STile: Searching Hybrid Sparse Formats for Sparse Deep Learning Operators Automatically

ABSTRACT

Sparse operators, i.e., operators that take sparse tensors as input, are of great importance in deep learning models. Due to the diverse sparsity patterns in different sparse tensors, it is challenging to optimize sparse operators by seeking an optimal sparse format, i.e., leading to the lowest operator latency. Existing works propose to decompose a sparse tensor into several parts and search for a hybrid of sparse formats to handle diverse sparse patterns. However, they often make a trade-off between search space and search time: their search spaces are limited in some cases, resulting in limited operator running efficiency they can achieve. In this paper, we try to extend the search space in its breadth (by doing flexible sparse tensor transformations) and depth (by enabling multi-level decomposition). We formally define the multi-level sparse format decomposition problem, which is NP-hard, and we propose a framework STile for it. To search efficiently, a greedy algorithm is used, which is guided by a cost model about the latency of computing a sub-task of the original operator after decomposing the sparse tensor. Experiments of two common kinds of sparse operators, SpMM and SDDMM, are conducted on various sparsity patterns, and we achieve 2.1 − 18.0× speedup against cuSPARSE on SpMMs and 1.5 − 6.9× speedup against DGL on SDDMM. The search time is less than one hour for any tested sparse operator, which can be amortized.

CCS CONCEPTS

• **Mathematics of computing** → **Combinatorial optimization**; • **Software and its engineering** → *Source code generation.*

KEYWORDS

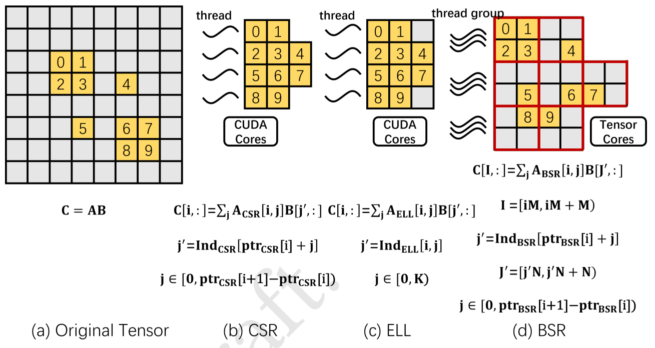
sparse operation, sparse format, hybrid format

ACM Reference Format:

. 2018. STile: Searching Hybrid Sparse Formats for Sparse Deep Learning Operators Automatically. In *Proceedings of Make sure to enter the correct conference title from your rights confirmation emai (Conference acronym 'XX).* ACM, New York, NY, USA, 16 pages. https://doi.org/XXXXXXX.XXXXXXX

1 INTRODUCTION

Sparse operators, i.e., operators that take sparse tensors as input, are a kind of important operators in deep learning to be optimized. For example, the Sparse Matrix Multiplication (SpMM) operator computes the multiplication result of a sparse matrix and a dense



thread
thread
thread group
CUDA Cores
CUDA Cores
Tensor Cores
$\mathbf{C = AB}$
$\mathbf{C[i,:]=\sum_j A_{CSR}[i,j]B[j',:]}$
$\mathbf{j'=Ind_{CSR}[ptr_{CSR}[i]+j]}$
$\mathbf{j \in [0, ptr_{CSR}[i+1]-ptr_{CSR}[i])}$
$\mathbf{C[i,:]=\sum_j A_{ELL}[i,j]B[j',:]}$
$\mathbf{j'=Ind_{ELL}[i,j]}$
$\mathbf{j \in [0, K)}$
$\mathbf{C[I,:]=\sum_j A_{BSR}[i,j]B[J',:]}$
$\mathbf{I=[iM, iM+M)}$
$\mathbf{j'=Ind_{BSR}[ptr_{BSR}[i]+j]}$
$\mathbf{J'=[j'N, j'N+N)}$
$\mathbf{j \in [0, ptr_{BSR}[i+1]-ptr_{BSR}[i])}$
(a) Original Tensor
(b) CSR
(c) ELL
(d) BSR

Figure 1: Common sparse formats and corresponding SpMM implementations. The non-zeros are colored. For CSR, $Ind_{CSR}[ptr_{CSR}[i] : ptr_{CSR}[i+1]]$ stores the column indices of the non-zeros in row i. For ELL, $Ind_{ELL}[iK : iK+K]$ stores the column indices of the non-zeros and the padded zeros in row i, where K is the maximum number of non-zeros per row. For BSR, $Ind_{BSR}[ptr_{BSR}[i] : ptr_{BSR}[i+1]]$ stores the column indices of the non-zero blocks in block row i; (M, N) is the block size, i.e., $(2, 2)$ in figure (d).

matrix. Compared with dense matrix multiplication, SpMM can skip the multiply-add operations related to the zeros of the sparse matrix. Another example is Sampled Dense-Dense Matrix Multiplication (SDDMM), which computes dense matrix multiplication but only samples a part of the result as output. The sparsity has multiple sources. For example, the sparsity of tensors may come from the datasets themselves, like the sparse adjacent matrices [15, 20, 33] used in graph neural networks. Besides, model pruning [23, 32] also generates sparse tensors. These factors give rise to a variety of sparsity patterns in the tensors used in deep learning.

A number of sparse formats have been developed to cope with the diverse sparse patterns in the increasing sparse operators. These formats have advantages and disadvantages when dealing with different sparsity patterns. For example, the CSR format compactly stores the non-zeros of a sparse matrix by rows, as Figure 1(b) shows. In SpMM, it is common to make each GPU thread compute one row of the output ($C[i,:]$ in Figure 1(b)), but the irregular row lengths after compression will cause workload imbalance among threads which damages the operator efficiency. The ELL format is different from CSR in that it pads zeros to each row of the ragged compressed matrix and therefore returns a regular matrix (row length is K in Figure 1(c)). It removes the ptr array in CSR to simplify the process of getting valid column indices, eliminates control flow divergence in threads, and enables memory coalescing if with column-major storage, allowing better operator efficiency. The BSR format stores non-zero blocks (i.e., the blocks with at least one nonzero), e.g., Figure 1(d) stores non-zero 2×2 blocks. Compared with CSR, BSR stores the column indices for blocks instead of those

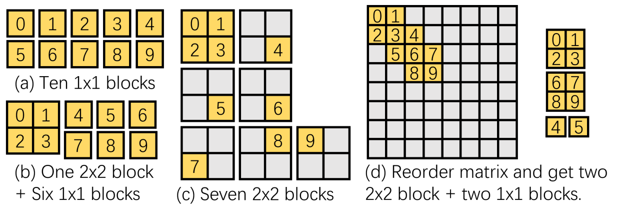

(a) Ten 1x1 blocks
(b) One 2x2 block + Six 1x1 blocks
(c) Seven 2x2 blocks
(d) Reorder matrix and get two 2x2 block + two 1x1 blocks.

Figure 2: Examples of block collections to cover non-zeros.

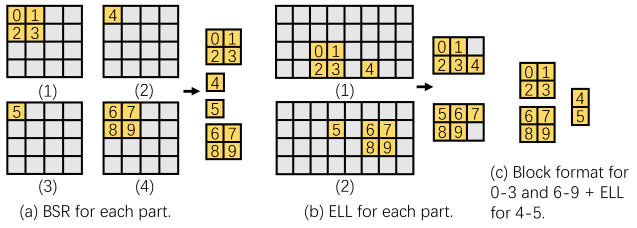


Figure 3: (a, b): Examples of partition-based decomposition; (c): Multi-level decomposition.

for non-zeros (Ind_{BSR}, ptr_{BSR} in Figure 1(d)) and hence reduces the memory usage for sparsity structure information. It also provides opportunities for vectorization or hardware intrinsics [10]. For instance, in SpMM, it is common to use a group of threads (a GPU thread block) to compute a set of output rows like $C[I, :]$ in Figure 1(d), which can run on the powerful GPU Tensor Cores for higher efficiency. However, the extra computation workload brought by padding may be a severe problem for both ELL and BSR. Therefore, ELL is more suitable for matrices where the number of nonzeros per row does not vary a lot, and BSR is more suitable for matrices with dense sub-matrices. It is well-recognized that no single format can suit all sparse tensors. Moreover, if a sparse tensor has different sparsity patterns in different regions, we may need to combine different formats together to achieve optimal performance.

To deal with the diverse sparsity patterns of different sparse tensors and of different parts of a sparse tensor, people try to use hybrid formats, i.e., taking the sparse operator into consideration, they decompose a sparse tensor into several sparse sub-tensors and select the best sparse format for each of them [12, 19, 30, 31, 40, 42], then they schedule the computation based on the hybrid format.

There are two major types of decomposition methods: the set-cover-based decomposition [42], where the decomposed sparse sub-tensors may have intersections, and the partition-based decomposition [12, 19, 30, 31, 40], where there are no intersections between the decomposed sparse sub-tensors. However, the existing works often make a trade-off between search space and search time. As a result, their search spaces are limited in some cases, limiting the optimized operator efficiency they can achieve.

Set-cover-based decomposition. Given a sparse tensor, a sub-tensor, also called a block, covers a set of non-zeros of the original tensor. The set-cover-based decomposition is to find the best collection of blocks such that all the non-zeros of the original sparse tensor are covered to ensure the accuracy of the computation result of the sparse operator. The existing work in this category is [42]. It only considers using the block format to store the decomposed blocks, i.e., if not full of zeros, each block will be stored entirely in the memory, including both the zeros and the non-zeros in it. We use the sparse tensor in Figure 1(a) to explain this kind of decomposition below. In practice, the complete computation of a sparse operator will be divided into several sub-tasks and assigned to multiple parallel computation units. For simplicity, in this example and other examples in Section 1, we assume (1) each sub-task runs the computation related to one block, and hence we directly refer to the sub-task cost (i.e., the running time) as the corresponding block cost; (2) sub-tasks run on one computation unit so the total cost of the selected blocks is the summation of their costs. As the candidate blocks are generated by partitioning the sparse tensor evenly in [42], we assume the block shapes we consider are 1×1, 2×2, and 4×4 and the associated costs are 2, 4, and 15, respectively. Figure 2 shows some possible block collections to cover the non-zeros. If we only select 1×1 blocks to cover the non-zeros (Figure 2(a)), the total cost is 20 (10×2). If we only select 2×2 blocks, 7 blocks will be selected in the end (Figure 2(c)), the total cost is 28 (7×4). The example of Figure 2(b) considers hybrid block shapes: it selects one 2×2 block for the non-zeros 0-3 and six 1×1 blocks for the remaining non-zeros, and the total cost is 16 ($4 + 6 \times 2$). However, as [42] does not consider other possible sparse formats for the local sparse patterns other than the block format and does not perform the possible transformations of the original sparse tensor, like matrix row/column reordering, its search space of the decomposition solutions is limited. For example, as shown by Figure 2(d), we can first transform the sparse matrix by reordering its rows from rows 0, 1, 2, 3, 4, 5, 6, 7 to rows 2, 3, 5, 6, 0, 1, 4, 7, and reordering its columns from columns 0, 1, 2, 3, 4, 5, 6, 7 to columns 2, 3, 5, 6, 0, 1, 4, 7. Then we use two 2×2 blocks to cover non-zeros 0-3 and 6-7, respectively, and another two 1×1 blocks for non-zeros 4, 5. The cost is 12 ($2 \times 4 + 2 \times 2$).

Partition-based decomposition. Given a sparse matrix, the partition-based decomposition method will partition the matrix into several parts such that we can find a good sparse format for each part that is suitable for the corresponding local sparse pattern. To make the local sparse patterns more obvious, the existing works [12, 19, 40] will transform the matrix, like reordering the rows by the number of non-zeros. Taking the sparse matrix of Figure 1(a) as an example, Figure 3(a, b) show some partition-based decomposition results. Figure 3(a) divides the original tensor into four parts and reorders the rows and columns of each part. Each part is then stored in the BSR format. The total cost is 12 ($2 \times 4 + 2 \times 2$). Figure 3(b) partitions the original tensor into two parts and stores each part in the ELL format. Similar to the block cost, we assume each of the ELL-format submatrices is of cost 6, then the total cost is 12 (6×2). However, the search space of the partition-based decomposition method is also limited because there may be some non-zeros in a submatrix that do not fit the selected format, and it will be better for us to use another format to store such non-zeros of each submatrix. For example, if we select two ELL formats as in Figure 3(b), we need to pad two zeros, which wastes the computation. By contrast, as shown by Figure 3(c), we can select the block format for "0-3" in the first matrix part ("4" is left) and the block format for "6-9" in the second part ("5" is left). Then we use ELL for the remaining "4" and "5". Assume the cost of the ELL format for 4 and 5 is 3. The total cost is 11 ($2 \times 4 + 3$). Our experiments show that considering such multiple levels (two levels in this example) of format selection can lead to 3.6× operator efficiency improvement in some cases.

From the above examples, we observe that to find a good decomposition of a given sparse tensor, it is necessary to make the search space large enough. Specifically, we should not only extend the breadth of the search space by considering the matrix transformations and various sparse formats (as shown by Figure 2(d) and Figure 3(c)) but also extend the depth of the search space by considering the multi-level decomposition (i.e., select formats for a part of the non-zeros first, then reorganize the remaining non-zeros and repeat this process, as shown by Figure 3(c), where the two block formats are selected for the first level, and the ELL format is selected for the second level).

Therefore, we face two major challenges in finding a good sparse format decomposition: (1) how to construct a large search space with its breadth and depth extended; (2) how to search efficiently in the large search space to find a good hybrid sparse format.

To solve the first challenge, when we extend the breadth of the search space, we introduce the concept of a sparse tile, which is a sub-tensor with a specified sparse format. We have summarized a set of basic sparse formats for the sparse tiles and a set of transformations on the sparse tensor such that our search space of sparse tiles can cover common sparse formats. To extend the depth of the search space, we consider multi-level sparse tiles. Specifically, the first-level sparse tiles are generated from the original sparse tensor or its transformed versions (e.g., with rows reordered). After selecting one or more sparse tiles, we can get a new tensor with the remaining non-zeros and its transformed versions, from which the second-level sparse tiles can be extracted. The third-level sparse tiles and so on can be extracted in the same way.

To solve the second challenge, we propose a multi-level sparse format decomposition problem (shorted for MSFD), which is proved to be NP-hard, and we propose a framework STile to solve it, where a greedy algorithm is used to search a good hybrid format based on our cost model. The greedy algorithm provides an approximation guarantee when we are limited to the first-level sparse tiles. To reduce the search time, we also use local search.

We make the following contributions in this work:

(1) We extend the search space of possible sparse tiles in both its breadth and depth.
(2) We define the MSFD problem and prove its hardness.
(3) We design a framework STile with a greedy search algorithm and analyze the approximation ratio in a special case.
(4) We conduct experiments to compare our method with state-of-the-art: STile achieves $2.1 - 18.0\times$ speedup against cuSPARSE on SpMM and $1.5 - 6.9\times$ speedup against DGL on SDDMM. The search time is less than one hour for any tested sparse operator, which can be amortized.

2 DEFINITIONS AND PROBLEM

As introduced in Section 1, a sparse tensor is a tensor where only a part of the elements are non-zeros, and a sparse format is how we store the non-zeros. The first concept in this paper is sparse tile:

Table 1: Notation table.

Symbol	Meaning
E	a set of non-zeros
α	a sparse format
$t = (E, \alpha)$	a sparse tile
T, A, B, C	a (sparse) tensor
O	a (sparse) operator
$S = \{t_1, ..., t_m\}$	a valid tile selection
$I, J, K, ...$	the operator iteration space dimensions
U	the set of cannot-skip iteration points of O
u	a subset of U
x	a sub-task of to complete a sparse operator
$X = \{x_1, ..., x_m\}$	a task partition of a sparse operator
p	a task partition way
P	a set of task partition ways
W	a compute worker (e.g., an SM for GPUs)
$cost(X)$	the cost of a task partition for an operator

Definition 1 (Sparse Tile). *A sparse tile t is a sub-tensor of a sparse tensor with a specified sparse format, denoted by (E, α), where E is the set of non-zeros covered by t and α is the sparse format to store E.*

For example, the first sparse tile in Figure 3(a) can be denoted by $(E = \{(2, 2), (2, 3), (3, 2), (3, 3)\}, \alpha = \text{block format})$, where E stores the coordinates of the four non-zeros in the original sparse matrix. Given a sparse tensor, denoted by T, we can generate multiple candidate sparse tiles by transforming T (the transformation details will be introduced later), which are first-level sparse tiles. After we select one or more sparse tiles, we can get a new sparse tensor containing the remaining non-zeros of T, and we can transform the new tensor to get new sparse tiles, i.e., the second-level sparse tiles. By repeating this process, our search space includes multi-level sparse tiles. Taking the sparse matrix in Figure 1(a) and the sparse tiles in Figure 3(c) as an example. The sparse tiles storing $0 - 3$ and $6 - 9$ respectively are first-level sparse tiles because they can be obtained by transforming the original sparse matrix in Figure 1(a); while the sparse tile with ELL format storing $4, 5$ is a second-level sparse tile because it can only be generated after selecting the previous two sparse tiles and reorganizing the remaining non-zeros.

We need to select a collection of sparse tiles to determine the hybrid sparse format (a sparse format decomposition) for T, which is defined below.

Definition 2 (Valid Tile Selection). *A valid tile selection for a sparse tensor T, denoted by S, is a set of sparse tiles such that all non-zeros of T are covered by at least one tile in S.*

Given an operator and the tile selection(s) for its sparse input tensor(s), we can partition the computation task accordingly into a set of sub-tasks and generate efficient code for the sub-tasks to get the final program. Specifically, the computation task of a sparse operator O can be represented by an iteration space whose dimensions correspond to the loops in the computation expression of O, and some points in the iteration space will be skipped because of the sparsity. A sub-task can be defined as follows:

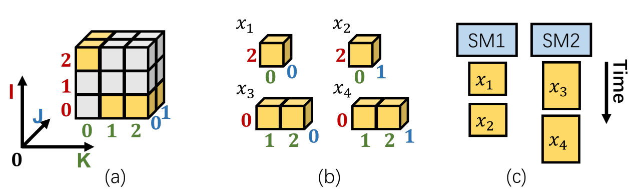


Figure 4: (a) The iteration space of an SPMM operator O is three-dimensional (I, J, K), where the gray points can be skipped while the yellow points cannot; U includes the yellow points. (b) By selecting two sparse tiles $t_1 = (\{(2,0)\}, \text{ELL})$, $t_2 = (\{(0,1),(0,2)\}, \text{ELL})$ for the sparse input tensor and dividing loop J into $J_1 = \{0\}, J_2 = \{1\}$, we get four sub-tasks of O, each covers a subset of U, e.g., x_1 has $u_1 = \{(2,0,0)\}$. (c) The cost of running the four sub-tasks is the maximum cost of the SMs, i.e., the cost of SM2.

Definition 3 (Sub-task). *Given a sparse operator, O, and the set of points that O cannot skip in the iteration space, denoted by U, a sub-task x iterates in a sub-space of the iteration space of O and covers a subset of U, denoted by u.*

Definition 4 (Task Partition). *A task partition X for a sparse operator O is a set of the sub-tasks s.t. $\bigcup_{x_i \in X} u_i = U$.*

Take an SpMM operator O as an example, which performs the multiplication between a sparse matrix $A_{I\times K}$ and a dense matrix $B_{K\times J}$, i.e., it computes $C[i,j] = \sum_{k\in K} A[i,k] \times B[k,j], i \in I, j \in J$. The expression includes three loops I, J, K where I, J are spatial loops and K is the reduction loop, so the iteration space of O is 3-dimensional, but the iteration over I, K will skip some points because of the sparsity of A, while the iteration over J cannot skip any points, i.e., J is a dense loop. Figure 4(a) illustrates this iteration space. Suppose the tile selection for A is $S = \{t_1, t_2\}$, where $t_1 = (\{(2,0)\}, \text{ELL})$, $t_2 = (\{(0,1),(0,2)\}, \text{ELL})$, the computation task may be partitioned by dividing the iteration over J into two parts $J_1 = \{0\}, J_2 = \{1\}$ and making each sub-task iterate over the points determined by t_i and J_j, where $t_i \in S, J_j \in \{J_1, J_2\}$. Therefore, we get four sub-tasks: x_1 with $u_1 = \{(2,0,0)\}$, x_2 with $u_2 = \{(2,1,0)\}$, x_3 with $u_3 = \{(0,0,1),(0,0,2)\}$, x_4 with $u_4 = \{(0,1,1),(0,1,2)\}$, as shown in Figure 4(b).

Definition 5 (Task Partition Way). *A task partition way for a sparse operator O, denoted by $p = (i_I, i_J, i_K, \ldots)$, is the way to partition each dense loop of O, where i_I (or $i_J, i_K, \ldots$) is the size of each part of the dense loop I (or $J, K, \ldots$) after partition.*

In the above example, the task partition way is $p = (i_J), i_J = 1$. For a sparse operator, given a tile selection S and a task partition way p, the task partition X is determined.

The cost of a sub-task is its processing time on hardware; the cost of a task partition is the total running time to finish all the sub-tasks on hardware. Specifically, on GPU, which we target in this paper, the sub-tasks will be assigned to multiple streaming multiprocessors (SMs) which work in parallel, and therefore the total running time of all the sub-tasks is the maximum running time of the SMs. Figure 4(c) shows an example of running multiple sub-tasks on SMs, in which the total cost is the cost of SM2. As we do not know how GPU will schedule the sub-tasks, we assume it will use the best schedule which minimizes the maximum running time of the SMs. The cost of a task partition is defined below.

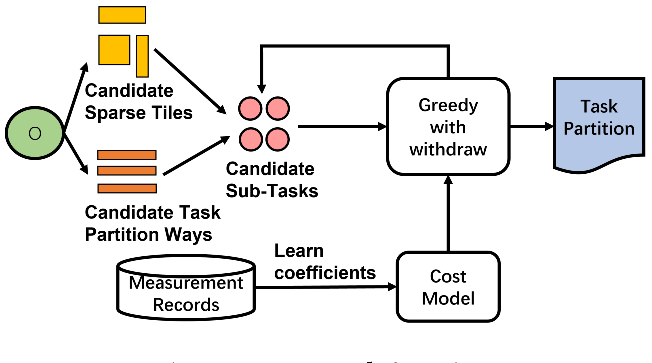


Figure 5: Framework Overview.

Definition 6 (Cost of Task Partition). *Given a task partition X with sub-tasks $x_1, \ldots, x_m$ and a GPU with SMs $W_1, \ldots, W_n$, when the assignment of the sub-tasks to the SMs is $f = \{X_1, \ldots, X_n\}$ (X_i is the set of the sub-tasks assigned to W_i), the cost of X under f is $cost_f(X) = \max_i \sum_{x\in X_i} cost(x)$. Therefore, the cost of X is $cost(X) = \min_f cost_f(X)$.*

We now define the multi-level sparse tiles to extend the depth of the search space.

Definition 7 (Multi-level sparse tile). *Given a sparse operator whose sparse input is denoted by T_0, we can get the set of possible first-level sparse tiles, denoted by ST_1, by transforming T_0; after selecting a subset S_1 of ST_1, we get a new sparse tensor T_1 with the remaining non-zeros of T_0 and a set of possible second-level sparse tiles, denoted by ST_2, by transforming T_1. We repeat this process: in the i-th round, after selecting a subset S_i of ST_i, we get a sparse tensor T_i with the remaining non-zeros of T_{i-1} and a set of possible $(i+1)$-th-level sparse tiles ST_{i+1}, by transforming T_i. The process stops in round N when $S = \bigcup_1^N S_i$ is a valid tile selection for T_0.*

Now, we formally define the multi-level sparse format decomposition problem (shorted for MSFD) below.

Definition 8 (MSFD problem). *Given a sparse operator whose sparse input is denoted by T_0 and a set of task partition ways P, The MSFD problem is to get a valid tile selection S by selecting sparse tiles level by level in the way defined in Definition 7 and choose a task partition way $p \in P$ s.t. the cost of the corresponding task partition X, i.e., $cost(X)$, is minimized.*

Theorem 2.1. *The MSFD problem is NP-hard.*

Proof sketch. Consider a simplified version of the problem: we only consider the first-level sparse tiles and the GPU only has one SM. In this setting, we get a fixed set of possible sub-tasks. Finding a task partition with the minimum cost is equivalent to solving the weighted set cover problem, which is NP-hard. Therefore, the multi-level sparse format decomposition problem is NP-hard. □

Table 2: Transformation Primitives

No.	Primitive	Parameter	Description
1	Reorder rows/columns	-	Reorder the rows/columns of a matrix by #non-zeros per row/per column
2	Divide rows/columns	size i	Divide the matrix into several parts along rows/columns s.t. each part has i rows/columns
3	Squeeze	-	Only store non-zeros in memory
4	Split rows	size j	Split each row into several parts s.t. each parts has j non-zeros after squeeze
5	Zero padding	-	Pad zeros along rows/columns
6	flatten	-	Store elements into 1D array

Table 3: FLOPs, Bytes, and Bytes$_{atomic}$ of a sub-task (#bytes: the unit byte number of the data type (e.g., 2 for float16)). $Ind[]$ **maps an element in the ELL format to its original column in the sparse matrix.** $row[], col[]$ **map an element in the 1D format to its original row and column in the sparse matrix, respectively.**

Op	Format	Expression	FLOPs	Bytes	Bytes$_{atomic}$
SpMM	ELL	$C[i,j] = A[i,k] \times B[Ind[i,k], j]$	$2IJK$	$(IK + \lvert set(Ind[I,K])\rvert J + IJ)\times$#bytes	Bytes+$IJ\times$#bytes
SDDMM	1D	$C[i] = A[row[i],k] \times B[col[i],k]$	$2IK$	$(K(\lvert set(row[I])\rvert + \lvert set(col[I])\rvert) + I) \times \#bytes$	-

3 STILE

We propose STile to solve the MSFD problem. Figure 5 shows the overview of STile. Specifically, given a sparse operator O and its input tensors, we design the search spaces of possible sparse tiles and possible task partition ways (Section 3.1), which determine the search space of sub-tasks. We use a cost model to estimate the cost of a sub-task (Section 3.2), and based on it, we use a greedy-with-withdraw algorithm to get a task partition for O (Section 3.3).

3.1 Search Space

As we have mentioned, the search space of sub-tasks is determined by two factors: multi-level sparse tiles and task partition ways. Table 2 shows the primitives we use to generate sparse tiles, which are summarized from existing works. For example, the ELL format (Figure 1(c)) can be obtained by primitive 3 $\longrightarrow$ primitive 5 (for each row). The BSR format (Figure 1(d)) can be obtained by primitive 2 (along rows, $i = 2$) $\longrightarrow$ primitive 2 (along columns, $i = 2$). The condense format in [36] can be obtained by primitive 2 (along rows, $i = 16$) $\longrightarrow$ primitive 1 (along columns) $\longrightarrow$ primitive 2 (along columns, $i = 16$). The hyb format in [40] can be obtained by primitive 2 (along columns, i = any value) $\longrightarrow$ primitive 1 (along rows) $\longrightarrow$ primitive 3 $\longrightarrow$ primitive 4 ($j = 2^k$) $\longrightarrow$ primitive 5 (for each row) $\longrightarrow$ primitive 2 (along rows, $i = 2, 4, 8, \ldots$). The 1D format used for 1-D tiling in [13, 41] (i.e., storing the non-zeros consecutively) can be obtained by primitive 3 $\longrightarrow$ primitive 6 $\longrightarrow$ primitive 4 ($j = 2^k$) $\longrightarrow$ primitive 2 (along rows, $i = 1$).

In this work, we consider three basic formats for sparse tiles: the block format, the ELL format, the 1D format. If a sparse tile is obtained without applying the "Squeeze" primitive (i.e., the zeros are kept), we use the block format for it; if a sparse tile is obtained by using the "flatten" primitive, the "squeeze" primitive must also be applied to it, and this means the non-zeros are consecutively stored, so we use the 1D format for it; otherwise, we always apply the "Zero padding" primitive after "Squeeze" and use the ELL format for it. We run the computation related to block-format sparse tiles on GPU Tensor Cores, so the sub-tensor covered by such a sparse tile is required to be of certain shapes, e.g., (16, 16), (8, 16), which are consistent with the hardware intrinsics like wmma. The computation related to sparse tiles of other formats is run on GPU CUDA Cores. For ELL-format sparse tiles, there is a constraint about the total number of the elements (both non-zeros and zeros) kept by a sparse tile like [40], and we only allow sparse tiles satisfying this constraint in the search space.

When doing task partitioning, we split each dense loop (i.e., over which the iteration skips no point) in the computation of a sparse operator evenly into multiple parts, whose sizes are controlled by a parameter, and this determines the task partition way.

The partitioned dense loops and the candidate sparse tiles together determine the candidate sub-tasks. For example, suppose the dense loop of an SpMM operator O, which is loop J, is of length 64 and is partitioned into two parts, each of size 32: $J_1 = \{1, \ldots, 32\}, J_2 = \{33, \ldots, 64\}$. Let $\{t_1, \ldots, t_m\}$ be the candidate sparse tiles, where $t_i = (E_i, \alpha_i)$. Then the search space of sub-tasks is $Y = \{x_1, \ldots x_n\}$, where each $x_i \in Y$ iterates over a subset of the valid points in the iteration space U of the SpMM O, denoted by $u_i \subseteq U$, and u_i is from $\{E_1, \ldots, E_m\} \times \{J_1, J_2\}$.

3.2 Cost Model

We estimate the costs of block-format sub-tasks (sub-tasks based on block-format sparse tiles), the ELL-format sub-tasks (sub-tasks based on ELL-format sparse tiles), and the 1D-format sub-tasks (sub-tasks based on 1D-format sparse tiles) in different ways. For block-format sub-tasks, if the iteration space shapes of two sub-tasks x_1, x_2 are the same, their costs are the same, as they do not skip any point in their iteration space when doing computation (even if they can). Therefore, we list all the iteration space shapes of the block-format sub-tasks our search space contains and measure their costs in advance to build a cost table.

For ELL-format sub-tasks and 1D-format sub-tasks, the cost of a sub-task x is estimated by

$$cost(x) = g(x) * Atomic(x).$$

$g(x)$ is the latency to finish x without using the atomic operation (e.g., the "atomicAdd()" function in CUDA) when storing the computation result back to the GPU global memory. The atomic operation is necessary when different GPU threads compute the value of the same address together. For example, given two matrices A, B where A is sparse, suppose A is partitioned into 1×2 blocks, $A = [A_{1,1}, A_{1,2}]$, and B is partition into 2×1 blocks, $B = [B_{1,1}; B_{2,1}]$, then $C = A \times B = A_{1,1} \times B_{1,1} + A_{1,2} \times B_{2,1}$. If we store $A_{1,1}$ in the block format and $A_{1,2}$ in the ELL format, then there must be two sets of GPU threads to compute $A_{1,1} \times B_{1,1}$ and $A_{1,2} \times B_{2,1}$, respectively, and they all write to C, so the write operations need to be atomic. However, it will cause extra overhead because of additional memory operations and the serialization of the conflicting operations. Therefore, we use $Atomic(x)$ to estimate the influence of atomic operations if any.

Specifically, $g(x) = aR(x) + b$ is a linear function, where a, b are two constants, and $R(x)$ is the roofline model result [38] when not using atomic operations. $R(x)$ is computed by $FLOPs/\min(AI, PEAK)$: FLOPs is the total floating-point operations performed by x; AI is the arithmetic intensity of x, which is the ratio of FLOPs to the total data movement (Bytes) required to support those FLOPs; PEAK is the ratio of the GPU peak performance (FLOP/s) to the GPU peak bandwidth (Byte/s). Table 3 lists how we compute FLOPs and Bytes for the ELL-format sub-tasks and 1D-format sub-tasks we currently support. The values of a, b are obtained by fitting the linear function to the measurement results of different sub-tasks we collect in advance (line 12 of Algorithm 1).

When x does not need atomic operations, $Atomic(x) = 1$. For 1D-format sub-tasks, $Atomic(x)$ is usually 1 as they are often used for SDDMM (Sampled Dense-Dense Matrix Multiplication), whose output is sparse and the non-zero output points are often computed independently and in parallel. If there are atomic operations, $Atomic(x) = cR_{atomic}(x)/R(x) + d$, where $R_{atomic}(x)$ is the roofline model result considering atomic operations, which is $\frac{FLOPs}{\min(FLOPs/Bytes_{atomic}, PEAK)}$ and Bytes$_{atomic}$ is listed in Table 3; c, d are also constant parameters obtained by fitting the function to the pre-collected measurement results (line 13 of Algorithm 1).

Algorithm 1: Get Cost Model Coefficients

```
Data: None
Result: cost model coefficients $a, b, c, d$
1  Sample two sets of sub-tasks $\{x_1, ..., x_m\}, \{x'_1, ..., x'_m\}$, where
     $x_i, x'_i$ are the same except that $x_i$ does not do atomic
     operations while $x'_i$ does;
2  $PEAK \leftarrow \frac{GPU\ peak\ FLOPS}{GPU\ peak\ Byte/s}$;
3  $C, C_{atomic} \leftarrow$ new List, new List;
4  $\mathcal{R}, \mathcal{R}_{atomic} \leftarrow$ new List, new List;
5  for $i = 1, ..., m$ do
6      $c_1 \leftarrow$ latency of $x_i$ we collect; // details in Section 4
7      $c_2 \leftarrow$ latency of $x'_i$ we collect; // details in Section 4
8      $C$.append($c_1$), $C_{atomic}$.append($c_2$);
9      $\mathcal{R}$.append($\frac{FLOPs(x_i)}{\min(FLOPs(x_i)/Bytes(x_i), PEAK)}$);
10     $\mathcal{R}_{atomic}$.append($\frac{FLOPs(x'_i)}{\min(FLOPs(x'_i)/Bytes_{atomic}(x'_i), PEAK)}$);
11 end
12 $a, b \leftarrow$ LeastSquaresPolynomialFit($\mathcal{R}, C$);
13 $c, d \leftarrow$ LeastSquaresPolynomialFit($\mathcal{R}_{atomic}/\mathcal{R}, C_{atomic}/C$);
```

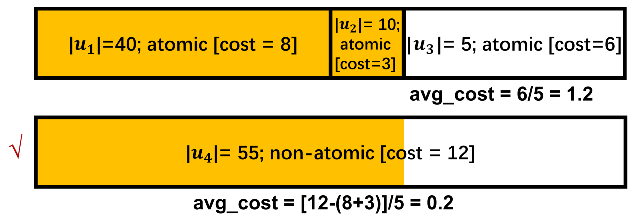
3.3 Greedy with Withdraw

We use a greedy algorithm to solve the MSFD problem. The main idea is as follows. In the beginning, we are given a sparse operator O with its input tensors. We can first get the search space of sub-tasks Y, which is based on the first-level sparse tiles. Let U be the set of points that O cannot skip in its iteration space to get the correct output (Definition 3). We set U' to be U initially, which will be updated when selecting sub-tasks. Each sub-task $x_i \in Y$ covers a subset of U, denoted by u_i. We iteratively select sub-tasks to cover all the points in U. At each iteration, we calculate the average cost for x_i by dividing $cost(x_i)$ by $|u_i \cap U'|$ (i.e., the number of the uncovered points of U that x_i can cover). The sub-task with the minimum average cost, x_i, is selected and U' is updated to be $U' - u_i$. As we allow multi-level sparse tiles, after we select a sub-task, we will reorganize the remaining non-zeros of the sparse tensors and generate next-level sparse tiles (using the method in Section 3.1) and the corresponding new search space of sub-tasks. Then we select the next sub-task from the new sub-tasks. This algorithm stops when there are no uncovered points in U. Based on the above basic workflow, we use some techniques for better effectiveness and efficiency, and Algorithm 2 shows the pseudocode.

Algorithm 2: Greedy with Withdraw

```
Data: operator $O$, ratio $r$ for local search
Result: task partition $X$
1  $X \leftarrow \emptyset$;
2  $Y \leftarrow$ first-level sub-tasks of $O$;
3  $Y' \leftarrow$ subset of $Y$ we may use to replace sub-tasks in $X$;
4  $U \leftarrow$ the set of points $O$ cannot skip in the iteration space;
5  $U' \leftarrow U$;
6  while $|U'| \neq 0$ do
7      $x_i \leftarrow \arg\min_{x_j \in Y} \frac{cost(x_j)}{|u_j \cap U'|}$, $c \leftarrow \min_{x_j \in Y} \frac{cost(x_j)}{|u_j \cap U'|}$;
       /* $Z_j \subseteq X$ is the sub-tasks $x_j$ can replace */
8      $c_1 \leftarrow \min_{x_j \in Y'} \frac{cost(x_j) - \sum_{x_k \in Z_j} cost(x_k)}{|u_j \cap U'|}$;
9      if $c_1 \leq c$ then
10         $x_i \leftarrow \arg\min_{x_j \in Y'} \frac{cost(x_j) - \sum_{x_k \in Z_j} cost(x_k)}{|u_j \cap U'|}$, $c \leftarrow c_1$;
11     end
12     $X' \leftarrow$ sub-tasks of $Y$ with average costs $\leq rc$;
13     $U' \leftarrow U' - \bigcup_{x_i \in X'} u_i$, $X \leftarrow X'$;
14     Update $Y, Y'$ with the next-level sub-tasks of $O$;
15 end
```

Withdraw sub-tasks (line 8-11, Algorithm 2). The greedy algorithm is sometimes short-sighted. Consider the example in Figure 6. We use rectangles to represent U and the subsets of U covered by sub-tasks. Suppose we already select two sub-tasks x_1, x_2, and the corresponding u_1, u_2 are colored in Figure 6. Suppose the current best sub-task to select is x_3 whose average cost is 1.2. Suppose there is only one SM on the GPU we run. Then we can get a task partition of the original operator with a cost of $8 + 3 + 6 = 17$. However, if we withdraw x_1, x_2 and select x_4 as shown on the lower part



|u1|=40; atomic [cost = 8]
|u2|= 10; atomic [cost=3]
|u3|= 5; atomic [cost=6]
avg_cost = 6/5 = 1.2
√
|u4|= 55; non-atomic [cost = 12]
avg_cost = [12-(8+3)]/5 = 0.2

Figure 6: Example to motivate withdrawing sub-tasks.

of Figure 6, we can get a task partition with a cost of 12, which is better. Therefore, we adapt the "withdraw" idea from [17] to our greedy algorithm. Specifically, we only consider the block-format non-atomic sub-tasks as the candidates to replace other selected sub-tasks. This is because the example in Figure 6 is due to the unbalanced non-zero distribution of the sparse tensors, and block-format sub-tasks are more likely to encounter this problem as they are obtained without compressing the non-zeros in sparse tensors; on the other hand, we are more interested in using non-atomic sub-tasks to replace atomic sub-tasks because atomic operations are expensive. We compute the average cost of a sub-task x_j to replace a subset of selected sub-tasks Z_j by

$$\frac{cost(x_j) - \sum_{x_k \in Z_j} cost(x_k)}{u_j \cap U'}$$

In the example of Figure 6, the average cost of x_4 to replace x_1, x_2 is 0.2, so using this average cost definition, we will select x_4 instead of x_3 in this iteration.

Local search (line 12, Algorithm 2). To speed up the selection process, every time we select a sub-task x_i, we also search for other sub-tasks whose average costs are close to x_i: less than r times the average cost of x_i, we will select x_j together. Here r is a user-provided parameter like 120%.

3.4 Theoretical Analysis.

Theorem 3.1 analyzes the effectiveness of the greedy algorithm for a simplified version of the multi-level sparse format decomposition problem: only first-level sparse tiles and the corresponding sub-tasks are considered. The proof can be found in our technical report [3].

Theorem 3.1. *The approximation ratio of Algorithm 2 for the simplified problem is $r\beta \ln(|U|)$, where U is the set of points that the operator cannot skip in its iteration space to get the correct output, r is the ratio parameter for local search and $\beta < 2$ in most cases.*

Proof. The GPU has n SMs in total: $W_1, ..., W_n$. Let X be the finally selected task partition and X^* be the optimal task partition; ALG be $\sum_{x_j \in X} cost(x_j)$ and OPT be $\sum_{x_j \in X^*} cost(x_j)$; $u'_j = u_j \cap U'$ be the subset of U that $x_j \in X$ is the first picked sub-task to cover; x_i be $\arg\min_{x_j \in X} cost(x_j)/|u'_j|$.

We have $\frac{cost(x_i)}{|u'_i|} \leq \frac{ALG}{|U|}$, so $cost(x_i) \leq \frac{|u'_i|}{|U|} ALG$. Suppose we schedule x_i as the last sub-task to run. The start time of x_i must be no later than $\sum_{x_j \in X - \{x_i\}} cost(x_j)/n$, so

$$cost(X) \leq \sum_{x_j \in X - \{x_i\}} cost(x_j)/n + cost(x_i) \leq (\frac{1}{n} + \frac{|u'_i|}{|U|}) ALG \quad (1)$$

Let e_k be the k^{th} point in U covered by the greedy algorithm, breaking ties arbitrarily. At the iteration that e_k is covered for the first time, $|U'| \geq |U| - k + 1$. Let X^*_{SUM} be the task partition with the minimum sub-task cost summation, i.e., $\arg\min_{X'} \sum_{x_j \in X'} cost(x_j)$; OPT_{SUM} be $\sum_{x_j \in X^*_{SUM}} cost(x_j)$; u_j be the subset of U that $x_j \in X^*_{SUM}$ covers. We can evenly assign $cost(x_j)$ to $u_j \cap U'$ for each $x_j \in X^*_{SUM}$. Therefore, there must be a $x_l \in X^*_{SUM}$ s.t. $\frac{cost(x_l)}{|u_l \cap U'|} \leq \frac{OPT_{SUM}}{|U'|}$. Let x_a be the first selected sub-task in X that covers e_k. Let $cost(e_k)$ be $\frac{cost(x_a)}{|u'_a|}$ or $\frac{cost(x_a) - \sum_{x_j \in Z_a} cost(x_j)}{|u'_a|}$, where Z_a is the sub-tasks replaced by x_a. We have $cost(e_k)/r \leq \frac{cost(x_l)}{|u_l \cap U'|} \leq \frac{OPT_{SUM}}{|U| - k + 1}$. Therefore, $ALG = \sum_{k=1}^{|U|} cost(e_k) \leq OPT_{SUM} \cdot r \ln(|U|) \leq OPT \cdot r \ln(|U|)$. From Equation (1), we have

$$cost(X) \leq (\frac{1}{n} + \frac{|u'_i|}{|U|}) r \ln(|U|) OPT = (1 + \frac{n|u'_i|}{|U|}) r \ln(|U|) \frac{OPT}{n}$$

$$cost(X) \leq (1 + \frac{n|u'_i|}{|U|}) r \ln(|U|) cost(X^*)$$

The subset of U covered by a sub-task in our search space is typically small compared with U in most cases, making $\frac{n|u'_i|}{|U|} < 1$. □

Time Complexity. The number of iterations in Algorithm 2 (line 6-15) is $O(|X|)$, where $|X|$ is the number of the finally selected sub-tasks and $|X| = O(|U|)$. In each iteration, we need to enumerate the candidate sub-tasks and compare their average costs. The number of candidate sub-tasks $|Y| = O(\prod_{i=1}^{d} \ell_i)$, where ℓ_i is the length of the i-th dimension of the sparse operator O's iteration space. We also need to update U' in each iteration, which takes $O(|U|)$. The total time complexity is $O(|U|(\prod_{i=1}^{d} \ell_i + |U|))$.

Memory Complexity. We need to store U, U' of the sparse operator O, the currently selected sub-tasks $|X|$, and the candidate sub-tasks in memory, where the u of a sub-task can be obtained from U so we do not need to store it in memory. The total memory complexity is $O(|U| + \prod_{i=1}^{d} \ell_i)$.

4 IMPLEMENTATION

We prepare templates for block-format, ELL-format, and 1D-format sub-tasks, respectively. The code of different sub-tasks is fused into a single GPU kernel through horizontal fusion [24]. For example, suppose we have two sub-tasks with different code and each sub-task requires a GPU thread block to execute its instructions, then the fused kernel combines the instructions for the two sub-tasks with branch statements. There will be two thread blocks to run the fused kernel, each for one sub-task. The branch condition checks the current thread block ID to dispatch the execution to the path of the instructions of the corresponding sub-task. In the general case, the number of branches in a fused kernel is limited by the number of templates (no larger than 32 in our experiments). These templates are parameterized for some implementation choices, such as the number of threads, the vectorized load/store size, and whether

Table 4: Tested sparse matrix information.

Sparse Matrix	M	N	#edges
cora [33]	2,708	2,708	10,556
citeseer [33]	3,327	3,327	9,228
pumbed [33]	19,717	19,717	88,651
ppi [15]	44,906	44,906	1,271,274
arxiv [20]	169,343	169,343	1,166,243
NotreDame (ND) [2]	325,729	325,729	1,497,134
proteins [20]	132,534	132,534	39,561,252
reddit [15]	232,965	232,965	114,615,892
unstructured	768-3,072	768-3,072	2,950-418,402
structured	128-3,072	128-3,072	6,144-503,808
LogSparse [25]	4,096	4,096	90,115
Strided [7]	4,096	4,096	766,144

to explicitly unroll the loops. Therefore, we can enumerate the parameters and select the best implementation according to the kernel latency. The parameter space is very small in the experiments (< 100), so the enumeration overhead is low.

For the cost model construction, the cost of a sub-task is measured by running the code based on the template as well. Specifically, we want to collect the running time of a sub-task when the GPU cores are saturated because otherwise estimating the running time of an SM based on summing the costs of sub-tasks up may be less accurate. Therefore, given a sub-task, we create a fake sparse operator whose computation workload can be partitioned into a large enough number of the sub-task and divide the total latency by (#sub-task/#SM) as the cost of the sub-task.

However, if a sparse operator is too sparse that the selected task partition does not have enough sub-tasks, the performance based on the above cost model may not be good enough. Therefore, besides the task partition found by the greedy algorithm, we also search for the best task partition with one format sub-tasks only, e.g., only block-format sub-tasks or only ELL-format sub-tasks or only 1D-format sub-tasks. We compare the latency of these three task partitions and select the best one from them to generate the final code for the sparse operator.

5 EXPERIMENTS

The experiments are designed to answer the following questions:

(1) Can STile outperform the competitors in operator efficiency?
(2) Are the techniques we use in the greedy algorithm useful?
(3) Are multi-level decomposition and more than one basic format necessary?

5.1 Experiment Setting

Environment. We implement our method based on SparseTIR [40] using Python 3.9. The experiments are done on a machine with two AMD EPYC 7413 24-Core Processors, one NVIDIA A100-SXM4-80GB, 1006 GB RAM, Ubuntu 20.04.5 and CUDA 11.7.

Competitors. There are five competitors in our experiments:

- cuSPARSE [11] is an Nvidia library that provides accelerated basic linear algebra subroutines for sparse matrices on GPU. We test the cuSPARSE kernels with the CSR format.
- DGL [35] is a popular framework for GNN. Its SDDMM implementation is based on FeatGraph [21].
- VectorSparse [6] adopts the 1D-vector formats and makes use of GPU tensor cores to accelerate sparse operators.
- Triton [34] is a tiling-based IR for deep learning, and we test its block sparse operator implementation.
- SparseTIR [40] is a state-of-the-art method that accelerates sparse operators in deep learning with composable formats and composable schedule transformations.
- STile is our method, where r of Algorithm 2 is set to 1.2.

The first four competitors only support single-sparse-format operators, while SparseTIR supports hybrid formats. As done in SparseTIR's experiments, we test Triton only on the block-pruned matrices and the sparse attention matrices. SparseTIR uses different sparse formats for different sparse matrices, and we also refer to its experiment in our comparison.

Datasets. We test the performance of our method on two kinds of operators: SpMM (Sparse Matrix Multiplication) and SDDMM (Sampled Dense-Dense Matrix Multiplication). Table 4 lists the information of the sparse matrices used in the experiments, which can be divided into four categories: (1) the **graph adjacency matrices** used for Graph Neural Networks with different sizes, densities and sparse patterns, i.e., the first eight sparse matrices (from cora to reddit) in Table 4; (2) the sparse matrices obtained by performing unstructured pruning to the weight matrices of Transformers, thus having unstructured sparse patterns, denoted by **"unstructured"** in Table 4; (3) the sparse matrices obtained by performing structured pruning to the weight matrices of Transformers, thus having structured sparse patterns (block-sparse patterns, specifically), denoted by **"structured"** in Table 4; (4) the **sparse attention matrices** used for Transformers with different sparse patterns, i.e., LogSparse, Strided in Table 4. There are 72 unstructured sparse matrices extracted from the movement-pruned PruneBERT [1] and 72 structured sparse matrices extracted from the block-pruned PruneBERT [2] with block size 32. The data type in the experiments is float16.

5.2 Main Results

This subsection compares our method and the competitors on SpMM and SDDMM in terms of the operator latency and the operator memory usage. We also report the search time of Algorithm 2.

5.2.1 *SpMM.*

We select cuSPARSE, VectorSparse, Triton, SparseTIR as the competitors, where Triton is only used for the structured sparse matrices, LogSparse and Strided. Figures 7 and 8 show the geometric mean speedups and the normalized memory usage values of different methods against cuSPARSE, respectively.

Graph Adjacency Matrix. For each graph adjacency matrix, the feature size of the SpMM operator is set to J, where $J \in \{32, 64, 128, 256, 512\}$. Our method achieves the best operator latency on each sparse matrix. We are $2.1 - 18.0\times$ better than cuSPARSE, $2.3 - 5.3\times$ better than VectorSparse, $1.4 - 2.7\times$ better than SparseTIR.

[1]https://huggingface.co/huggingface/prunebert-base-uncased-6-finepruned-w-distil-squad
[2]https://huggingface.co/madlag/bert-base-uncased-squad1.1-block-sparse-0.07-v1

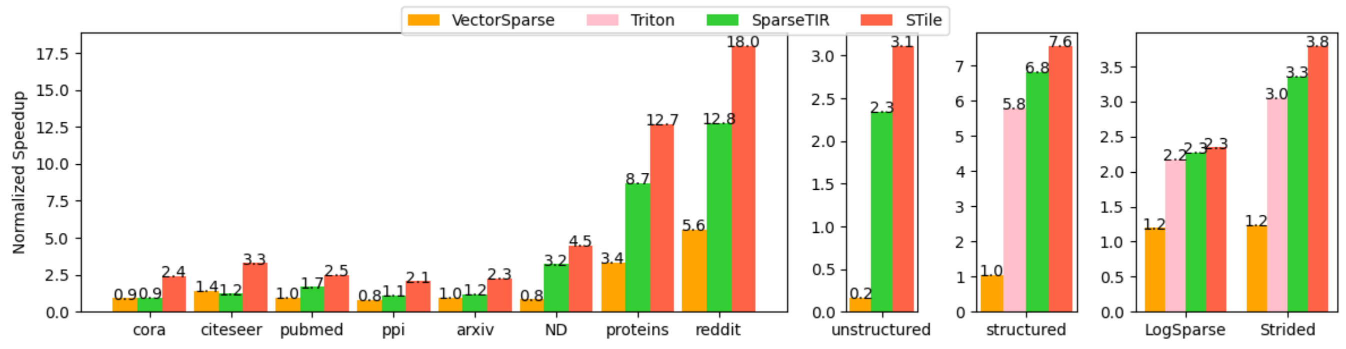


Figure 7: Normalized speedup against cuSPARSE for SPMM.

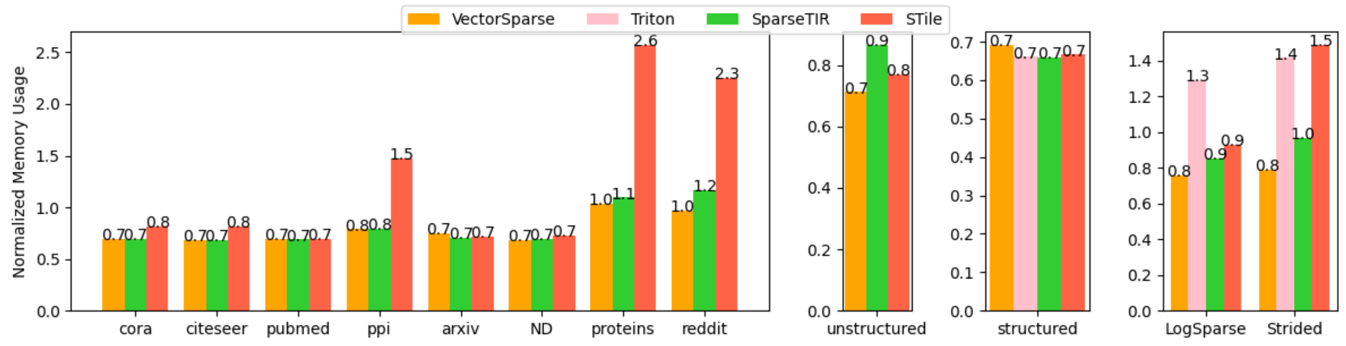


Figure 8: Normalized memory usage against cuSPARSE for SPMM.

SparseTIR uses the hyb format for the sparse matrices. The hyb format partitions the sparse matrix into several parts along columns first, then puts the rows into different buckets according to the #non-zeros per row. Each bucket is stored in a separate ELL format. The hyb format can effectively improve the cache locality and the load balance, avoiding having too much computation waste due to the padding in ELL at the same time. However, SparseTIR with hyb format does not use GPU Tensor Cores. VectorSparse adopts the 1D-vector format, where a vector is a column of elements with specific height (including both zeros and non-zeros in the sparse matrix). VectorSparse stores a sparse matrix into multiple vectors and uses Tensor Cores to do matrix multiplication. Tensor Cores are special computation units that are much more efficient than the GPU CUDA cores, but they only accept dense matrices of specific shapes as the input for matrix multiplication. This limits their application to sparse matrix operations. Although the 1D-vector format needs to store fewer zeros (i.e., less computation waste) than simply using the block-format for Tensor Core computation, it is still not good enough when the sparse matrix is not dense enough.

Compared with these competitors, STile has higher speedups for several reasons. First, we consider both the block-format sub-tasks and the ELL-format sub-tasks in our search space and try to find the best combination of them. In this way, we can use Tensor Cores only for the dense part of the sparse matrix, i.e., the block-format sub-tasks. Second, our templates for the ELL-format sub-tasks make use of the GPU shared memory to reduce the expensive data movement from the GPU global memory to the local registers. Third, we can reorder the matrix rows and columns to improve the local density of the sparse matrix, and our search space includes more possible block shapes for the block-format sub-tasks, which enables us to find better hybrid formats.

In terms of the operator memory usage, compared with the CSR format adopted by cuSPARSE, "hyb" and "ELL" do not need the row length array as their row lengths are constants; "1D-vector" and "block" store the positions for vectors or blocks (rather than for non-zero elements) in the CSR way, which cost less memory. However, CSR does not pad zeros, unlike all the other formats. This explains why VectorSparse, SparseTIR and STile can save memory on some datasets (e.g., cora, citeseer) compared with cuSPARSE, while not on other datasets (e.g., proteins, reddit). STile costs much more memory on ppi, proteins and reddit than all the baselines, because on these datasets, the majority of the selected sub-tasks adopt the block format. By contrast, the length of each 1D-vector used by VectorSparse is shorter than our block height, so it is more flexible than the blocks we use; the hyb format of SparseTIR is also more memory efficient as it requires fewer zero paddings. Although we store more zeros than other methods on these datasets, thanks to the Tensor Core computation enabled by the block format and our efficient kernel templates, we still achieve lower operator latency.

Unstructured Sparse Matrix. We extract 72 SpMM operators from the movement-pruned PruneBERT, with the sequence length set to 512. SparseTIR uses the BSR format or the SR-BCRS format [26] (choosing the better one from them). The BSR format stores the blocks of a sparse matrix with at least one non-zero. The SR-BCRS format is similar to the 1D-vector sparse format of VectorSparse, but it stores the vectors in a stride-wise row-major manner, while VectorSparse stores the vectors in a consecutive array. Both the BSR format and the SR-BCRS format of SparseTIR perform computation on Tensor Cores. Figure 7 shows that, on average, we are 3.1× better than cuSPAESE, 15.5× better than VectorSparse, and 1.3× better than SparseTIR. Although VectorSparse, SparseTIR, and our method all support Tensor Core computation in this part of the experiments, our operator latency is still better than theirs because we allow more flexible matrix reordering and more possible block-format sub-tasks, i.e., more possible computation schedules. In terms of operator memory usage, VectorSparse outperforms STile for the reason mentioned in the experiments on graph adjacency matrices. However, our block height is shorter than the BSR block height and the SR-BCRS vector length in SparseTIR, so our blocks are more flexible, which together with the flexible matrix reordering makes our format use less memory than SparseTIR's.

Structured Sparse Matrix. We extract 72 SpMM operators from the block-pruned PruneBERT (sequence length: 512). SparseTIR here uses the BSR format or the DBSR (doubly-compressed BSR) format for a sparse matrix. The DBSR format is similar to the BSR format but skips the all-zero rows. Triton is also included as the competitor in this part of the experiment, with the block size set to 32, which is exactly the block size used in the block pruning. Figure 7 shows that, on average, we are 7.6× better than cuSPARSE, 7.6× better than VectorSparse, 1.3× better than Triton, and 1.1× better than SparseTIR. The competitors, VectorSparse, Triton, and SparseTIR, all use Tensor Cores for computation. The reason that we perform better than them is similar to that for the experiments of unstructured sparse matrices. The operator memory usage values of all methods except cuSPARSE are close, because their formats (storing non-zero vectors or blocks) suit the block-sparse matrices well.

Sparse Attention Matrix. In this part of the experiment, the SpMM operator feature size is set to J, where $J \in \{32, 64, 128, 256, 512\}$. The sparse patterns of LogSparse and Strided consist of sliding windows and diagonals. Triton chooses its block size from 16 and 32. SparseTIR selects the best format from CSR, BSR, DBSR and SR-BCRS, and the formats selected are BSR (for one LogSparse SpMM) and SR-BCRS (for all other SpMMs). For Strided SpMMs, STile only selects block-format sub-tasks; for LogSparse SpMMs, our formats either consist of block basic formats only, or ELL basic formats only. Figure 7 shows that the operator latency we achive is 2.3−3.8× better than cuSPARSE, 2.0−3.1× better than VectorSparse, 1.1−1.2× better than Triton, 1.0 − 1.1× better than SparseTIR. On Strided, our format costs more memory than the SR-BCRS of SparseTIR, despite our shorter block height because we select blocks with large widths for better operator latency, which requires more padding.

5.2.2 SDDMM.

We now compare the operator latency and operator memory usage of our method and the competitors on SDDMM operators. We select DGL, VectorSpaese, SparseTIR (BSR), SparseTIR, and Triton as the competitors. SparseTIR (BSR) uses the BSR format for sparse matrices, while SparseTIR uses the 1D format for sparse matrices, which stores the non-zeros in a 1D array. SparseTIR (BSR) is not used for graph adjacency matrices. Triton is used for structured sparse matrices, LogSparse and Strided. Figures 9 and 10 show the geometric mean speedups and the normalized operator memory usage of different methods against DGL, respectively.

Graph Adjacency Matrix. For each graph adjacency matrix, the feature size of the SDDMM operator is set to K, where $K \in \{32, 64, 128, 256, 512\}$. Figure 9 shows that our method outperforms all the competitors on all the sparse matrices. Specifically, we are 1.5 − 6.9× better than DGL, 3.4 − 2283.5× better than VectorSparse, and 1.1 − 1.3× better than SparseTIR. For SDDMM, the computation workloads of different output points are independent of each other, and the workload amounts are the same. Thus, using the 1D format for a sparse matrix and assigning the computation of different output points to threads does not have the problem of workload balance. The optimization techniques like vectorized load/store intrinsics and parallelization on the reduction loop also help SparseTIR perform better. Therefore, SparseTIR achieves 1.4 − 5.6× speedup against DGL. VectorSparse runs the computation on Tensor Cores, and it also implements the reduction-loop parallelization, but its performance is much worse, which shows simply using the 1D-vector format is not a good choice for these tested matrices. Our method considers both block-format sub-tasks (running on Tensor Cores) and 1D-format sub-tasks in the search space, which helps us take advantage of both and perform better than the competitors. For example, on reddit with $K = 32$, if we only consider 1D-format sub-tasks, the SDDMM latency is 3.24 ms; if we only consider block-format sub-tasks, the latency is 2.66 ms; when considering both, the latency is 2.07 ms, i.e., 1.6× and 1.3× better than the first two cases, respectively. For the operator memory usage, the 1D format stores the row and column indices for each non-zero element, while the block format or the 1D-vector format uses less memory for positions of blocks or vectors but may store extra zeros. In Figure 10, SparseTIR performs the same as DGL because they both use the 1D format. VectorSparse performs worse (1.0 − 1.3×) because it selects a large vector length for better operator latency, causing a lot of padding. STile performs the same or better than DGL (0.9× on reddit) because our hybrid format saves memory for position information on denser parts and avoids padding on sparser parts.

Unstructured Sparse Matrix. The SDDMM operators take the unstructured sparse matrices as input, and the reduction loop length is set to $K = 512$. Figure 9 shows that we are 2.4× better than DGL, 4.8× better than VectorSparse, 3.4× better than SparseTIR (BSR), and 1.1× better than SparseTIR. The hybrid formats finally selected by our method only include the 1D-format sparse tiles. However, we still outperform SparseTIR because the parameter space of our templates for the 1D-format sub-tasks is larger than SparseTIR. Although STile, SparseTIR and DGL all use the 1D format, STile and SparseTIR lead to less operator memory because DGL pads the two input matrices for multiplication to be of the same shape.

Structured Sparse Matrix. The SDDMM operators take the structured sparse matrices as input, and the reduction loop length is set to $K = 512$. Figure 9 shows that the operator latency we achieve is 4.3× better than DGL, 3.6× better than VectorSparse, 1.7× better than SparseTIR (BSR), 1.6× better than SparseTIR, and 1.1× better

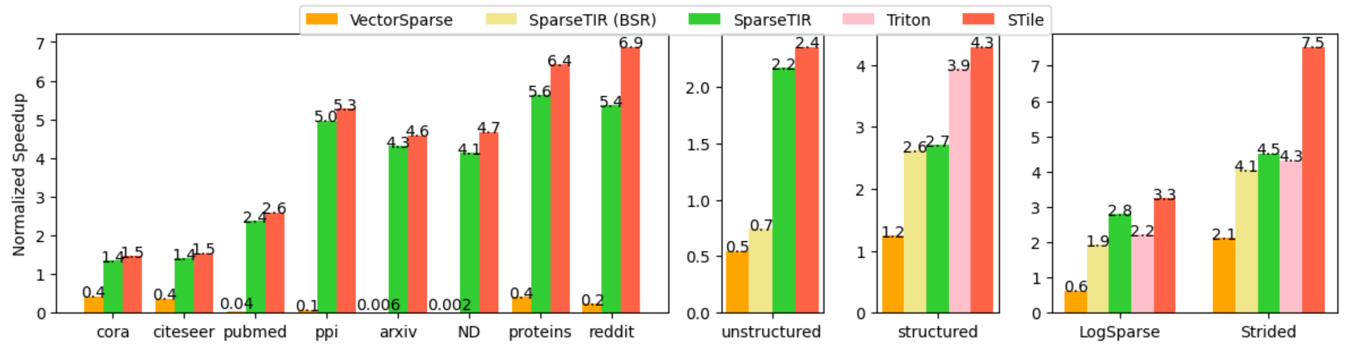


Figure 9: Normalized speedup against DGL for SDDMM.

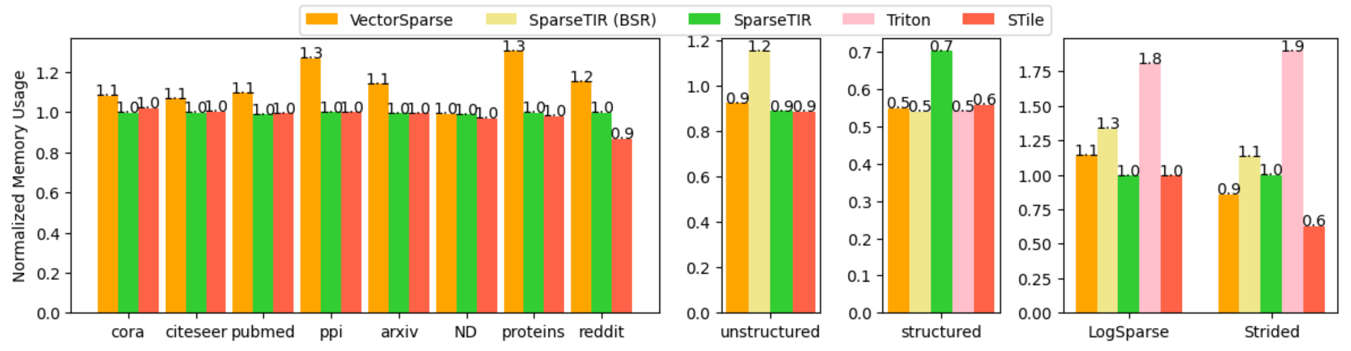


Figure 10: Normalized memory usage against DGL for SDDMM.

Table 5: The search time (seconds) of Algorithm 2.

Sparse Matrix	SpMM	SDDMM
cora	0.5	0.3
citeseer	0.6	0.6
pumbed	93.9	4.9
ppi	88.0	28.2
arxiv	1154.7	247.9
ND	3015.7	732.8
proteins	1161.8	1242.6
reddit	2831.0	1772.5
unstructured	0.1-12.9	0.1-3.4
structured	0.03-0.4	0.03-0.9
LogSparse	1.0	0.7
Strided	2.3	1.7

than Triton. We are better than SparseTIR (with 1D format) because we support Tensor Cores, which are suitable for block-sparse patterns. We are better than other competitors that also run on Tensor Cores because our candidate block-format sub-tasks in the search space implemented with our templates require fewer resources (e.g., threads) and are more efficient. The operator memory usage values of all the methods are only $0.5 - 0.7\times$ of DGL.

Sparse Attention Matrix. The SDDMM operators take the sparse attention matrices as input, and the feature size is set to K, where $K \in \{32, 64, 128, 256, 512\}$. Figure 9 shows that the operator latency we achieve is $3.3 - 7.5\times$ better than DGL, $3.5 - 5.2\times$ better than VectorSparse, $1.7 - 1.9\times$ better than SparseTIR (BSR), $1.2 - 1.7\times$ better than SparseTIR, and $1.5 - 1.7\times$ better than Triton. We only select 1D-format spare tiles for LogSparse. For Strided, we select both block-format and 1D-format sparse tiles, which also makes our format the most memory-efficient among the competitors.

5.2.3 Search Time. Table 5 shows the average search time of Algorithm 2 for SPMM and SDDMM operators. The search time is consistent with our time complexity analysis in Section 3.4. We can finish the search for any of the tested datasets in one hour. For those small sparse matrices (e.g., citeseer, cora), the search time is less than one second. Our search time is acceptable as it can be amortized when the optimized sparse operators are executed multiple times during the inference or training tasks. Other competitors do not need tuning or only tune in a very small parameter space, so we do not report their search time results.

5.3 Ablation Study

This section studies the influence of the optimization techniques we propose for Algorithm 2: withdrawing sub-tasks and local search.

Table 6: Normalized latency and search time against the default setting of STile.

Sparse Matrix	w/o withdrawing	w/o local search	
	Latency	Latency	Search Time
cora	1.0	1.0	1.0
citeseer	1.0	1.0	1.3
pumbed	1.0	1.0	2.2
ppi	1.4	1.2	61.7
arxiv	1.1	1.0	22.1
ND	1.2	1.2	56.1
proteins	1.5	1.0	213.5
reddit	1.1	1.0	267.4

5.3.1 With withdrawing v.s. without withdrawing. We compare the performance of our greedy algorithm with and without the withdrawing step for SpMMs on the graph adjacency matrices. The feature size is set to $J = 32$. The second column of Table 6 shows the normalized latency against the "with withdrawing" setting. Without withdrawing, the SpMM latency is $1.0 - 1.5\times$ higher, which verifies that the withdrawing step can effectively improve the performance of the greedy algorithm. When analyzing the finally selected sub-tasks in different settings, we find that when without withdrawing, the greedy algorithm tends to select more atomic sub-tasks, which makes the latency higher. For example, on proteins, without withdrawing, there are 81745 atomic block-format sub-tasks selected by the greedy algorithm; while with withdrawing, only one atomic block-format sub-task is selected by the greedy algorithm.

5.3.2 With local search v.s. without local search. The local search ratio r in Algorithm 2 is set to 1.2 by default. In this part of experiments, we compare the performance of Algorithm 2 when with and without local search on the SpMM operators taking the graph adjacency matrices as input, and report the results in Table 6 (the last two columns). When not doing the local search, the SpMM latency is the same or even higher, e.g., 1.2× of the latency when doing the local search. By looking at the sub-tasks selected for ppi, we find that Algorithm 2 selects more atomic sub-tasks when not doing the local search, so the latency is higher. Regarding the search time, doing local search greatly speeds up the search process, i.e., $1.0 - 267.4\times$ speedup. Local search enables us to tackle the optimization of larger sparse operators.

5.4 Case Study

To show the importance of multi-level decomposition for SpMM, we test our method on a sparse matrix with the sparse pattern shown in Figure 11: part of the non-zeros fall in the 2D green blocks, and the others fall in the 1D yellow blocks. No matter how we reorder and partition the spare matrix, the sub-matrix covering the green non-zeros, bordered in red, always intersects the sub-matrix covering the yellow non-zeros, bordered in purple. We use an SpMM with the feature size 32 for the test. Figure 12(a) and Figure 12(b) show how the sparse matrix is stored by the hybrid format found by single-level decomposition and by the multi-level decomposition, respectively. Specifically, we zoom in on the top left corner of the matrix to see details. The elements in the same color are stored in the same basic format. As single-level decomposition cannot do the non-zero reorganization, it has to select some sparse tiles with overlaps or zeros: Figure 12 shows that the hybrid format it selects stores more zeros than the multi-level decomposition, causing much computation waste. The real latency by multi-level decomposition is $0.62ms$, 3.6× faster than that by single-level decomposition ($2.24ms$).

5.5 Search Space Study

This section shows how the search space depth and breadth influence the final operator latency and the search time of Algorithm 2. Specifically, the search space depth is controlled by the number of decomposition levels, which is set to values in $\{1, 2, 10, 100, \infty\}$, where ∞ is adopted by STile by default. The search space breadth is controlled by the set of basic sparse formats we consider, which is varied from one basic format to two basic formats (the default setting). Table 7 shows the results on two datasets with natural hybrid sparse patterns (reddit, ND) and two datasets with manually designed hybrid sparse patterns (LogSparse, Strided). The feature size is set to 64 for SpMM and 256 for SDDMM.

For both SpMM and SDDMM, the operator latency generally improves as the depth increases and becomes stable after the depth is large enough (the default setting is $1.0 - 1.7\times$ better than the single-level setting). For SpMM on ND, the operator latency when the depth is 10 is longer than that when the depth is 2 because more atomic sub-tasks are selected when the depth is 10, which is possible because Algorithm 2 is not guaranteed to find the optimal task partition. The best depth for SpMM on LogSparse and Strided is 1 (i.e., all sparse tiles are extracted from the original sparse matrix), so changing depth does not influence the operator latency. The best depth is determined by the number of distinct sparse patterns in an operator, which varies from 1 to 24 for the tested operators in Section 5.2. With the depth set to ∞, our algorithm will adaptively explore the depth according to the cost model. However, although increasing depth extends the search space, the search time of Algorithm 2 does not necessarily increase, e.g., SDDMM on Strided. The reason is that although setting a larger depth increases the time of generating next-level sparse tiles, the remaining non-zeros will be reorganized in this process, making the new sparse tiles more effective in covering these non-zeros, which speeds up the search process, as some old sparse tiles may become partially useful if the non-zeros they cover have already been covered by the selected sparse tiles. For example, for SDDMM on Strided, the search process under the single-level setting takes 15 iterations and selects 13720 sparse tiles in total, while the search process of the multi-level setting takes 2 iterations only and selects 9848 sparse tiles in total, which explains the search time difference.

Increasing the search space breadth improves the operator latency for both SpMM and SDDMM in general (considering two basic formats is $1.0 - 2.6\times$ better than considering one basic format). Therefore, we appreciate the set of basic sparse formats to be efficient for different sparse patterns, e.g., the block format and ELL for SpMM work well on high density and low density, respectively, so their combinations work well on various sparse patterns. On the other hand, setting the search space breadth to 1 significantly reduces the search space size, so the search time of Algorithm 2 is $0.002 - 0.7\times$ of the default setting for SDDMM on ND.

Table 7: Normalized latency and search time against the default setting of STile.

Sparse Matrix	Operator	Latency						Search Time					
		Level Num				Basic Format		Level Num				Basic Format	
		1	2	10	100	block	ELL/1D	1	2	10	100	block	ELL/1D
reddit	SpMM	1.1	1.0	1.0	1.0	1.1	1.9	1.2	0.8	0.9	1.0	0.2	0.1
	SDDMM	1.4	1.3	1.0	1.0	2.3	1.1	2.0	1.8	1.0	1.0	0.3	0.1
ND	SpMM	1.7	1.4	1.6	1.0	1.1	1.3	1.7	2.7	1.4	1.0	0.1	0.01
	SDDMM	1.1	1.1	1.0	1.0	1.6	1.1	2.7	2.1	1.0	1.0	0.1	0.002
LogSparse	SpMM	1.0	1.0	1.0	1.0	1.0	1.0	0.9	1.0	1.0	1.0	0.7	0.1
	SDDMM	1.3	1.0	1.0	1.0	2.1	1.0	1.3	1.1	1.1	1.1	0.2	0.1
Strided	SpMM	1.0	1.0	1.0	1.0	1.0	2.0	1.0	1.0	1.0	1.0	0.7	0.3
	SDDMM	1.2	1.0	1.0	1.0	2.6	1.4	2.5	1.1	1.1	1.0	0.2	0.4

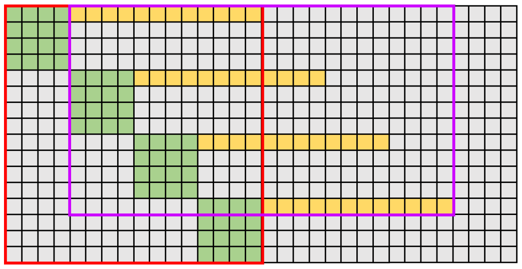


Figure 11: A sparse pattern we use to verify the necessity of multi-level decomposition. Some non-zeros fall in the green 2D blocks and the remaining non-zeros are in the yellow 1D blocks. The sub-matrix containing the green non-zeros (bordered in red) always intersects the sub-matrix containing the yellow non-zeros (bordered in purple), no matter how we reorder and partition the matrix.

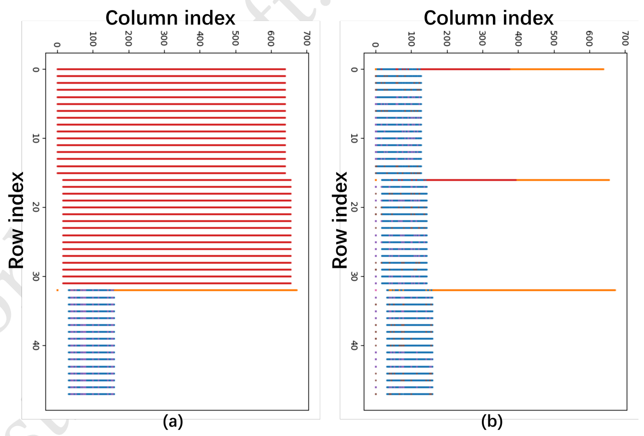


Figure 12: Illustration of how a sparse matrix is stored in (a) the format obtained in the single-level setting and (2) the format obtained in the multi-level setting. Elements in the same color are in the same basic format. The single-level decomposition leads to more padding than the multi-level decomposition, which causes computation waste.

Figure 13 shows the sparse formats for SDDMM on Strided that we can obtain in different settings (we use a small matrix with the Strided sparse pattern for illustration). The green elements are stored in the block format, while the blue ones are in the 1D format. In the default setting, we can separate the dense part and the sparse part of the sparse matrix *A* well into two parts and store them in suitable formats respectively. When the depth is 1, to store the non-zeros in the sparser part, the format has to redundantly store some non-zeros as we cannot generate new 1D-format sparse tiles for the remaining non-zeros, which wastes computation. Only considering the block basic format also wastes computation; while only considering the 1D basic format fails to use the Tensor Cores.

5.6 Cost Model Performance

To show that our cost model is effective, we randomly sample 5000 sub-tasks for SpMM and 13060 sub-tasks for SDDMM from the sub-tasks we select. The sampled sub-tasks are all ELL-format for SpMM and 1D-format for SDDMM, because for block-format sub-task, we already built a cost table to look for their costs. We measure the latency of each sub-task on GPU when all the GPU cores are saturated using the method in Section 4. Figure 14 compares the real latency values and the predicted costs of our cost model for the ELL-format sub-tasks and for the 1D-format sub-tasks, respectively. There is a linear correlation between our prediction and the real

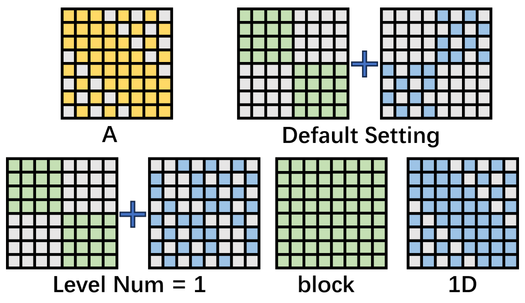


Figure 13: The sparse formats obtained in different settings.

value. The Pearson correlation coefficients of the cost model for the ELL-format sub-tasks and for the 1D-format sub-tasks are 0.9243 and 0.9997, respectively, and the p-values are 0.0 for both. The RMSE results are 1.4e-4 and 4.8e-05 for these two types of sub-tasks,

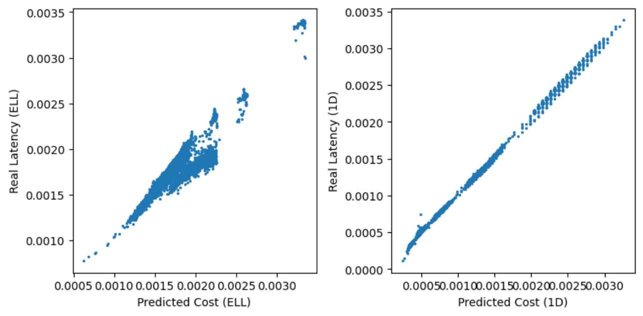


Figure 14: 1D-format sub-task: real latency v.s. predicted cost.

respectively. The cost model performs very well on 1D-format sub-tasks because we only consider one type of 1D-format sparse tiles in our search space. On the other hand, although the cost model for ELL-format sub-tasks is less accurate (the empirical approximation bound is 24%), given that block-format sub-tasks with Tensor Core computation can be a magnitude faster, this cost model still helps us find good hybrid formats for SpMMs. We also try MLP [29] as the cost model for ELL-format sub-tasks. The RMSE of MLP is 3.3e-05, which is almost a magnitude smaller than ours. The operator latency values we achieve on the datasets in Section 5.2 are the same as those when using our cost model, but the search time increases a lot ($1.1 - 6.1\times$) because we need to compute more features for MLP to do prediction. Therefore, our cost model works well in terms of both the operator latency and the efficiency of doing prediction.

6 RELATED WORKS

Sparsity pattern. There are two types of sparse patterns in deep learning in general: structured sparse patterns and unstructured sparse patterns. The structured sparsity patterns are manually designed. E.g., the fixed and strided sparse attention patterns [8]; the "global+sliding window" attention pattern in Longformer [4]; the butterfly attention matrix in Pixelated Butterfly Transformer [5]. Besides, the model pruning also generates sparse weight matrices, e.g., the row or column-based pruning [28], Block Pruning [23]. On the other hand, the unstructured sparse patterns may come from the real-world datasets, e.g., the sparse graph adjacency matrices for GNNs, or from the unstructured network pruning, which does not have a requirement for the pattern of the pruned weights. Magnitude Pruning [16] and Movement Pruning [32] define the weight importance score and only keep the important weights in the weight matrix. For unstructured sparse patterns, reorganizing the non-zeros by transformation is important for a good format.

Sparse format and implementation. A lot of sparse formats have been proposed, e.g., CSR, BSR, ELL [10]. BELL [9] stores the non-zero block in the ELL way. Labini et al. [22] use a 1-D reordering-based blocking algorithm to build dense blocks from arbitrary sparse matrices. ALTO [18] stores the nonzeros of a given tensor in a 1-D data structure along with compact indexing metadata, such that neighboring non-zeros in the tensor are close to each other in memory. CSR5 [27] partitions all nonzero entries to multiple 2-D blocks of the same size for better load balance. TC-GNN [36] proposes to condense the sparse matrix in the row window level for a better block density. CHOU et al. [10] summarize a format abstraction to express common sparse formats with the same interface. A lot of implementation techniques have also been proposed to generate efficient GPU kernels for sparse operators. The segmented sum for workload balance in CSR-based SpMV (sparse matrix-vector multiplication) [27]. Vector memory instructions mitigate bandwidth bottlenecks and reduces the instructions required [14]. Subwarp tiling and reverse offset memory alignment [14] solve the problem of too short rows for vectorization and misaligned address when using vector instruction. Shared memory improves data reuse [14, 19, 26] and helps prepare data for Tensor Core [26, 36]. Tensor Cores suit the block format and its variants [22, 36, 37]. uGrapher [43] implements templates with different parallelization strategies and predicts the best strategy with machine learning. WACO [39] optimizes the sparse format and the implementation together, but does not support hybrid formats.

Hybrid sparse format. A lot of works partition the sparse matrix in different ways and use different formats for each part, e.g., 1D-VBR [1] partitions rows into groups of different sizes, Tile-SpGEMM [31], TileSpMV [30] partition the sparse matrix into blocks, ASpT [19] reorders the columns in each row panel to form dense blocks and sparse blocks, SELL [9] and hyb [40] divided the sparse matrix into multiple row groups and use a separate ELL format for each row group to reduce zero paddings, and AlphaSparse [12] uses transformation primitives to partition the matrix more flexibly and generate efficient code. Besides, SparTA [42] combines block formats with different block sizes. Compared with these works, STile supports more possible sparse tiles and more flexible combinations of different formats (i.e., multi-level format decomposition), and we use parameterized implementation templates for each basic sparse format.

7 CONCLUSION

In this paper, we introduce the MSFD problem and prove its hardness. We propose a framework STile to automatically find the best hybrid format for a given sparse operator. We make our search space large by using rules to transform and partition the sparse tensor flexibly and by considering multi-level sparse tiles. A greedy-with-withdraw method is used to do the search, based on a cost model for sub-tasks. Experiments show that STile achieves $2.1 - 18.0\times$ speedup against cuSPARSE on SpMMs and $1.5 - 6.9\times$ speedup against DGL on SDDMM. The search time is less than one hour for any tested sparse operator, which can be amortized.

REFERENCES

[1] Peter Ahrens and Erik G. Boman. 2020. On Optimal Partitioning For Sparse Matrices In Variable Block Row Format. *CoRR* abs/2005.12414 (2020). arXiv:2005.12414 https://arxiv.org/abs/2005.12414

[2] Réka Albert, Hawoong Jeong, and Albert-László Barabási. 1999. Diameter of the world-wide web. *nature* 401, 6749 (1999), 130–131.

[3] The Authors. 2023. *Technical report.* Retrieved Oct 20, 2023 from https://github.com/ExperimentAnonymous/STile/blob/main/technical_report.pdf

[4] Iz Beltagy, Matthew E. Peters, and Arman Cohan. 2020. Longformer: The Long-Document Transformer. *CoRR* abs/2004.05150 (2020). arXiv:2004.05150 https://arxiv.org/abs/2004.05150

[5] Beidi Chen, Tri Dao, Kaizhao Liang, Jiaming Yang, Zhao Song, Atri Rudra, and Christopher Ré. 2022. Pixelated Butterfly: Simple and Efficient Sparse training for Neural Network Models. In *The Tenth International Conference on Learning*

Representations, ICLR 2022, Virtual Event, April 25-29, 2022. OpenReview.net. https://openreview.net/forum?id=Nfl-iXa-y7R
[6] Zhaodong Chen, Zheng Qu, Liu Liu, Yufei Ding, and Yuan Xie. 2021. Efficient tensor core-based gpu kernels for structured sparsity under reduced precision. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*. 1–14.
[7] Rewon Child, Scott Gray, Alec Radford, and Ilya Sutskever. 2019. Generating long sequences with sparse transformers. *arXiv preprint arXiv:1904.10509* (2019).
[8] Rewon Child, Scott Gray, Alec Radford, and Ilya Sutskever. 2019. Generating Long Sequences with Sparse Transformers. *CoRR* abs/1904.10509 (2019). arXiv:1904.10509 http://arxiv.org/abs/1904.10509
[9] JeeWhan Choi, Amik Singh, and Richard W. Vuduc. 2010. Model-driven autotuning of sparse matrix-vector multiply on GPUs. In *Proceedings of the 15th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, PPOPP 2010, Bangalore, India, January 9-14, 2010*, R. Govindarajan, David A. Padua, and Mary W. Hall (Eds.). ACM, 115–126. https://doi.org/10.1145/1693453.1693471
[10] 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.
[11] NVIDIA Corporation. 2022. *cuSPARSE :: CUDA Toolkit Documentation v11.7.1*. Retrieved July 15, 2023 from https://docs.nvidia.com/cuda/cusparse/index.html
[12] Zhen Du, Jiajia Li, Yinshan Wang, Xueqi Li, Guangming Tan, and Ninghui Sun. 2022. AlphaSparse: Generating High Performance SpMV Codes Directly from Sparse Matrices. In *SC22: International Conference for High Performance Computing, Networking, Storage and Analysis, Dallas, TX, USA, November 13-18, 2022*. IEEE, 1–15. https://doi.org/10.1109/SC41404.2022.00071
[13] 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*, Christine Cuicchi, Irene Qualters, and William T. Kramer (Eds.). IEEE/ACM, 17. https://doi.org/10.1109/SC41405.2020.00021
[14] 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.
[15] Will Hamilton, Zhitao Ying, and Jure Leskovec. 2017. Inductive representation learning on large graphs. *Advances in neural information processing systems* 30 (2017).
[16] Song Han, Jeff Pool, John Tran, and William J. Dally. 2015. Learning both Weights and Connections for Efficient Neural Networks. *CoRR* abs/1506.02626 (2015). arXiv:1506.02626 http://arxiv.org/abs/1506.02626
[17] Refael Hassin and Asaf Levin. 2005. A Better-Than-Greedy Approximation Algorithm for the Minimum Set Cover Problem. *SIAM J. Comput.* 35, 1 (2005), 189–200. https://doi.org/10.1137/S0097539704444750
[18] Ahmed E. Helal, Jan Laukemann, Fabio Checconi, Jesmin Jahan Tithi, Teresa M. Ranadive, Fabrizio Petrini, and Jeewhan Choi. 2021. ALTO: adaptive linearized storage of sparse tensors. In *ICS '21: 2021 International Conference on Supercomputing, Virtual Event, USA, June 14-17, 2021*, Huiyang Zhou, Jose Moreira, Frank Mueller, and Yoav Etsion (Eds.). ACM, 404–416. https://doi.org/10.1145/3447818.3461703
[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 ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, PPoPP 2019, Washington, DC, USA, February 16-20, 2019*, Jeffrey K. Hollingsworth and Idit Keidar (Eds.). ACM, 300–314. https://doi.org/10.1145/3293883.3295712
[20] Weihua Hu, Matthias Fey, Marinka Zitnik, Yuxiao Dong, Hongyu Ren, Bowen Liu, Michele Catasta, and Jure Leskovec. 2020. Open graph benchmark: Datasets for machine learning on graphs. *Advances in neural information processing systems* 33 (2020), 22118–22133.
[21] Yuwei Hu, Zihao Ye, Minjie Wang, Jiali Yu, Da Zheng, Mu Li, Zheng Zhang, Zhiru Zhang, and Yida Wang. 2020. FeatGraph: a flexible and efficient backend for graph neural network systems. 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*, Christine Cuicchi, Irene Qualters, and William T. Kramer (Eds.). IEEE/ACM, 71. https://doi.org/10.1109/SC41405.2020.00075
[22] Paolo Sylos Labini, Massimo Bernaschi, Francesco Silvestri, and Flavio Vella. 2022. Blocking Techniques for Sparse Matrix Multiplication on Tensor Accelerators. *CoRR* abs/2202.05868 (2022). arXiv:2202.05868 https://arxiv.org/abs/2202.05868
[23] François Lagunas, Ella Charlaix, Victor Sanh, and Alexander M. Rush. 2021. Block Pruning For Faster Transformers. In *Proceedings of the 2021 Conference on Empirical Methods in Natural Language Processing, EMNLP 2021, Virtual Event / Punta Cana, Dominican Republic, 7-11 November, 2021*, Marie-Francine Moens, Xuanjing Huang, Lucia Specia, and Scott Wen-tau Yih (Eds.). Association for Computational Linguistics, 10619–10629. https://doi.org/10.18653/v1/2021.emnlp-main.829
[24] Ao Li, Bojian Zheng, Gennady Pekhimenko, and Fan Long. 2022. Automatic Horizontal Fusion for GPU Kernels. In *IEEE/ACM International Symposium on Code Generation and Optimization, CGO 2022, Seoul, Korea, Republic of, April 2-6, 2022*, Jae W. Lee, Sebastian Hack, and Tatiana Shpeisman (Eds.). IEEE, 14–27. https://doi.org/10.1109/CGO53902.2022.9741270
[25] Shiyang Li, Xiaoyong Jin, Yao Xuan, Xiyou Zhou, Wenhu Chen, Yu-Xiang Wang, and Xifeng Yan. 2019. Enhancing the locality and breaking the memory bottleneck of transformer on time series forecasting. *Advances in neural information processing systems* 32 (2019).
[26] Shigang Li, Kazuki Osawa, and Torsten Hoefler. 2022. Efficient Quantized Sparse Matrix Operations on Tensor Cores. In *SC22: International Conference for High Performance Computing, Networking, Storage and Analysis, Dallas, TX, USA, November 13-18, 2022*, Felix Wolf, Sameer Shende, Candace Culhane, Sadaf R. Alam, and Heike Jagode (Eds.). IEEE, 37:1–37:15. https://doi.org/10.1109/SC41404.2022.00042
[27] Weifeng Liu and Brian Vinter. 2015. CSR5: An efficient storage format for cross-platform sparse matrix-vector multiplication. In *Proceedings of the 29th ACM on International Conference on Supercomputing*. 339–350.
[28] JS McCarley, Rishav Chakravarti, and Avirup Sil. 2019. Structured pruning of a bert-based question answering model. *arXiv preprint arXiv:1910.06360* (2019).
[29] 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. https://doi.org/10.1109/ISPASS51385.2021.00016
[30] Yuyao Niu, Zhengyang Lu, Meichen Dong, Zhou Jin, Weifeng Liu, and Guangming Tan. 2021. TileSpMV: A Tiled Algorithm for Sparse Matrix-Vector Multiplication on GPUs. In *35th IEEE International Parallel and Distributed Processing Symposium, IPDPS 2021, Portland, OR, USA, May 17-21, 2021*. IEEE, 68–78. https://doi.org/10.1109/IPDPS49936.2021.00016
[31] Yuyao Niu, Zhengyang Lu, Haonan Ji, Shuhui Song, Zhou Jin, and Weifeng Liu. 2022. TileSpGEMM: a tiled algorithm for parallel sparse general matrix-matrix multiplication on GPUs. In *PPoPP '22: 27th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, Seoul, Republic of Korea, April 2 - 6, 2022*, Jaejin Lee, Kunal Agrawal, and Michael F. Spear (Eds.). ACM, 90–106. https://doi.org/10.1145/3503221.3508431
[32] Victor Sanh, Thomas Wolf, and Alexander M. Rush. 2020. Movement Pruning: Adaptive Sparsity by Fine-Tuning. In *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.). https://proceedings.neurips.cc/paper/2020/hash/eae15aabaa768ae4a5993a8a4f4fa6e4-Abstract.html
[33] Prithviraj Sen, Galileo Namata, Mustafa Bilgic, Lise Getoor, Brian Galligher, and Tina Eliassi-Rad. 2008. Collective classification in network data. *AI magazine* 29, 3 (2008), 93–93.
[34] Philippe Tillet, Hsiang-Tsung Kung, and David D. Cox. 2019. Triton: an intermediate language and compiler for tiled neural network computations. In *Proceedings of the 3rd ACM SIGPLAN International Workshop on Machine Learning and Programming Languages, MAPL@PLDI 2019, Phoenix, AZ, USA, June 22, 2019*, Tim Mattson, Abdullah Muzahid, and Armando Solar-Lezama (Eds.). ACM, 10–19. https://doi.org/10.1145/3315508.3329973
[35] Minjie Wang, Da Zheng, Zihao Ye, Quan Gan, Mufei Li, Xiang Song, Jinjing Zhou, Chao Ma, Lingfan Yu, Yu Gai, et al. 2019. Deep graph library: A graph-centric, highly-performant package for graph neural networks. *arXiv preprint arXiv:1909.01315* (2019).
[36] Yuke Wang, Boyuan Feng, and Yufei Ding. 2021. TC-GNN: Accelerating Sparse Graph Neural Network Computation Via Dense Tensor Core on GPUs. *CoRR* abs/2112.02052 (2021). arXiv:2112.02052 https://arxiv.org/abs/2112.02052
[37] Yang Wang, Chen Zhang, Zhiqiang Xie, Cong Guo, Yunxin Liu, and Jingwen Leng. 2021. Dual-side Sparse Tensor Core. In *48th ACM/IEEE Annual International Symposium on Computer Architecture, ISCA 2021, Valencia, Spain, June 14-18, 2021*. IEEE, 1083–1095. https://doi.org/10.1109/ISCA52012.2021.00088
[38] Samuel Williams, Andrew Waterman, and David A. Patterson. 2009. Roofline: an insightful visual performance model for multicore architectures. *Commun. ACM* 52, 4 (2009), 65–76. https://doi.org/10.1145/1498765.1498785
[39] 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*. 920–934.
[40] 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, ASPLOS 2023, Vancouver, BC, Canada, March 25-29, 2023*, Tor M. Aamodt, Natalie D. Enright Jerger, and Michael M. Swift (Eds.). ACM, 660–678. https://doi.org/10.1145/3582016.3582047
[41] Zhongming Yu, Guohao Dai, Guyue Huang, Yu Wang, and Huazhong Yang. 2021. Exploiting Online Locality and Reduction Parallelism for Sampled Dense Matrix Multiplication on GPUs. In *39th IEEE International Conference on Computer*

Design, ICCD 2021, Storrs, CT, USA, October 24-27, 2021. IEEE, 567–574. https://doi.org/10.1109/ICCD53106.2021.00092

[42] 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 *16th USENIX Symposium on Operating Systems Design and Implementation (OSDI 22)*. 213–232.

[43] Yangjie Zhou, Jingwen Leng, Yaoxu Song, Shuwen Lu, Mian Wang, Chao Li, Minyi Guo, Wenting Shen, Yong Li, Wei Lin, et al. 2023. uGrapher: High-Performance Graph Operator Computation via Unified Abstraction for Graph Neural Networks. In *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2*. 878–891.

Received 20 February 2007; revised 12 March 2009; accepted 5 June 2009