SparTA: Deep-Learning Model Sparsity via Tensor-with-Sparsity-Attribute

Ningxin Zheng, Microsoft Research; Bin Lin, Microsoft Research and Tsinghua University; Quanlu Zhang, Lingxiao Ma, Yuqing Yang, Fan Yang, Yang Wang, Mao Yang, and Lidong Zhou, Microsoft Research

https://www.usenix.org/conference/osdi22/presentation/zheng-ningxin
SparTA: Deep-Learning Model Sparsity via Tensor-with-Sparsity-Attribute

Ningxin Zheng1*, Bin Lin1,2*, Quanlu Zhang1, Lingxiao Ma1, Yuqing Yang1, Fan Yang1, Yang Wang1, Mao Yang1, Lidong Zhou1

1Microsoft Research, 2Tsinghua University

Abstract

Sparsity is becoming arguably the most critical dimension to explore for efficiency and scalability, as deep learning models grow significantly larger and more complex. After all, the biological neural networks, where deep learning draws inspirations, are naturally sparse and highly efficient.

We advocate an end-to-end approach to model sparsity via a new abstraction called Tensor-with-Sparsity-Attribute (TeSA), which augments the default Tensor abstraction that is fundamentally designed for dense models. TeSA enables the sparsity attributes and patterns (e.g., for pruning and quantization) to be specified, propagated forward and backward across the entire deep learning model, and used to create highly efficient, specialized operators, taking into account the execution efficiency of different sparsity patterns on different (sparsity-aware) hardware. The resulting SparTA framework can accommodate various sparsity patterns and optimization techniques, delivering $1.7x \sim 8.4x$ average speedup on inference latency compared to seven state-of-the-art (sparse) solutions with smaller memory footprints. As an end-to-end model sparsity framework, SparTA facilitates sparsity algorithms to explore better sparse models.

1 Introduction

As deep neural network (DNN) models become large and complex, they are inevitably getting sparse (or made sparse) for efficiency, just as manifested in the highly sparse biological neural networks [89]. A DNN model is usually modeled as a data flow graph (DFG), where each node is an operator with one or multiple input and output tensors. Model sparsity involves introducing some sparsity patterns on the tensors; for example, to quantize some tensors with lower precision (e.g., 16 to 8-bit); to prune the model by setting the value of some (or all) parts of some tensors to zero (e.g., block sparsity [61, 63] or fine-grained sparsity [43, 54, 55]); or to apply the combination of pruning and quantization to a model. With careful pruning and quantization, a DNN model can be compressed into a smaller memory footprint without losing too much accuracy. With DNN operators customized for the sparsity patterns, the resulting model will, hopefully, come with a lower inference latency.

Unfortunately, deep learning systems are not yet effective in exploiting sparsity: the increase in sparsity might not translate into actual gains in efficiency for a variety of reasons. First, the computation kernels for general sparse operations remain far from optimal. For example, cuSPARSE [3], the CUDA library for sparse matrix operations, has been shown to underperform cuBLAS, its dense counterpart, even when the sparsity of the matrices reaches 98% (Table 1). Second, as DNN computation usually takes multiple stages, the sparsity pattern might vary significantly across stages, making it hard to develop sparsity-aware optimizations for end-to-end gains. Finally, any effective sparsity-aware optimization might involve additional support across the vertical stack, from the deep learning framework, compiler, optimizer, operators and kernels, and all the way to hardware. Insufficient support at any of the layers could lead to inefficiency.

We therefore propose SparTA, a new framework that treats sparsity as a first-class citizen, with the following design principles. The design is customizable and extensible to accommodate new innovations on model sparsity; it is end-to-end and covers the whole-stack, rather than being limited to one operator or to one layer; it aims for extreme performance without sacrificing general applicability; it can facilitate existing sparsity algorithms to explore sparse models more efficiently.

At the core of SparTA is a new abstraction, Tensor-with-Sparsity-Attribute or TeSA, which augments the standard tensors with attributes to describe sparsity properties and patterns. Examples include low-precision weights, block (structured) sparsity, and fine-grained (unstructured) sparsity. A set of TeSA propagation rules guides the forward and backward propagation of sparsity attributes for end-to-end coverage. The rules can either be defined by the proposed TeSA algebra,
Table 1: Speed of matrix multiplication ($1024 \times 1024 \times 1024$) in cuSPARSE and cuBLAS on NVIDIA 2080Ti (unit: us).

<table>
<thead>
<tr>
<th>Sparsity Ratio</th>
<th>50%</th>
<th>90%</th>
<th>95%</th>
<th>99%</th>
</tr>
</thead>
<tbody>
<tr>
<td>cuSPARSE</td>
<td>1652.5</td>
<td>633.9</td>
<td>463.0</td>
<td>181.7</td>
</tr>
<tr>
<td>cuBLAS</td>
<td>208.3</td>
<td>208.3</td>
<td>208.3</td>
<td>208.3</td>
</tr>
</tbody>
</table>

or be inferred in a probabilistic way (§3.2).

With the sparse attributes in TeSA, SparTA can generate an efficient execution plan, taking into account the sparsity-aware hardware and specific sparse operators/kernels in certain sparsity patterns and conditions. SparTA may transform an execution plan to decompose complex sparsity attributes into a combination of simple ones with known effective optimizations. In the execution plan, SparTA can perform code specialization to generate efficient kernels for simple and regular sparse attributes, instead of resorting to generic but less efficient sparse kernels. This is how SparTA achieves extreme efficiency without sacrificing generality (§3.3).

Due to the whole-stack support (all the way to the codegen on accelerators), SparTA is able to provide the ground-truth performance metrics (e.g., latency) that can help evaluate different execution plans given a TeSA with fixed sparsity attributes and also offer valuable feedback for practitioners to search for the set of sparsity attributes with the ideal tradeoff between performance and accuracy (§5.4).

SparTA is highly customizable and extensible. With TeSA, one can define new sparsity properties and patterns for new ways of exploiting sparsity, provide new TeSA propagation rules, and incorporate new sparsity-aware operators, kernels, and (sparsity-aware) hardware accelerators.

We have implemented SparTA based on Rammer [60], a state-of-the-art open-source DNN compiler with no special support for sparsity. We extensively evaluate SparTA on three popular DNN models with four representative sparsity patterns on three accelerators (i.e., CUDA GPU, ROCm GPU, Intel CPU). Our evaluation shows that SparTA achieves up to 8.4x average speedup on model inference latency with less memory consumption, compared to seven state-of-the-art solutions (§5). We have also used SparTA to speed up sparse DNN model training and achieved more than 2x speedup than previous solutions (§5.5). By open sourcing SparTA\(^1\), we hope that this work can bring the community together in this extensible and unified framework to accelerate innovations on model sparsity.

2 Background and Motivation

The size of deep neural networks grows significantly over the past years [25, 37], which incurs large inference latency and heavy memory burden. Model sparsity is arguably the most critical dimension to explore for efficiency and scalability.

---

\(^1\)Code available at https://github.com/microsoft/SparTA
terns that are difficult for existing solutions to understand or optimize, leading to diminishing end-to-end return from sparsity.

In Figure 1, tensor \( W_2 \) illustrates a fine-grained sparsity pattern (63% sparsity). Such an initial sparsity pattern of \( W_2 \) incurs ripple effects. \( W_2 \) would propagate its sparsity attribute to the down-stream and up-stream tensors, including \( W_1, T_2, T_3, T_4, T_5, \) and \( W_5 \). For example, because the second column of \( W_2 \) is pruned, the second column of \( T_3 \) is destined to be all zero, hence can be pruned too (as \( T_2 \times T_2 = T_3 \)). Likewise, as the third row of \( W_2 \) is pruned, the third column of \( T_2 \) can also be pruned. It is therefore desirable for a deep learning compiler to understand such propagation of sparsity so as for further sparsity-aware optimization.

**Across-stack sparsity innovations in silos.** Due to the above limitations, sparsity innovations either are constrained to individual operators and evaluated with proxy metrics without knowing the end-to-end effects, or have to be implemented manually on a few neural models, difficult to be ported to other models [42, 77]. More problematically, individual solutions are hard to be extended to or combined with other proposals. All these motivate SparTA, a common foundation to facilitate sparsity innovations, which can be evaluated end-to-end.

### 3 SparTA Design

Figure 2 summarizes the overall architecture of SparTA. At the core of SparTA is the TeSA abstraction, which augments the existing tensor abstraction with *sparsity attribute* (§3.1). An algorithm designer can specify the sparsity patterns in selected tensors of a deep learning model as “Initial Tensor Sparsity Attribute”.

Given the initial sparsity attribute, SparTA performs *attribute propagation* to infer the sparsity attributes of all other tensors in the deep learning model, according to the propagation rules (§3.2). Sparsity attribute propagation exposes more optimization opportunities than the original sparse tensors, as shown, for example, in Figure 1.

After attribute propagation, SparTA runs a multi-pass compilation process to generate efficient end-to-end code (§3.3). Compared to a traditional DNN compiler, SparTA conducts two additional compilation passes to exploit model sparsity fully. The first pass transforms the original execution plan of a DNN model into a new one that takes advantage of the given sparsity patterns. A further compilation pass then performs sparsity-aware code specialization. The awareness of tensor sparsity patterns allow SparTA to generate highly customized code. This process may be iterated for further improvement.

Finally, with the final compiled code, model designers can profile the DNN model to obtain authentic performance metrics, including memory consumption and inference latency. Given the feedback, model designers may further update the sparsity attributes in some tensors and repeat this process iteratively to find the best tradeoff. Thus SparTA enables a feedback loop, facilitating the innovation in model sparsity.

#### 3.1 The TeSA abstraction

TeSA augments a traditional tensor with an additional tensor with the same shape, where each element is a scalar value, representing the sparsity attribute of the corresponding element in the original tensor. This allows a user to specify arbitrary sparsity patterns in a tensor, a key requirement of the evolving research on model sparsity [40, 47, 72]. Figure 3 shows an example of TeSA. The left shows the original dense tensor. The right shows the corresponding sparsity attribute, where one prunes the second row in the tensor, uses 8-bit to quantize the bottom-right element and 4-bit for the remaining elements. This example shows that TeSA can unify tensor quantization and pruning in one abstraction. The unified abstraction facilitates the co-optimization of pruning and quantization, e.g., picking the right block size to cover (represent) the remaining (non-pruned) elements while aligning with low-bit hardware instructions (e.g., *wmma* [5]). With TeSA, SparTA can understand the sparsity pattern at compile time, which enables further optimizations. Note that the sparse attribute will only be used in the compile phase, thus it does not impose additional resource burden to the actual compute phase.

#### 3.2 Sparsity Attribute Propagation

The number of tensors in a deep learning model is usually large. A user can only set the sparsity attribute for a subset of the tensors. To maximize end-to-end sparsity, SparTA performs attribute propagation along the DFG of the DNN model

![Figure 3: An example of TeSA. Sparsity Attribute denotes the sparsity pattern, including quantization (4 means *uint4*, 8 means *uint8*) and pruning (0 means the element is pruned).](image-url)
Algorithm 1: TeSA attribute propagation.

Data: G: DFG of TeSA annotated DNN model.
Result: G with updated TeSAs.

Function Propagate(G):
1. \( S = \text{Set}(\text{AllNodesOf}(G)) \);
2. while \( S \neq \emptyset \) do
3. \( N = S.\text{PopOneNode}(); \) /* can start from any node */
4. \( / / I, (O_i) \text{ is node } N \text{'s input (output) TeSA */}
5. \( I, O_i = \text{TeSAOf}(N); \)
6. \( \text{I}_{\text{updated}}, \text{O}_{\text{updated}} = \text{PropOneNode}(N, I, O_i); \)
7. foreach \( T \in (\text{I}_{\text{updated}} \cup \text{O}_{\text{updated}}) \) do
8. \( B = \text{NeighborNodesOf}(T); \)
9. \( S.\text{Insert}(B.\text{RemoveOf}(N)); \)
10. return G;

![Initial sparsity attributes](image1) ![Resulting sparsity attributes](image2) ![Propagation direction](image3)

Figure 4: The propagation of sparsity attribute. The gray blocks are propagated sparsity attributes.

to derive the TeSA of other tensors, shown in Algorithm 1. Given a node, if the TeSA of any input or output tensor of a node is updated (TeSAOf in Line 5), PropOneNode (Line 6) updates the TeSA of other tensors associated with this node, according to a certain propagation rule (as discussed later, the propagation could be bidirectional). The propagation repeats until no TeSA requires further update.

Note that if being propagated multiple times, a sparsity attribute will be updated to increase the sparsity until convergence. Multiple pruning updates lead to the union of the pruned elements in all the updates (The tensor \( W_3 \) in Figure 4(c) is an example). For quantization, the attribute will be converged to the fewest quantization bits (or 0-bit, \textit{i.e.}, being pruned). As each propagation monotonically increases sparsity and both the propagations of pruning and quantization are commutative and associative, Algorithm 1 is guaranteed to terminate.

**Intra-operator propagation.** The propagation behavior of PropOneNode varies across different type of operators and attributes. In Figure 4(b), the pruned element [0,0] in tensor \( W_3 \) cannot propagate to \( W_1 \) and \( W_2 \) through the operator Matmul, while it does propagate to upstream tensor if the operator is element-wise computation like ReLU. Propagation could be bidirectional. Figure 4(a) shows that the input \( W_2 \) can affect the output \( W_3 \) and another input \( W_1 \). And in Figure 4(b), \( W_2 \) becomes sparse due to the TeSA of the output \( W_3 \).

The sparsity attribute of the quantization type propagates differently. If an output tensor has a low precision (\textit{e.g.}, 4bit) while the input tensor’s precision is high (\textit{e.g.}, 16bit), the input may use fewer quantization bits with little impacts on output (\textit{i.e.}, information bottleneck [73]).

Next, we show two propagation rules used by SparTA for pruning and quantization attribute. Note that it is possible to extend PropOneNode to support more rules as shown in line 27-line 36 of Algorithm 2. New propagation rules can be registered and invoked in PropOneNode.

**Pruning rule.** The propagation of pruning attributes depends on the computation logic of an operator (\textit{e.g.}, \( +, x \) in Matmul). To capture such property, SparTA defines a TeSA algebra that maps the operator’s element-wise computation to a set with two elements, \{\textit{pruned}, \textit{non-pruned}\}. The TeSA algebra is shown in Table 2. Given an input TeSA, its output TeSA can be computed using the TeSA algebra, following the same computation flow of the operator. Note that Table 2 can be extended to support new operators.

SparTA also proposes Tensor Scrambling, a probabilistic propagation rule that handles black-box or complex operators, where the detailed computation logic is unavailable or unclear. This rule derives the pruned elements of a tensor by scrambling the values of other related tensors. Specifically, the rule sets the pruned elements in the input tensor to zeros, and assigns random values to the remaining elements (\textit{i.e.,} scrambling). It then runs the operator to obtain its output tensor (assuming at least the dense version of the operator is available). By repeating this process enough times (see §5.2), the rule treats those elements that always stay zero as pruned elements in an output tensor.

In addition, the sparsity also propagates from the output to the input, or from one input tensor to another. To achieve this, SparTA leverages the auto differentiation (AD) of DNN computation. An operator’s backward operator is also available for the back-propagation in the AD. Let \( I_{1...n} \) and \( O_{1...n} \) denote an operator’s inputs and outputs respectively. Its backward operator’s inputs are \( I_{1...n} \) and \( gO_{1...n} \) with its outputs being \( gI_{1...n} \), where the prefix \( g \) denotes the gradient of the corresponding tensor. According to AD’s property, \( gI_i \) and \( gO_i \) should have the same TeSA of \( I_i \) and \( O_i \) (both shape and value). To infer the TeSA propagated from tensor \( I_i \) (or \( O_i \)) to \( I_j \), SparTA applies TeSA algebra or Tensor Scrambling to the backward operator: using the TeSAs of \( I_{1...n} \) and \( gO_{1...n} \) as the input, SparTA applies either rule to compute (PropOneNode).

<table>
<thead>
<tr>
<th>Type</th>
<th>Computation</th>
<th>Computation in TeSA Algebra</th>
</tr>
</thead>
<tbody>
<tr>
<td>Unary</td>
<td>( f(x) \Rightarrow y )</td>
<td>( x \rightarrow y )</td>
</tr>
<tr>
<td>Binary</td>
<td>( f(x,y) \Rightarrow z )</td>
<td>( (x = \phi) \land (y = \phi) \rightarrow (z = \phi) )</td>
</tr>
</tbody>
</table>

Table 2: TeSA algebra on a set of attribute values. \( \phi \) and \( \alpha \) represent pruned and non-pruned element respectively.
Algorithm 2: TeSA attribute propagation rules.

```
Data: N: a node in DFG, I_j: node N’s inputs, O_j: node N’s outputs.
Result: Updated input/output TeSAs after propagation on N.
11 Function PruningPropRule(N, I_j, O_j):
12   S_updated = \emptyset;
13   foreach T \in (I_j \cup O_j) do
14       T_updated = TensorScrambling(N, (I_j \cup O_j) \setminus T);
15       if T_updated \neq T then
16           S_updated = S_updated \cup \{T_updated\};
17       return SplitTosOs(S_updated);
18 Function QuantizationPropRule(N, I_j, O_j):
19   S_updated = \emptyset, S = (I_j \cup O_j); D_calib = GetCalibrationDataOf(N);
20   foreach T \in (I_j \cup O_j) do
21       T_updated = LowerBitAndFinetune(N, S, T, D_calib);
22       if T_updated \neq T then
23           S_updated = S_updated \cup \{T_updated\};
24       S_updated = S_updated \cup \{T_updated\};
25       return SplitTosOs(S_updated);
27 RegisterPropRule(PruningPropRule);
28 RegisterPropRule(QuantizationPropRule);
29 Function PruneOneNode(N, I_j, O_j):
30   I_updated = O_updated = \emptyset;
31   foreach RegisteredPropRule do
32       I_proposed, O_proposed = RegisteredPropRule(N, I_j, O_j);
33       I_update(I_proposed), O_update(O_proposed);
34       I_updated = I_updated \cup I_proposed;
35       O_updated = O_updated \cup O_proposed;
36   return I_updated, O_updated;
```

Figure 5: Two-pass compilation to generate an efficient kernel for an operator (MatMul).

distilled into a lower precision with acceptable information loss. Since the information loss can be measured through the operator’s input and output tensors, we first perform inference on the corresponding DNN model using train/test dataset, and collect the resulting input and output tensors of that operator to construct calibration data (line 20). Next, we gradually reduce the quantization precision (e.g., 32-bit to 16-bit) of one tensor of this operator while keeping other tensors unchanged. The operator is then quantized and fine-tuned using the calibration data under the new precision. The fine-tuning is to minimize Mean Squared Error (MSE) between the output tensors in calibration data and the output tensors of that operator after lowering the precision. If the drop of model accuracy is smaller than a predefined threshold (e.g., 1% in our experiment), the new quantization attribute of that operator is accepted (line 22). The process repeats for other tensors in the operator, until all tensors are evaluated. To reduce the cost of collecting calibration data, SparTA works through all the operators in a DNN model in a topological order and caches the activations computed in earlier quantization propagations. Collecting an operator’s calibration data only needs partial inference from the nearest cached activations to this operator. For example, consider a sequential model with two layers \( L_a \) and \( L_b \). The propagation on \( L_a \) has collected its output activations in its calibration data. When working on \( L_b \), its calibration data can be collected by doing partial inference from the collected output activations of \( L_a \). Our evaluation results in §5.2 show the effectiveness of this propagation rule.

### 3.3 Code Generation with TeSA

After attribute propagation, tensors in the DNN model may show a mixture of different sparsity patterns [42, 57, 74, 84]. Such complex patterns make it difficult to generate efficient kernel code. SparTA therefore transforms a tensor with a complex sparsity pattern to multiple tensors, each with a simpler sparsity pattern. Correspondingly, SparTA rewrites the execution plan of the associated operator to accommodate...
new operators that compute the transformed tensors. Finally, SparTA performs code generation for the transformed execution plan, with sparsity-aware specialization.

Figure 5 shows an example of such a two-pass compilation process for Matmul. The weight tensor \( W \) is a tensor with a mixed precision, where two structured blocks use 8-bit quantization and one fine-grained element uses 32-bit. SparTA transforms \( W \) into \( W_1 \) and \( W_2 \), each using a simpler quantization scheme. Consequently, two operators are introduced to compute \( W_1 \times I \) and \( W_2 \times I \), respectively, using the hardware instructions fit for the specific sparsity attribute. As a result, the original execution plan with one operator is transformed into a new plan that requires more tensor operations.

**Execution-plan transformation.** This compilation pass transforms a tensor with a complex sparsity pattern to “regular” (simple) sparsity patterns, which facilitates further optimizations in later passes. In SparTA, a regular sparsity pattern means the TeSA of a tensor shows only one type of quantization attribute and one block size of pruning attribute.

The detailed execution plan transformation for a DNN model is performed operator-by-operator, shown in Algorithm 3. For simplicity, we assume the operator has \( m \) inputs and a single output. The process starts from the operator’s input and output TeSAs, each of which could be transformed to one or multiple TeSAs using TransformTeSA. Correspondingly, the operator is transformed to \( |T_o| \prod_{i=1}^{m} |T_i| \) sub-operators, which are the Cartesian product of those decomposed TeSAs. The sub-operator usually has the same computation logic as the original operator (e.g., the operators that can be expressed with Einstein summation [7]), an approach that has also been taken in the context of DNN model partitioning [82]. The system performs code generation for each sub-operator (line 45), profiles the resulting kernel in the real hardware, and records the proflined result (line 47).

Note that the transformation is a repetitive search process. Given a TeSA, TransformTeSA may have multiple transformation plans. The process iterates over each plan to find the satisfied one (line 38). Figure 6 shows an example. On the left, a mixed precision TeSA can be decomposed to multiple TeSAs each of which has one precision. It can also be transformed to one TeSA where all the elements are aligned to the highest precision. Similarly, for the right example, the sparse TeSA can be decomposed to two TeSAs, one with a block size of 2x2 and the other with a block size of 1x1. It can also be transformed to one TeSA of block size 2x2 with the pruned elements set to zero, or transformed to one TeSA of block size 1x1. Note that the algorithm may decide not to decompose a tensor and choose a block size of 1x1, indicating that the TeSA has a fine-grained sparsity that is hard to be transformed to regular sparsity.

Algorithm 3: Transform an operator’s execution plan.

```plaintext
Algorithm 3: Transform an operator’s execution plan.

Data: \( N \): An \( n \) inputs single output operator to be transformed; \( P \): A transformation policy.

Function TransformOp(\( N \)):
/* HasBudget determines the number of transformation options that can be searched */
while \( P.HasBudget() \) and \( P.PerfNotSatisfied() \) do
/* loop body is one transformation option */
\( S = [] \);
\( T_i \), \( O_i \) = InputsOutputsTeSA(\( N \));
foreach \( i \in 1, \ldots, m \) do
\( T_i = P.TransformTeSA(\( i \));
\( T_0 = P.TransformTeSA(\( 0 \));
foreach \( (t_1, \ldots, t_n, t_0) \in T_1 \times \cdots \times T_m \times T_0 \) do
\( N_{sub} = SpecializeOp(op=N, in=t_1, \ldots, t_m, out=t_0);\)
\( S.Append(N_{sub});\)
end foreach
end for
end while
return \( P.BestTransformation();\)

Function P::TransformTeSA(\( T \)):
/* return a new transformation option per run */
\( T_{transformed} = [] \);
/* BitOption: (i) 4 and 8, (ii) only 8, if hardware natively supports 4-bit and 8-bit */
\( T_1, \ldots, T_k = P.BitRounding(T, self, SampleBitOption());\)
foreach \( i \in 1, \ldots, k \) do
\( T_{transformed} = T_{transformed}.Extend(T_1, \ldots, T_k);\)
end foreach
return \( T_{transformed};\)

Function P::WeightedBlockCover(\( T \)):
\( B_{chosen} = [] \);
/* covering non-pruned elements to blocks using */
every available block size, to produce \( B \)
\( B = P.AllCoveredBlocks(T, self, AvailableBlockSizes());\)
while not AllElementsCovered(\( T, B_{chosen} \)) do
\( B_{covered} = self.UpdateBlockCost(B);\)
\( B = BlockWithMinCost(B_{covered});\)
\( B_{chosen}.Append(b);\)
end while
/* decompose \( T \) to the TeSAs with different block sizes based on \( B_{chosen} */
return DecomposedTeSAs(T, B_{chosen});
```

Figure 6: Multiple transformation plans produced by a transformation policy. The specialization hints are used by the second pass compilation for code specialization.
to 8-bit. For each TeSA returned by \texttt{BitRounding}, function \texttt{WeightedBlockCover} chooses one or multiple proper block sizes to cover the non-pruned elements, which we treat as a \textit{weighted set cover} problem [19]. The weight of each block size corresponds to the cost of computation with the block size on the underlying hardware (see §4 for details). We use a simple greedy algorithm to pick the blocks with the lowest cost (i.e., \( \text{num\_covered\_elements} \)), until all non-pruned elements are covered (line 56).

Transformation policy \( P \) can be further customized to incorporate new optimizations (e.g., supporting Sparse Tensor Cores [1] detailed in §5.3).

To help codegen, each transformed TeSA is attached with the information about the bit width and block size, named as \textit{specialization hints} (illustrated in Figure 6). The hints will be passed to the second pass, elaborated next.

\textbf{TeSA code specialization}. The second compilation pass specializes kernel code for each (sub)operator (i.e., line 45 in Algorithm 3). The specialization hints generated from the previous pass guide the specialization strategy. For example, the bit-width of an operator suggests whether to leverage a specific hardware instruction (e.g., \texttt{DP4A}). And the loop tiling of the operator should be aligned with the block size for effective dead code elimination (DCE). In addition, SparTA can leverage traditional DNN compilers for dense computation. For example, some intra-block computation is dense and thus can use a dense implementation generated by existing DNN compilers [29,60,91].

The specialization process starts from a dense version of the operator, implemented as multi-level loops generated by a traditional DNN compiler [29]. It first specializes under the guidance of the block size in the specialization hints. It searches from the outermost loop until the level (say \( l \)) of inner loop body aligns with the block size. Since the pruning sparsity attribute is specified at the granularity of block size, many runs of the loop body within level \( l \) are dead computation. To eliminate the dead computation, we introduce a new schedule primitive \texttt{dismantle} that jointly performs loop unrolling and DCE. When \texttt{dismantle} is applied on a loop, this loop and all its outer loops are unrolled and specialized with the given sparsity attribute. An example is shown in Figure 7(b). \texttt{dismantle} is applied on the third loop, thus the top three loops are unrolled, generating eight small Matmuls (i.e., \([2,2] \times [2,2]\)). According to the sparsity pattern in Figure 7(a), six of them are dead computation and can be eliminated. In essence, \texttt{dismantle} embeds a specific sparsity pattern into the code, which eliminates the need of sparsity encoding, e.g., compressed sparse row (CSR) [26]. As the index to the non-pruned blocks/elements is specialized in the code, the overhead of indirect addressing on the index is removed.

Given a different transformation plan (and the specification hints), the code can be specialized differently. The hint in Figure 7(c) show a smaller block size. In this case, the loop body is a smaller Matmul (i.e., \([2,1] \times [1,1]\)), which enables more DCEs. Besides the dead computations eliminated in the previous case, four computations of the small Matmul can be removed. Furthermore, as the small Matmul only accesses one value in \( W \), the value can be directly specialized to the code (i.e., \( W_{x,y} \)) without maintaining a sparse tensor \( W \) in memory.

A specialization hint could also specify the block size equal to the tensor’s shape (i.e., one block covers the whole tensor). In this case, SparTA can directly use the existing state-of-the-art general sparse kernel implementation (e.g., \texttt{cuSPARSE} [3], \texttt{taco} [53]) or even use the dense kernel implementation if it perform better. SparTA’s specialization framework is general to incorporate any sparsity-aware techniques, including the off-the-shelf sparse kernel and even its dense version.

Specializing operators with quantization attributes also works on the multi-level loops, but starting from the innermost loop. SparTA picks a proper hardware instruction based on the bit-width denoted in TeSA, e.g., \texttt{DP4A} [13] or \texttt{wmma} [5] for 8bit, \texttt{wmma} for 4bit. The specialized tiling of the innermost loop(s) is then aligned to the computation shape of the instruction. For example, one supported computation shape of \texttt{wmma} is the Matmul \([16,16] \times [16,16]\). To specialize for this instruction, the tiling should rearrange the innermost loop body to align with the shape, and then replace the rearranged loop body with the instruction. The tiling for the instruction \texttt{DP4A} with a shape \([1,4] \times [4,1]\) can be done similarly.
4 Implementation

We implemented SparTA on Rammer [60], a state-of-the-art open-source DNN compiler with no special support for sparsity. We implemented sparsity attribute propagation as a dedicated compilation pass over Rammer. A DNN model is converted to an ONNX graph [14] before compiling. Each TeSA element has a two-byte attribute: 7 bits record the bit width of the element, 4 bits specify the element’s data format (e.g., unsigned int, float32, bfloat16), and the rest 5 bits are reserved. The bit-width zero means the element is pruned. TeSA exists only during compile time and therefore incurs no runtime memory cost. We implemented the execution plan transformation within Rammer, with an additional compilation pass that rewrites the graph with a better execution plan. The specialized sub-operators are injected into Rammer’s kernel DB for constructing the whole executable.

For the efficient execution of weighted block cover in execution plan transformation (§3.3), SparTA calculates the weight of different block sizes. Specifically, we implemented a kernel template for block sparsity, and evaluated 13 different block sizes on that template. The overhead of each block size is profiled by measuring the latency of the kernel with zero sparsity and dividing the latency with the number of blocks. The overhead is used as the weight of the block sizes. When an operator is too sparse to saturate available cores, the weights may become less accurate. In such cases, we enumerate all the combinations of block sizes to run the weighted block cover algorithm and pick the best one. For kernel specialization, we implement the dismantle primitive based on loop unrolling. When a loop is unrolled with dismantle, we read the corresponding TeSA and eliminate dead computations accordingly.

SparTA, as a full-stack solution for model sparsity, has supported 21 model sparsity algorithms, including 16 pruning algorithms and 5 quantization algorithms (full list omitted due to page limit). Those algorithms can run on SparTA with little code modifications, and benefit from SparTA not only on sparsity exploration but also on model fine tuning, which will be demonstrated in §5.4.

5 Evaluation

We evaluate SparTA on three popular DNN models with four different sparsity patterns across different task domains, shown in Table 3. We evaluate four representative sparsity patterns, covering different pruning and quantization schemes and their combination. Unstructured sparsity prunes model weights in the granularity of an element in weight tensors to reach the desired sparsity ratio [43, 54, 55]. Structured sparsity prunes weights in the granularity of column, row, channel, or block, depending on specific models [44, 56, 59]. We apply different sparsity patterns to the three selected models to show SparTA’s effectiveness under various patterns. BERT is pruned in row combined with a block of size 32x32 [87]; MobileNet gets pruned in the output channel [94]; for HuBERT, it is a combination of channel pruning in the Conv layer and head pruning in the transformer layer [56, 62].

To further demonstrate the powerful expressiveness of TeSA, we apply structured sparsity, based on which 8bit quantization is further applied on the remaining tensor elements to construct the third sparsity pattern, i.e., Structured+8bit. Finally, we introduce an even more complicated Mixed Sparsity for BERT. On top of the Structured+8bit sparsity, we apply unstructured sparsity with 32bit quantization back to 0.01% of the pruned elements [48, 84]. This leads to a total sparsity ratio of 94.99%.

We trained the models (BERT on dataset QQP [51], MobileNet on ImageNet-Dogs [33], HuBERT on SUPERB [85]) with the above sparsity patterns, the accuracy change

<table>
<thead>
<tr>
<th>Model</th>
<th>Type</th>
<th>Sparsity</th>
<th>Ratio</th>
<th>Acc (%)</th>
</tr>
</thead>
<tbody>
<tr>
<td>BERT</td>
<td>NLP</td>
<td>Structured [87]</td>
<td>93%</td>
<td>89.7-&gt;88.49</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Unstructured [43]</td>
<td>95%</td>
<td>89.7-&gt;88.67</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Structured+8bit [52]</td>
<td>95%</td>
<td>89.7-&gt;88.03</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Mixed Sparsity [48, 84]</td>
<td>94.99%</td>
<td>89.7-&gt;88.63</td>
</tr>
<tr>
<td>MobileNet</td>
<td>CV</td>
<td>Structured [94]</td>
<td>60%</td>
<td>78.27-&gt;75.62</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Unstructured [43]</td>
<td>95%</td>
<td>78.27-&gt;74.15</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Structured+8bit [52]</td>
<td>60%</td>
<td>78.27-&gt;75.13</td>
</tr>
<tr>
<td>HuBERT</td>
<td>Speech</td>
<td>Structured [56, 62]</td>
<td>80%</td>
<td>95.61-&gt;95.5</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Unstructured [43]</td>
<td>95%</td>
<td>95.61-&gt;95.55</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Structured+8bit [52]</td>
<td>80%</td>
<td>95.61-&gt;95.43</td>
</tr>
</tbody>
</table>

Table 3: Evaluated DNN models with different sparsity patterns and their resulting accuracy. The second column lists the major operators of each model. The column Ratio denotes the initial sparsity ratio for pruned weights.

- Sparsity attribute propagation increases the end-to-end sparsity ratio by up to 39.7%. With execution plan transformation and code specialization, SparTA can achieve up to 6.7x speed up over the state-of-the-art sparse kernel implementation for a sparse DNN operator (e.g., Matmul) with complex sparsity patterns. (§5.2, §5.3)

- SparTA facilitates the development and exploration of sparse DNN models, producing DNN models with lower inference latency and/or higher accuracy. (§5.4)

5.1 End-to-End Experiments

We evaluate SparTA on the inference latency and memory usage of three popular DNN models across different task domains, shown in Table 3. We evaluate four representative sparsity patterns, covering different pruning and quantization schemes and their combination. Unstructured sparsity prunes model weights in the granularity of an element in weight tensors to reach the desired sparsity ratio [43, 54, 55]. Structured sparsity prunes weights in the granularity of column, row, channel, or block, depending on specific models [44, 56, 59]. We apply different sparsity patterns to the three selected models to show SparTA’s effectiveness under various patterns. BERT is pruned in row combined with a block of size 32x32 [87]; MobileNet gets pruned in the output channel [94]; for HuBERT, it is a combination of channel pruning in the Conv layer and head pruning in the transformer layer [56, 62].

To further demonstrate the powerful expressiveness of TeSA, we apply structured sparsity, based on which 8bit quantization is further applied on the remaining tensor elements to construct the third sparsity pattern, i.e., Structured+8bit. Finally, we introduce an even more complicated Mixed Sparsity for BERT. On top of the Structured+8bit sparsity, we apply unstructured sparsity with 32bit quantization back to 0.01% of the pruned elements [48, 84]. This leads to a total sparsity ratio of 94.99%.

We trained the models (BERT on dataset QQP [51], MobileNet on ImageNet-Dogs [33], HuBERT on SUPERB [85]) with the above sparsity patterns, the accuracy change
of those sparse models is shown in the Acc column of Table 3. Overall, the accuracy drops are consistent to those reported in the corresponding papers. The accuracy of sparse BERT drops around 1%. Mixed sparsity has the accuracy of 88.63% at 94.99% sparsity, nearly the same accuracy as unstructured sparsity, but is much easier to be accelerated. For MobileNet, unstructured sparsity has a higher accuracy drop, as its sparsity is high (i.e., 95%). For HuBERT, 95% unstructured sparsity can outperform the structured ones with 80% sparsity ratio.

The models are evaluated on three types of accelerators: NVIDIA GeForce RTX 2080 Ti, AMD Radeon VII, and Intel Xeon Silver 4210 CPU. We compare SparTA with seven representative solutions, including one popular deep learning framework: PyTorch (v1.7) with JIT [67], two vendor-specific toolkits: TensorRT (v7.2) for NVIDIA GPUs [18] and OpenVINO (v2021.4.1) for Intel CPUs [15], two DNN compilers: TVM (0.9.dev0) [29] and Rammer [60] (which offers the state-of-the-art performance). To evaluate the state-of-the-art sparse kernels/libraries in an end-to-end model, we create Rammer sparse (or Rammer-S) by wrapping in Rammer these sparse kernel libraries/implementations, including cuSPARSE [3], taco [53], and Sputnik [39] for NVIDIA GPU, hipSPARSE [17] for AMD GPU, MKL Sparse Linear Algebra [12] for Intel CPU. For TVM, we also evaluate its sparsity support [22] (denoted by TVM-S). Each model on TVM is tuned with 1,000 trials per task using Ansor [91], aligned with the common practice [91]. The batch size we used in the end-to-end experiments (except Figure 12) is 32.

5.1.1 SparTA on CUDA GPUs

Structured sparsity. The first row of Figure 8 shows the inference latency of the three models on the structured sparsity. PyTorch, TensorRT, TVM, and Rammer treat them as three dense models. TensorRT performs the best among them. Compared to TensorRT, SparTA is 3.7x, 2.9x, and 2.4x faster on BERT, MobileNet, and HuBERT, respectively. TVM-S and Rammer-S are aware of sparsity. TVM-S incurs high inference latency, as the kernel templates it uses cannot efficiently support different sparsity patterns. Rammer-S performs marginally better than TensorRT on MobileNet and HuBERT. The SOTA sparse kernel uses Sputnik, which performs better than cuSPARSE and taco on those models. SparTA performs 1.7x, 2.6x, and 2.3x faster than Rammer-S. Its performance gain comes mainly from sparsity propagation, which increases the whole model’s sparsity (see §5.2) and sparsity transformation, i.e., covered with different block sizes on different layers (see §5.3).

Memory footprints in the inference are shown in the first row of Figure 9. SparTA shows the smallest footprint. For MobileNet, PyTorch and TensorRT consume much more memory, because they use cuDNN, which requires additional memory to store weights and activations. SparTA’s memory usage is smaller than TVM-S and Rammer-S due to sparsity propagation, which increases the sparsity ratio.

Unstructured sparsity. For unstructured sparsity (i.e., second row of Figure 8), TensorRT also performs the best among those dense baselines, marginally better than Rammer. SparTA is 1.6x, 2.2x, and 1.5x faster than TensorRT on BERT, MobileNet, HuBERT, respectively. Rammer-S still uses Sputnik. SparTA outperforms Rammer-S by 1.13x, 2.4x, and 1.3x on BERT, MobileNet, HuBERT, respectively. The speedup on MobileNet is high because the sparsity is easier to be propagated on depthwise and pointwise convolution even with unstructured sparsity. On BERT and HuBERT, the performance gain over Rammer-S mainly comes from code specialization (i.e., weight values are embedded into kernel code). For the memory usage (i.e., the second row of Figure 9), SparTA shows a usage similar to TVM-S and Rammer-S, and performs better than the other baselines.

Structured+8bit. Shown in the third row of Figure 8, Ten-
Mixed sparsity. The left figure in Figure 10 shows the latency of BERT with Mixed Sparsity. SparTA is 5.9x, 5.0x, 6.8x, 8.7x, 5.2x, 2.2x faster than PyTorch, TensorRT, TVM, TVM-S, Rammer, Rammer-S, respectively. Unlike structured+8bit, TensorRT shows slight advantage over other baselines on mixed sparsity, although most elements are 8-bit.

Latency Breakdown. Figure 11 shows the performance breakdown of BERT on the four sparsity patterns. “+Sparse Kernel” applies our generated sparse kernels following the original sparsity ratio without operator transformation and kernel specialization. It can be treated as Rammer-S. “+Propagation” applies sparsity propagation on the model and regenerates the sparse kernels without transformation and specialization. “+Transformation” tunes the block size for covering non-pruned elements of each sparse operator, and for mixed sparsity it also decomposes sparse tensors to multiple ones. “+Specialization” tunes intra-block implementation and embeds values into codes when necessary.

For mixed sparsity, the latency reduction brought by each optimization is 55.8%, 19.7%, 37.7%, and 12.6%, respectively. The other three sparsity patterns could be viewed as a type of breakdown of mixed sparsity. In structured sparsity and structured+8bit, transformation brings 20.5% and 26.5% latency reduction, respectively, while propagation brings 19.7% and 15.8% latency reduction, respectively. Finally, intra-block specialization brings 8.2%, 11.4%, and 13.7% latency reduction for structured, unstructured, and structured+8bit, respectively. The significance of a certain optimization depends on DNN models and sparsity patterns. For BERT in Figure 11, “+Sparse Kernel” brings 2.1x gain on average, SparTA brings an extra 2x gain. For MobileNet to be illustrated in §5.2, “+Propagation” brings the most gain (e.g., increasing sparsity from the 50% to 89.7%).

Latency of different batch sizes. Figure 12 shows the performance of BERT under different batch sizes on NVIDIA 2080 Ti. When batch size varies from 8 to 64, SparTA is on average 4.1x-4.6x, 2.4x-2.7x, 4.2x-6.3x, 5.9x-13.8x, 3.6x-4.1x, 2.2x-2.6x faster than PyTorch, TensorRT, TVM, TVM-S, Rammer, Rammer-S, respectively on three sparsity patterns. The overall speedup of SparTA is similar across different batch sizes. The range of the speedup over TVM-S is relatively large, because large batch size induces larger tuning space that makes the kernel tuning in TVM less effective.

Compiling overhead. The overhead of SparTA comes from the compiling phase, which consists of three parts: propagation, transformation, specialization. The overhead is positively related to the number of operators in the model. Taking BERT as an example, the propagation, transformation, and specialization take 3 minutes, 2 hours and 1.5 hours respectively using a single thread. It is possible to reduce the overhead by leveraging more prior knowledge in transformation policy and pre-tuned kernels. We leave it as future work.

5.1.2 SparTA on Other Accelerators

ROCm GPU. Figure 13 shows inference latency of the three models on AMD Radeon VII. The speedup of SparTA over PyTorch on the three sparsity patterns is up to 3.5x, 4.2x, 2.2x for BERT, MobileNet, and HuBERT, respectively. The kernel tuning of TVM on ROCm GPUs does not function properly (always stuck in Debug mode), the performance of TVM and TVM-S on BERT and HuBERT suffers a lot. They show reasonable performance on MobileNet, because they...
Although Rammer-S with hipSPARSE has higher latency than TVM, TVM-S, the speedup of SparTA is 1.8x, 1.6x, 1.5x, compared to OpenVINO, a high-performance inference engine on CPU, the result is shown in the right of Figure 10. Compared to PyTorch, TVM, TVM-S, Rammer, and Rammer-S, respectively. Although Rammer-S with hipSPARSE has higher latency than Rammer, it has a lower memory footprint.

**Intel CPU.** We evaluated mixed sparsity pattern of BERT on CPU, the result is shown in the right of Figure 10. Compared to OpenVINO, a high-performance inference engine for Intel CPUs, SparTA achieves 1.7x speedup. For PyTorch, TVM, TVM-S, the speedup of SparTA is 1.8x, 1.6x, 1.5x, respectively. Rammer-S uses the MKL library, which leverages the sparsity, thus faster than other baselines. SparTA still has 1.4x performance gain over Rammer-S, because it leverages low-bit instruction (i.e., AVX512 VNNI [10]) and further optimizes the model with sparsity propagation and execution plan transformation in a holistic way.

### 5.2 Sparsity Attribute Propagation

**Propagation of pruned elements.** The performance gain brought by propagation on BERT has been illustrated in Figure 11. The propagation has higher potentials on MobileNet, as convolution’s filter size is small (e.g., 3x3). Figure 14 shows how sparsity is propagated across layers on MobileNet, which increases each layer’s sparsity ratio. In this experiment, we tested three sparsity ratios (i.e., 50%, 70%, 90%) pruned by the same algorithm used in the end-to-end experiment. For each sparsity ratio, we prune every layer of MobileNet to the target ratio. Then the propagation rule is applied. The accuracy results of inference on train/test dataset are exactly the same before and after propagation, as the propagation rule for pruned elements does not affect computation logic.

For structured sparsity, the total sparsity ratio is increased from 50% to 89.7% after propagation. The curve’s zigzag is caused by different propagation potential of the interleaving depthwise convolution and pointwise convolution in MobileNet. Interestingly, when the original sparsity ratio is 90%, after propagation the sparsity ratio becomes 100%, which explains the anomaly that, although there are 10% filters left on each convolution (before propagation), the model’s accuracy is similar to a random image classifier. The propagation ability on unstructured sparsity is lower. Only high sparsity ratio could bring an obvious increase of sparsity ratio. For example, with 90% original sparsity, the total sparsity is increased to 95.3% after propagation. With Tensor Scrambling, our experiences show 256 randomly sampled tensors can identify sparsity correctly.

**Propagation of quantization bit.** In this experiment, we evaluate the propagation rule for quantization described in §3.2. We follow the same approach proposed in HAQ [77] to quantize MobileNet. Specifically, it uses reinforcement learn-
5.3 Efficient Code Generation with TeSA

Effectiveness of execution plan transformation. The sparsity-aware execution plan transformation in SparTA could handle complex sparsity patterns efficiently. We test two sophisticated sparsity patterns: (1) Mix of structured sparsity with block size 32x32 and unstructured sparsity (i.e., 1x1) [48, 53]. There are 1% unstructured elements, and the structured sparsity ratio varies from 70% to 90%. (2) Based on the first sparsity pattern, we further make the structured sparsity 8bit, and make unstructured sparsity 32bit [84]. To show the effectiveness of transformation, we compare SparTA with two baselines: one is our specialized kernel for structured sparsity (denoted by BlockSparse), where the unstructured elements are covered with 32x32 blocks; the other is Sputnik, which is optimized for unstructured sparsity.

The results are shown in Figure 16. For the first sparsity pat-
block size to cover those non-pruned elements. We picked 5 sparse tensors with different sparsity patterns in BERT, apply WeightedBlockCover to find the best block size of the 5 tensors. Table 4 shows the found block sizes. The chosen block sizes are all different from the original 32x32 block size and they all perform much better than the kernel implemented with the original block size. Essentially, the block covering makes a trade-off between the efficiency that a certain block size is optimized for the underlying hardware and the ratio of the computation wasted using that block size.

**Effectiveness of TeSA code specialization.** We evaluate SparTA’s specialized matrix multiplication kernel under different unstructured sparsity ratios, ranging from 50% to 99%. We compare the specialized kernels with cuSPARSE, taco [16, 53], and Sputnik [39]. The result is shown in Figure 18. At 99% sparsity, cuSPARSE outperforms cuBLAS, but incurs 2.2x slowdown at 95% sparsity. In most cases, cuSPARSE performs much worse than cuBLAS on latency, although it has a lower memory footprint due to encoded sparse tensors. taco performs worse than cuSPARSE due to its inefficient utilization of shared memory [70]. It is 15.6x slower than cuSPARSE for 99% sparsity; the slowdown is reduced to 4.0x when the sparsity is 50%. SparTA is up to 6.0x faster than cuSPARSE. It outperforms cuBLAS when the sparsity is only 70%. Sputnik also performs better than cuSPARSE and taco. SparTA is up to 1.7x faster than Sputnik.

Figure 18: Comparison of cuSPARSE, taco, Sputnik, and SparTA on matrix multiplication (1024x1024x1024) with fine-grained sparsity under different sparsity ratios. B is sparse for $A \times B$.

### 5.4 Augmented Model Sparsity Exploration

SparTA, as a full-stack solution for model sparsity, facilitates the exploration of existing model sparsity algorithms. In this section, we demonstrate this from the following two aspects. **Actual latency vs. FLOPS as proxy-metric for latency reduction in model pruning.** In this experiment, we use Simulated Annealing [58] to prune MobileNet to reduce 30% and 40% inference latency, respectively, i.e., the two dash lines in Figure 19. Our baseline uses FLOPS as the metric to filter out the disqualified models: the model whose FLOPS is larger than 70% of the original FLOPS. In contrast, SparTA uses the real latency to filter models. The result is shown in Figure 19. The best sparse models found by the two approaches have similar accuracy. However, the model found via FLOPS does not meet the latency target, 23.8% and 51.4% higher than the target, respectively. This shows FLOPS cannot faithfully reflect real inference latency. In contrast, the sparse models found by the algorithm on SparTA successfully satisfy the latency requirement.

**Speeding up sparsity exploration.** With high-performance sparse kernels, SparTA can speed up the exploration process of a sparsity algorithm, which usually searches for a sparsity pattern iteratively [58, 86]. In each iteration, the algorithm “sparsifies” a proportion of the model (e.g., 30%) and fine-tunes it. It repeats the iteration until achieving the targeted sparsity (e.g., 90%). In this process, model fine-tuning consumes significant exploration time. With SparTA, model fine-tuning can be accelerated. Figure 20 runs Simulated Annealing, an iterative sparsity algorithm, on ResNet50. The algorithm prunes 50% of the remaining weights and fine-tune 300 epochs in each iteration. SparTA reduces 31.8% of the total exploration time, compared to the baseline that always uses the original dense model.

### 5.5 Accelerating Sparse Model Training

In addition to model pruning and quantization, some DNN models are designed to be sparse from the beginning, e.g., sparse attention [72]. SparTA can also be used to speed up the training process of such sparse models.

We show this by applying SparTA to the training of NÜWA [81], a state-of-the-art visual synthesis pretrained model that adopts a novel 3D Nearby Attention (3DNA) mechanism. In 3DNA, each token computes the attention to the nearby tokens within a small 3D window, instead of to all the tokens (i.e., full attention).

We implement 3DNA using SparTA and compare the performance with its previous PyTorch implementation (a dense version), and another version implemented using OpenAI’s Triton (v1.1.1) [21], a compiler that supports sparse attention. As the two baselines are PyTorch-based, we integrate SparTA-based 3DNA into PyTorch for a fair comparison. The result is shown in Figure 21. Both Triton and SparTA perform much faster than the default PyTorch version, and consume less GPU memory. The default PyTorch version encounters out-of-memory when the batch size grows beyond 16. SparTA is 2.15~2.24x faster than Triton across different
6 Related Works

Sparsity support in DNN frameworks and compilers. Deep learning frameworks like PyTorch [67] and TensorFlow [23] or compilers like TVM/Ansor [29, 91] exploit sparsity by vendor-specific libraries like cuSPARSE/cuSPARSELt [3] or user-provided sparsity kernel templates [29]. The lack of understanding to the specific sparsity pattern across a sparse model leads to a subpar performance. In contrast, with TeSA, SparTA can capture arbitrary sparsity patterns and enable various sparsity-aware optimizations to generate efficient end-to-end code.

SparTA’s design incorporates several classic compiler techniques. For example, sparsity attribute propagation is similar to type qualifiers [38] and type inference [32]. OpenMP [35] also leverages attribute propagation in a different problem domain with a different mechanism. Code specialization based on value profiling [27] is also a well-known technique. Zeroploit [69] and PGZ [71] also use a similar idea, but focus on gaming applications. Instead of values, SparTA uses more general attributes for code specialization. And SparTA offers a complete framework for DNN model sparsity.

Sparsity acceleration of DNN models. Sparse matrix multiplication has been studied for decades in scientific computing [68, 80]. With the emerging accelerators (e.g., GPU [8, 20], TPU [4], FPGA [11], GraphCore [9]), some research optimizes sparse matrix multiplication for a certain type of hardware [24, 26, 39, 80, 95]. Another type of works study an efficient sparse data format (e.g., CSR, CSB, and DIA) to reduce memory footprint and improve cache efficiency. taco [30, 53, 70] generalizes various sparse data formats with a unified expression. It generates sparse kernel code using the proper data format best fit for a class of sparsity pattern (e.g., 99% sparsity). Unlike taco, SparTA proposes a holistic framework for sparsity, including sparsity propagation, execution plan transformation, and code specialization.

To optimize sparse kernels on GPU, SparseRT [79] embeds sparse weight values into kernel codes rather than stored in a sparse data format. It can be seen as a special case of code specialization in SparTA, i.e., unrolling all the loops. Hong et al [48] reorders elements in a sparse tensor and uses an adaptive tiling strategy to enhance the performance of sparse matrix multiplication. These optimizations are complementary to SparTA.

Some works [28, 88] co-design sparsity algorithms with hardware, which balance sparsity for efficient parallel execution on a GPU. Similar design has been incorporated in Sparse Tensor Core [93]. EIE [41] designs a new data encoding/decoding node and a new Processing Element (PE) to speed up matrix-vector multiplication. SCNN [65] designs another architecture of PE, which supports sparse convolution in a compressed format. SparTA can leverage these new accelerators with new transformations and specialization passes.

Sparsity exploration on DNN models. Research on both neural science and deep learning suggests that a deep neural network is sparse [54, 89]. Various model compression algorithms are shown to construct sparse models with little accuracy degradation. Unstructured pruning prunes model weights without a regular pattern [43, 54, 55], while other works prune DNN models in a regular granularity, such as in the filter [44], channel [56, 59] in CNN, and block level [61, 63]. Quantization is another way to sparsify a model, including single-precision [31, 52, 92], mixed-precision among layers [36, 57, 77], and mixed-precision within each tensor [66, 84]. Recent works further combine the pruning and quantization techniques [42, 74, 75, 78, 83, 90]. SparTA’s TeSA abstraction could capture the sparsity patterns in all these works and generate efficient code for the sparse model.

7 Conclusion

SparTA takes a principled system approach to model sparsity in deep learning, centered on the new TeSA abstraction. SparTA is designed to accommodate a rich set of sparsity patterns, work end-to-end and across the stack to support propagation of sparsity patterns and the optimizations that take advantage of those patterns, and leverage compiler technology and hardware support, all in an extensible framework. SparTA can not only contribute to superior sparsity-induced speedup, but also accelerate model sparsity innovations within a unified framework, for the first time.
References


[86] Takashi Yoshida and Kenichi Ohki. Natural images are reliably represented by sparse and variable populations.


A Artifact Appendix

Abstract

SparTA proposes the new TeSA abstraction which enables the sparsity optimization across the compiler stack. This artifact reproduces the main results of the evaluation on NVIDIA 2080Ti and A100.

Scope

This artifact will validate the following claims:

• End-to-end performance: By reproducing the experiments of Figure 8, 9, 10, 11, we can validate the end-to-end latency and memory footprint of SparTA claimed in §5.1.

• Effectiveness of the propagation: By reproducing the experiments of Figure 14, 15, we can validate the effectiveness of the propagation.

• Effectiveness of the transformation: By reproducing the experiments of Figure 16, 17, we can validate the effectiveness of the transformation.

• Effectiveness of the specialization: By reproducing the experiments of Figure 18, we can validate the effectiveness of the specialization.

• Augmentation of model sparsity exploration: By reproducing the experiments of Figure 20, we can validate that SparTA can augment the model sparsity exploration for the algorithms.

Contents

In this artifact, we will reproduce the Figure 8-11, 14-18, 20 on NVIDIA 2080Ti and A100. Each figure has a shell script to reproduce and visualize the experimental results automatically. In addition, there are many baselines compared in our evaluation, therefore, we also provide a Dockerfile containing all dependent environments for 2080Ti and A100 respectively. Users can quickly set up the experiment environment with the Dockerfile we provided.

Hosting

The artifact is hosted at https://github.com/microsoft/SparTA/tree/sparta_artifact. To get the code, please git clone the SparTA repository and checkout to the sparta_artifact branch.

Requirements

• Hardware requirements: Figure 17 requires a NVIDIA A100 GPU and the other Figures requires a NVIDIA 2080Ti GPU.

• Software requirements: Please use docker to build the image/Dockerfile to set up the environment for 2080Ti and image/Dockerfile.a100 to set up the environment for A100.

• CUDA Driver: Larger than 11.2.

Tutorial

Environment setup

To set up the environment, please first clone the code and build the docker image based the Dockerfile we provided. Second, please start a docker instance and install the SparTA in the python environment. Finally, please run the init_env.sh to initialize the environment variables and download the datasets. Listing 1 shows the commands used to set up the experiment environment.

Listing 1: Commands to set up the environment

1 # get the source code
2 git clone -b sparta_artifact https://github.com/microsoft /SparTA.git
3 cd SparTA/image
4 # build the docker image
5 sudo docker build -t artifact
6 # start a docker instance
7 sudo docker run -it --gpus all --shm-size 16G artifact
8 # Execute following commands in the docker instance
9 # install the sparta
10 mkdir workspace && cd
11 git clone https://github.com/microsoft/SparTA && cd SparTA && git checkout sparta_artifact
12 conda activate artifact
13 python setup.py develop
14 # initialize the environment
15 cd script && bash init_env.sh

Run experiments

SparTA provides the end-to-end scripts to reproduce all the experiments with one command on NVIDIA 2080Ti and A100 respectively. Listing 2 shows the commands to start all the experiments. The reproduced results will be visualized and saved automatically.

Listing 2: Commands to run the experiments

1 # go into the script directory
2 cd script
3 # for 2080Ti
4 bash run_all_2080ti.sh
5 # for A100
6 bash run_all_a100.sh