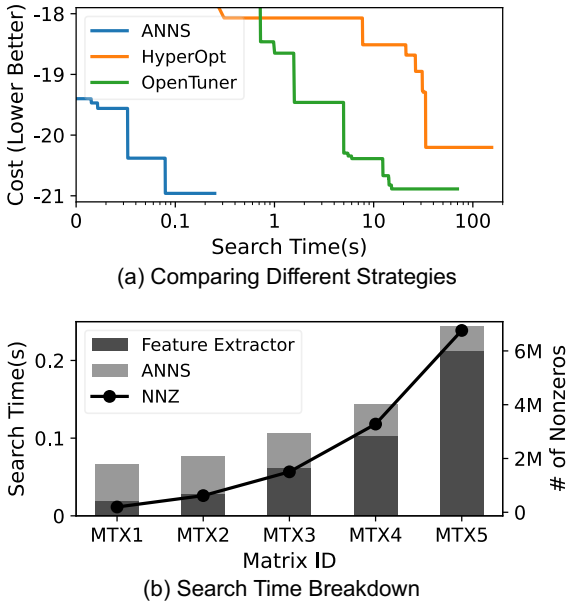


Figure 16: Exploring different search strategies and breaking down the search time of WACO on SpMM

optimal parameters of SuperSchedule on the SpMM cost model with a bcsstk29 matrix. As shown in Figure 16-(a), ANNS found the lowest cost within an equal number of trials. OpenTuner also found a comparable cost to that of ANNS, but the search time is much longer. We can summarize why ANNS is substantially faster than the others into three reasons. First, ANNS does not require any metadata update, which is common in machine learning-based black-box optimization, such as a training surrogate model in Bayesian optimization. ANNS can efficiently search for different formats and schedules merely by traversing the KNN graph. Second, a KNN graph memorizes the program embedding of each SuperSchedule(vertex) during the building phase. Thus, it does not have to run for the entire cost model; it only needs to run for the final part of the cost model (Figure 1-(c)). Finally, ANNS is implemented in C++, whereas most black-box optimization libraries are built on Python.

Search time breakdown. Since the sparsity pattern feature is reusable when calculating the cost of the different schedules, WACO does not run the feature extractor multiple times for an input matrix. Instead, the search can be divided into two phases (Figure 1-(c)): (1) extracting the sparsity pattern feature and (2) ANNS with the final part of the cost model. Figure 16-(b) shows the search time breakdown of five different matrices with varying numbers of non-zeros. When the number of non-zeros is less than 1.5 million, ANNS dominates the entire search time, but the feature extractor becomes more expensive when the number of non-zeros increases. This is because the computational cost of sparse convolution depends on the number of non-zeros.

Speedup over FixedCSR		Trained on	
		Intel CPU	AMD CPU
Tested on	Intel CPU	**1.26x**	1.12x
	AMD CPU	1.08x	**1.21x**

Table 7: WACO's SpMM geomean speedup over FixedCSR with a cost model trained on same/different hardware.

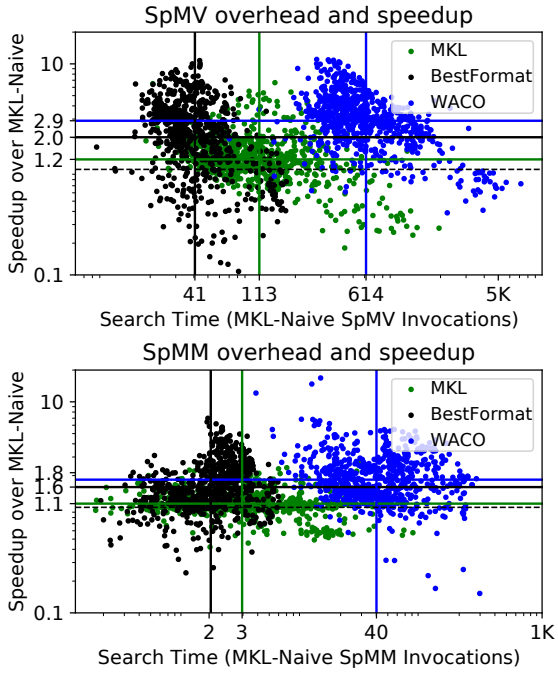


Figure 17: Tuning overhead of the MKL inspector-executor, the BestFormat, and the WACO. We compared all methods against the auto-tuning disabled MKL (MKL-Naive).

5.5 Generalization on Other Hardware

WACO's cost model is somewhat hardware-specific as it does better when trained on target hardware. However, the cost model does transfer general optimization patterns between hardware. For example, a sparsity pattern with skewed non-zero distribution will generally prefer a fine-grained load-balancing. To demonstrate this generalization, we trained a SpMM cost model on different hardware and compiler: 8-core(16 threads) AMD EPYC 7R32 with a 16MB L3 cache and gcc-11. Data collection took about 4 days on 8 nodes, and training a cost model took about 3 days on a single GPU. Table 7 shows the speedup of WACO under 2×2 possible configurations. As expected, the diagonal of the table shows the best performance because the cost model is trained for the target hardware. WACO, in general, found a better format and schedule than a baseline with a model trained on a different hardware.

5.6 Search Overhead and Usage Scenarios

(a) SpMV		End-to-end Execution Time (in MKL-Naive SpMV calls)		
Label	N_{runs}	WACO	BestFormat	MKL
Initial Cost	0	821	277	**113**
PageRank [49]	50	838	302	**153**
WACO=MKL	1,546	1,356	**1,044**	1,356
WACO=BestFormat	3,627	**2,075**	**2,075**	3,028
GMRES [30]	517K	**180K**	257K	416K
Mesh sim. [26]	1.8M	**623K**	892K	1.4M

(b) SpMM		End-to-end Execution Time (in MKL-Naive SpMM calls)		
Label	N_{runs}	WACO	BestFormat	MKL
Initial Cost	0	46	7	**3**
WACO=MKL	115	109	**80**	109
WACO=BestFormat	412	**271**	**271**	382
GNN [7]	10K	**5,511**	6,432	9,224
Pruned NN [16]	1.0M	**546K**	642K	922K

Table 8: Real-world applications that require repetitive (a) SpMVs and (b) SpMMs. Green cells indicate that the corresponding auto-tuner wins. Initial cost is computed as $T_{tuning} + T_{formatconvert}$, but only T_{tuning} for MKL.

Tuning overhead. Because the program auto-tuning pays for the search or tuning time(T_{tuning}) for the speedup, we will discuss the search overhead of WACO. Figure 17 shows the search time - speedup plot of three auto-tuning frameworks, MKL inspector-executor, BestFormat, and WACO. We compared these frameworks against naive MKL without an inspector-executor. For the matrices with a speedup >1.0×, SpMV and SpMM must be repeatedly run for 919 and 101 times to amortize the WACO's tuning cost on average. As pointed out in section 5.4, a feature extractor must be more lightweight to reduce this amortization cost. When comparing BestFormat and MKL, BestFormat showed better performance on both search overhead(T_{tuning}) and speedup than MKL. However, when comparing WACO and BestFormat, there was a clear trade-off; WACO achieves a better speedup by paying for more search time than BestFormat.

Real-world scenarios. Real-world applications that utilize an auto-tuner should consider both the tuning cost and the format converting cost. To be specific, the end-to-end execution time ($T_{tuning} + T_{formatconvert} + T_{tunedkernel} * N_{runs}$) needs to be considered [49, 52]. The auto-tuner with a significant search overhead, such as WACO, is only advantageous over other prior auto-tuners in applications requiring repetitive runs. We list some real-world applications with tens of thousands of runs of sparse routines in Table 8. We set $T_{formatconvert} = 0$ for MKL as it only tunes the schedule while fixing the format. Although BestFormat has the fastest T_{tuning} (Figure 17), MKL is advantageous when N is small due to no format conversion. It is better to use other auto-tuners if an application does not require many repetitions, such as PageRank. WACO is beneficial in scenarios that require a lot of runs, such as mesh simulation or GNN.

6 Related Works

Auto-scheduling and cost model. Halide auto-scheduler [1, 35] uses a cost model with hand-crafted program features and searches for the best schedule through a beam search. AutoTVM [10] uses a cost model that embeds the low-level loop AST. While AutoTVM automates the search process, its search space must be manually defined by the user's template. Recently, Ansor [50] allowed the auto-scheduler to find this template automatically by rewriting rules. Tiramisu auto-scheduler uses LSTM to embed the low-level loop AST [4]. There have also been many cost models that tried to predict the behavior of accelerator or x86 basic blocks [21, 33, 37, 38]. All these schemes attempted to design a cost model to embed a traversing order of iteration space alone since they usually targeted a dense tensor program, while WACO's cost model considers the sparsity pattern, the format and the schedule all together.

Auto-tuning sparse tensor programs. Previous auto-tuner of sparse tensor programs can be divided into two categories: format selection studies and schedule optimization studies. There has been a format selection approach designed a classifier, which took a downsampled tensor and predicted which format would be optimal for the input [42, 48]. However, the features extracted over the downsampled tensor did not capture the pattern well and considered only a few output classes, for example, five formats, whereas WACO considers a large number of formats from the TACO's abstraction. Some other frameworks estimate the number of non-zeros in the dense block to choose the optimal block size in the BCSR [11, 46]. Mehrabi et al. utilized a predictive model to learn the optimal permutation of rows for better load balancing [32].

Regarding auto-tuning of the schedule, ESB [29] suggested choosing an optimal load-balancing scheme by running a kernel several times, each time with different load-balancing schemes. Venkat et al. proposed an inspector-executor method to transform a sparse loop and data with polyhedral optimizations [45]. Their three proposed transformations *make-dense*, *compact*, and *compact-and-pad* can actually demonstrate the same search space as TACO's transformation framework [41] provided there is a single sparse input among all input operands. However, they only suggested how to transform the sparse loop but not how to transform it automatically. Therefore, WACO can be used as an auto-tuner to automatically transform the code by replacing TACO with their framework.

7 Conclusion

This paper presented WACO, a technique co-optimizing the format and the schedule for a given sparsity pattern. In the sparse tensor programs, it is crucial to design the cost model to consider various sparsity patterns. To address this, we proposed a novel feature extractor that employs a sparse convolutional network. Its obtained features were universal across various formats and were useful for predicting the coupled behavior between the format and the schedule. Furthermore, a graph-based ANNS, a discretized version of the gradient-based search, efficiently and accurately finds the best format and schedule in the large search space of the co-optimization.

Acknowledgments

We thank anonymous reviewers for their valuable suggestions. We thank Teodoro Collin, Stephen Chou, and Willow Ahrens for reading early draft of this paper and providing feedback. This work was supported by the Application Driving Architectures (ADA) Research Center, a JUMP Center cosponsored by SRC and DARPA; the U.S.Department of Energy, Office of Science, Office of Advanced Scientific Computing Research under Award Numbers DE-SC0008923 and DE-SC0018121; and DARPA under Awards HR0011-18-3-0007 and HR0011-20-9-0017; and NSF Award CCF-2107244. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of the aforementioned funding agencies.

References

[1] Andrew Adams, Karima Ma, Luke Anderson, Riyadh Baghdadi, Tzu-Mao Li, Michaël Gharbi, Benoit Steiner, Steven Johnson, Kayvon Fatahalian, Frédo Durand, et al. 2019. Learning to optimize halide with tree search and random programs. *ACM Transactions on Graphics (TOG)* 38, 4 (2019), 1–12.

[2] Martin Alnæs, Jan Blechta, Johan Hake, August Johansson, Benjamin Kehlet, Anders Logg, Chris Richardson, Johannes Ring, Marie E Rognes, and Garth N Wells. 2015. The FEniCS Project Version 1.5. *Archive of Numerical Software* Vol 3 (2015).

[3] 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.

[4] Riyadh Baghdadi, Massinissa Merouani, Mohamed-Hicham Leghettas, Kamel Abdous, Taha Arbaoui, Karima Benatchba, et al. 2021. A Deep Learning Based Cost Model for Automatic Code Optimization. *Proceedings of Machine Learning and Systems* 3 (2021).

[5] Riyadh Baghdadi, Jessica Ray, Malek Ben Romdhane, Emanuele Del Sozzo, Abdurrahman Akkas, Yunming Zhang, Patricia Suriana, Shoaib Kamil, and Saman Amarasinghe. 2019. Tiramisu: A polyhedral compiler for expressing fast and portable code. In *2019 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*. IEEE, 193–205.

[6] James Bergstra, Daniel Yamins, and David Cox. 2013. Making a science of model search: Hyperparameter optimization in hundreds of dimensions for vision architectures. In *International conference on machine learning*. PMLR, 115–123.

[7] Filippo Maria Bianchi, Daniele Grattarola, and Cesare Alippi. 2020. Spectral clustering with graph neural networks for graph pooling. In *International Conference on Machine Learning*. PMLR, 874–883.

[8] Chris Burges, Tal Shaked, Erin Renshaw, Ari Lazier, Matt Deeds, Nicole Hamilton, and Greg Hullender. 2005. Learning to rank using gradient descent. In *Proceedings of the 22nd international conference on Machine learning*. 89–96.

[9] Tianqi Chen, Thierry Moreau, Ziheng Jiang, Lianmin Zheng, Eddie Yan, Meghan Cowan, Haichen Shen, 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 Conference on Operating Systems Design and Implementation* (Carlsbad, CA, USA) *(OSDI'18)*. USENIX Association, USA, 579–594.

[10] Tianqi Chen, Lianmin Zheng, Eddie Yan, Ziheng Jiang, Thierry Moreau, Luis Ceze, Carlos Guestrin, and Arvind Krishnamurthy. 2018. Learning to Optimize Tensor Programs. In *Proceedings of the 32nd International Conference on Neural Information Processing Systems* (Montréal, Canada) *(NIPS'18)*. Curran Associates Inc., Red Hook, NY, USA, 3393–3404.

[11] Jee W Choi, Amik Singh, and Richard W Vuduc. 2010. Model-driven autotuning of sparse matrix-vector multiply on GPUs. *ACM sigplan notices* 45, 5 (2010), 115–126.

[12] Stephen Chou, Fredrik Kjolstad, and Saman Amarasinghe. 2018. Format abstraction for sparse tensor algebra compilers. *Proceedings of the ACM on Programming Languages* 2, OOPSLA (2018), 1–30.

[13] Christopher Choy, JunYoung Gwak, and Silvio Savarese. 2019. 4d spatio-temporal convnets: Minkowski convolutional neural networks. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*. 3075–3084.

[14] 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.

[15] Matteo Frigo and Steven G Johnson. 1998. FFTW: An adaptive software architecture for the FFT. In *Proceedings of the 1998 IEEE International Conference on Acoustics, Speech and Signal Processing, ICASSP'98 (Cat. No. 98CH36181)*, Vol. 3. IEEE, 1381–1384.

[16] 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.

[17] Benjamin Graham and Laurens van der Maaten. 2017. Submanifold sparse convolutional networks. *arXiv preprint arXiv:1706.01307* (2017).

[18] Kartik Hegde, Po-An Tsai, Sitao Huang, Vikas Chandra, Angshuman Parashar, and Christopher W. Fletcher. 2021. Mind Mappings: Enabling Efficient Algorithm-Accelerator Mapping Space Search. In *Proceedings of the 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems* (Virtual, USA) *(ASPLOS 2021)*. 16 pages.

[19] 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*. 300–314.

[20] Guyue Huang, Guohao Dai, Yu Wang, and Huazhong Yang. 2020. Ge-spmm: General-purpose sparse matrix-matrix multiplication on gpus for graph neural networks. In *SC20: International Conference for High Performance Computing, Networking, Storage and Analysis*. IEEE, 1–12.

[21] Qijing Huang, Aravind Kalaiah, Minwoo Kang, James Demmel, Grace Dinh, John Wawrzynek, Thomas Norell, and Yakun Sophia Shao. 2021. CoSA: Scheduling by Constrained Optimization for Spatial Accelerators. In *2021 ACM/IEEE 48th Annual International Symposium on Computer Architecture (ISCA)*. 554–566.

[22] Zhihao Jia, Oded Padon, James Thomas, Todd Warszawski, Matei Zaharia, and Alex Aiken. 2019. TASO: optimizing deep learning computation with automatic generation of graph substitutions. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles*. 47–62.

[23] Jeremy Kepner, Peter Aaltonen, David Bader, Aydin Buluç, Franz Franchetti, John Gilbert, Dylan Hutchison, Manoj Kumar, Andrew Lumsdaine, Henning Meyerhenke, Scott McMillan, Carl Yang, John D. Owens, Marcin Zalewski, Timothy Mattson, and Jose Moreira. 2016. Mathematical foundations of the GraphBLAS. In *2016 IEEE High Performance Extreme Computing Conference (HPEC)*. 1–9.

[24] Diederik P. Kingma and Jimmy Ba. 2015. Adam: A Method for Stochastic Optimization. In *3rd International Conference on Learning Representations, ICLR 2015, San Diego, CA, USA, May 7-9, 2015, Conference Track Proceedings*.

[25] 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.

[26] Fredrik Kjolstad, Shoaib Kamil, Jonathan Ragan-Kelley, David IW Levin, Shinjiro Sueda, Desai Chen, Etienne Vouga, Danny M Kaufman, Gurtej Kanwar, Wojciech Matusik, et al. 2016. Simit: A language for physical simulation. *ACM Transactions on Graphics (TOG)* 35, 2 (2016), 1–21.

[27] Jiajia Li, Guangming Tan, Mingyu Chen, and Ninghui Sun. 2013. SMAT: an input adaptive auto-tuner for sparse matrix-vector multiplication. In *Proceedings of the 34th ACM SIGPLAN conference on Programming language design and implementation*. 117–126.

[28] Wen Li, Ying Zhang, Yifang Sun, Wei Wang, Mingjie Li, Wenjie Zhang, and Xuemin Lin. 2019. Approximate nearest neighbor search on high dimensional data—experiments, analyses, and improvement. *IEEE Transactions on Knowledge and Data Engineering* 32, 8 (2019), 1475–1488.

[29] Xing Liu, Mikhail Smelyanskiy, Edmond Chow, and Pradeep Dubey. 2013. Efficient sparse matrix-vector multiplication on x86-based many-core processors. In *Proceedings of the 27th international ACM conference on International conference on supercomputing*. 273–282.

[30] Jennifer A. Loe, Heidi K. Thornquist, and Erik G. Boman. 2019. Polynomial Preconditioned GMRES to Reduce Communication in Parallel Computing. https://doi.org/10.48550/ARXIV.1907.00072

[31] Yu A Malkov and Dmitry A Yashunin. 2018. Efficient and robust approximate nearest neighbor search using hierarchical navigable small world graphs. *IEEE transactions on pattern analysis and machine intelligence* 42, 4 (2018), 824–836.

[32] Atefeh Mehrabi, Donghyuk Lee, Niladrish Chatterjee, Daniel J. Sorin, Benjamin C. Lee, and Mike O'Connor. 2021. Learning Sparse Matrix Row Permutations for Efficient SpMM on GPU Architectures. In *IEEE International Symposium on Performance Analysis of Systems and Software, ISPASS 2021, Stony Brook, NY, USA, March 28-30, 2021*. IEEE, 48–58.

[33] Charith Mendis, Alex Renda, Dr.Saman Amarasinghe, and Michael Carbin. 2019. Ithemal: Accurate, Portable and Fast Basic Block Throughput Estimation using Deep Neural Networks. In *Proceedings of the 36th International Conference on Machine Learning (Proceedings of Machine Learning Research, Vol. 97)*. PMLR.

[34] Intel MKL. 2022. Inspector-executor Sparse BLAS Routines. (2022). https://www.intel.com/content/www/us/en/develop/documentation/onemkl-developer-reference-c/top/blas-and-sparse-blas-routines/inspector-executor-sparse-blas-routines.html

[35] Ravi Teja Mullapudi, Andrew Adams, Dillon Sharlet, Jonathan Ragan-Kelley, and Kayvon Fatahalian. 2016. Automatically scheduling halide image processing pipelines. *ACM Transactions on Graphics (TOG)* 35, 4 (2016), 1–11.

[36] Sameer A Nene and Shree K Nayar. 1997. A simple algorithm for nearest neighbor search in high dimensions. *IEEE Transactions on pattern analysis and machine*

intelligence 19, 9 (1997), 989–1003.

[37] Angshuman Parashar, Priyanka Raina, Yakun Sophia Shao, Yu-Hsin Chen, Victor A. Ying, Anurag Mukkara, Rangharajan Venkatesan, Brucek Khailany, Stephen W. Keckler, and Joel Emer. 2019. Timeloop: A Systematic Approach to DNN Accelerator Evaluation. In *2019 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*. 304–315.

[38] Mangpo Phothilimthana, Mike Burrows, and Samuel J. Kaufman. 2019. Learned TPU Cost Model for XLA Tensor Programs. In *Workshop on ML for Systems at NeurIPS*.

[39] Jonathan Ragan-Kelley, Connelly Barnes, Andrew Adams, Sylvain Paris, Frédo Durand, and Saman Amarasinghe. 2013. Halide: a language and compiler for optimizing parallelism, locality, and recomputation in image processing pipelines. *Acm Sigplan Notices* 48, 6 (2013), 519–530.

[40] Naser Sedaghati, Te Mu, Louis-Noel Pouchet, Srinivasan Parthasarathy, and P Sadayappan. 2015. Automatic selection of sparse matrix representation on GPUs. In *Proceedings of the 29th ACM on International Conference on Supercomputing*. 99–108.

[41] Ryan Senanayake, Changwan Hong, Ziheng Wang, Amalee Wilson, Stephen Chou, Shoaib Kamil, Saman Amarasinghe, and Fredrik Kjolstad. 2020. A sparse iteration space transformation framework for sparse tensor algebra. *Proceedings of the ACM on Programming Languages* 4, OOPSLA (2020), 1–30.

[42] 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*. IEEE, 1–14.

[43] Vivienne Sze, Yu-Hsin Chen, Tien-Ju Yang, and Joel S Emer. 2020. Efficient processing of deep neural networks. *Synthesis Lectures on Computer Architecture* 15, 2 (2020), 1–341.

[44] Shulong Tan, Zhixin Zhou, Zhaozhuo Xu, and Ping Li. 2020. Fast item ranking under neural network based measures. In *Proceedings of the 13th International Conference on Web Search and Data Mining*. 591–599.

[45] Anand Venkat, Mary Hall, and Michelle Strout. 2015. Loop and Data Transformations for Sparse Matrix Code. In *Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation* (Portland, OR, USA) *(PLDI '15)*. Association for Computing Machinery, New York, NY, USA, 521–532.

[46] Richard Vuduc, James W Demmel, and Katherine A Yelick. 2005. OSKI: A library of automatically tuned sparse matrix kernels. In *Journal of Physics: Conference Series*, Vol. 16. IOP Publishing, 071.

[47] R Clinton Whaley and Jack J Dongarra. 1998. Automatically tuned linear algebra software. In *SC'98: Proceedings of the 1998 ACM/IEEE conference on Supercomputing*. IEEE, 38–38.

[48] Yue Zhao, Jiajia Li, Chunhua Liao, and Xipeng Shen. 2018. Bridging the gap between deep learning and sparse matrix format selection. In *Proceedings of the 23rd ACM SIGPLAN symposium on principles and practice of parallel programming*. 94–108.

[49] Yue Zhao, Weijie Zhou, Xipeng Shen, and Graham Yiu. 2018. Overhead-conscious format selection for SpMV-based applications. In *2018 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*. IEEE, 950–959.

[50] Lianmin Zheng, Chengfan Jia, Minmin Sun, Zhao Wu, Cody Hao Yu, Ameer Haj-Ali, Yida Wang, Jun Yang, Danyang Zhuo, Koushik Sen, et al. 2020. Ansor: Generating high-performance tensor programs for deep learning. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*. 863–879.

[51] Size Zheng, Yun Liang, Shuo Wang, Renze Chen, and Kaiwen Sheng. 2020. *FlexTensor: An Automatic Schedule Exploration and Optimization Framework for Tensor Computation on Heterogeneous System.* Association for Computing Machinery, New York, NY, USA, 859–873.

[52] Weijie Zhou, Yue Zhao, Xipeng Shen, and Wang Chen. 2019. Enabling runtime SpMV format selection through an overhead conscious method. *IEEE Transactions on Parallel and Distributed Systems* 31, 1 (2019), 80–93.

Received 2022-07-07; accepted 2022-09-22