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

This paper is included in the Proceedings of the 16th USENIX Symposium on Operating Systems Design and Implementation.

July 11–13, 2022 • Carlsbad, CA, USA

978-1-939133-28-1

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

Ningxin Zheng[1*], Bin Lin[1,2*], Quanlu Zhang[1], Lingxiao Ma[1], Yuqing Yang[1], Fan Yang[1], Yang Wang[1], Mao Yang[1], Lidong Zhou[1]

[1]Microsoft Research, [2]Tsinghua 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∼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.

*: Equal contribution.

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).

Sparsity Ratio	50%	90%	95%	99%
cuSPARSE	1652.5	633.9	463.0	181.7
cuBLAS	208.3	208.3	208.3	208.3

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

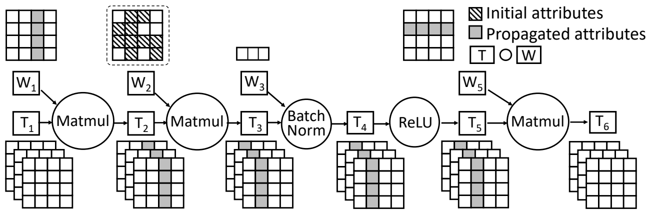


Figure 1: The sparsity attribute of one tensor can be propagated along the deep learning network.

Various forms of sparsity. Deep learning model sparsity is an active and extensively studied research topic. Currently, there are various sparsity patterns being studied. Structured (coarse-grained) sparsity, including channel-granularity sparsity, and block sparsity [56, 59, 61, 63], involves pruning a channel or a sub-block of tensors (e.g., weight or activation tensor) associated with some operators. With unstructured (fine-grained) sparsity, any element of a tensor [43, 54, 55] might be pruned. Quantization algorithms represent models at different levels of precision (*e.g.,* binarized models [31], 8-bit models [52, 92]), and even with different, mixed precision across neural network layers [36, 57, 77] or within a single tensor [66, 84]. Some research further combines pruning and quantization in order to achieve high accuracy under the strict latency and memory constraints [42, 74, 75, 78, 83, 90]. Overall, pruning and quantization have been shown effective in reducing the size and computation complexity of certain deep learning models, sometimes by more than 10 times, without losing much accuracy [42, 76].

The myth of FLOPs. Model sparsity does not translate directly into performance benefits. The existing practice of using "proxy metric" (*e.g.,* FLOPs, or Floating point operations) to evaluate the effect of their proposals such as model inference latency is flawed and leads to inaccurate results. For example, when an operator's weight is pruned by 50% with fine-grained sparsity, even though in theory its FLOPs can be reduced by half, the actual inference latency may become higher with a default sparse kernel (§5.3).

One reason is the sub-optimal implementation of current *generic* sparse kernels. A generic sparse kernel tends to apply a few default sparse encoding schemes (*e.g.,* Compressed Sparse Row [26]) to any sparse tensors. This may miss optimization opportunities in a tensor with a specific sparsity pattern, such as structured sparsity. As a result, a generic sparse kernel library like cuSPARSE [3] can outperform cuBLAS, its dense counterpart [2], only in some extreme sparse case (98%), as shown in Table 1. This motivates the need to find a general framework to implement *specialized* kernels tailored for individual sparsity schemes.

The diminishing end-to-end returns. Sparsity algorithms often focus on exploring the sparsity of a certain DNN operator (*e.g.,* convolution [64]). However, when placed in an end-to-end deep learning model, the sparsity pattern across the whole model can be impacted by each of the operators in the model. This may introduce sophisticated sparsity pat-

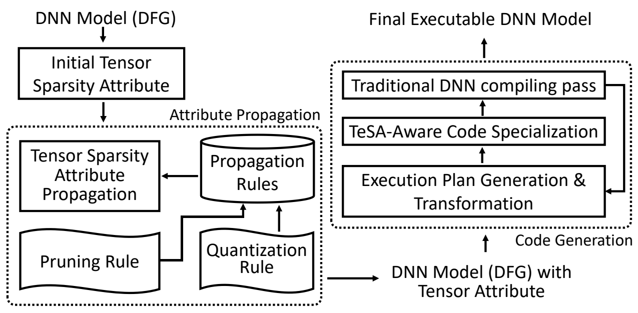

DNN Model (DFG)
Initial Tensor Sparsity Attribute
Attribute Propagation
Tensor Sparsity Attribute Propagation
Propagation Rules
Pruning Rule
Quantization Rule
Final Executable DNN Model
Traditional DNN compiling pass
TeSA-Aware Code Specialization
Execution Plan Generation & Transformation
Code Generation
DNN Model (DFG) with Tensor Attribute

Figure 2: The system architecture of 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 $T_2 \times W_2 = T_3$). 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

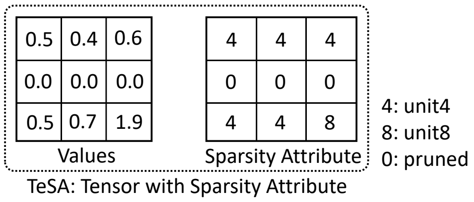


Figure 3: An example of TeSA. Sparsity Attribute denotes the sparsity pattern, including quantization (4 means `uint4`, 8 means `uint8`) and pruning (0 means the element is pruned).

Algorithm 1: TeSA attribute propagation.

```
   Data: G: DFG of TeSA annotated DNN model.
   Result: G with updated TeSAs.
1  Function Propagate(G):
2      S = Set(AllNodesOf(G));
3      while $S \neq \emptyset$ do
4          N = S.PopOne() ;   /* can start from any node */
           /* $I_s$ ($O_s$) is node N's input (output) TeSA  */
5          $I_s$, $O_s$ = TeSAOf(N);
6          $I_{updated}$, $O_{updated}$ = PropOneNode(N, $I_s$, $O_s$);
7          foreach $T \in (I_{updated} \cup O_{updated})$ do
8              B = NeighborNodesOf(T);
9              S.Insert(B.Remove(N));
10     return G;
```

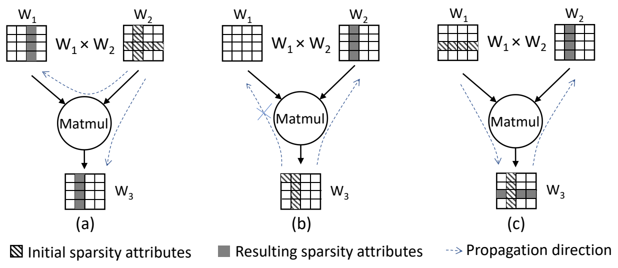

W1
W2
W1 × W2
Matmul
W3
(a)
(b)
(c)
▧ Initial sparsity attributes
■ Resulting sparsity attributes
--> Propagation direction

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, *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

Type	Computation	Computation in TeSA Algebra
Unary $f(x) \Rightarrow y$	$sin, cos, \lvert x \rvert, ReLU, \dots$	$(x = \phi) \rightarrow (y = \phi)$ $(x = \alpha) \rightarrow (y = \alpha)$
Binary $f(x,y) \Rightarrow z$	$+, -$	$((x = \phi) \wedge (y = \phi)) \rightarrow (z = \phi)$ $((x = \alpha) \vee (y = \alpha)) \rightarrow (z = \alpha)$
	$\times, \div, x^y$	$((x = \phi) \vee (y = \phi)) \rightarrow (z = \phi)$ $((x = \alpha) \wedge (y = \alpha)) \rightarrow (z = \alpha)$

Table 2: TeSA algebra on a set of attribute values. ϕ and α represent pruned and non-pruned element respectively.

differently. If an output tensor has a low precision (*e.g.,* 4bit) while the input tensor's precision is high (*e.g.,* 16bit), the input may use fewer quantization bits with little impacts on output (*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 (*e.g.,* $+$, $\times$ in `Matmul`). To capture such property, SparTA defines a TeSA algebra that maps the an operator's element-wise computation to a set with two elements, {*pruned*, *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 (*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_t) 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`)

Algorithm 2: TeSA attribute propagation rules.

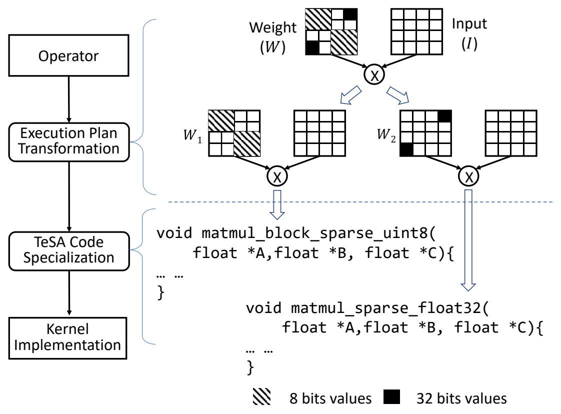
```
   Data: $N$: a node in DFG, $I_s$: node $N$'s inputs, $O_s$: node $N$'s outpus.
   Result: Updated input/ouput TeSAs after propagation on $N$.
11 Function PruningPropRule($N$, $I_s$, $O_s$):
12     $S_{updated} = \emptyset$;
13     foreach $T \in (I_s \cup O_s)$ do
14         $T_{updated}$ = TensorScrambling($N$, $(I_s \cup O_s) \setminus T$);
15         if $T_{updated}$ != $T$ then
16             $S_{updated}$.Update($T_{updated}$);
17     return SplitToIsOs($S_{updated}$);
18 Function QuantizationPropRule($N$, $I_s$, $O_s$):
19     $S_{updated} = \emptyset$; $S = (I_s \cup O_s)$;
20     $D_{calib}$ = GetCalibrationDataOf($N$);
21     foreach $T \in (I_s \cup O_s)$ do
22         $T_{updated}$ = LowerBitAndFinetune($N$, $S$, $T$, $D_{calib}$);
23         if $T_{updated}$ != $T$ then
24             $S$.Update($T_{updated}$);
25             $S_{updated}$.Update($T_{updated}$);
26     return SplitToIsOs($S_{updated}$);
27 RegisterPropRule(PruningPropRule);
28 RegisterPropRule(QuantizationPropRule);
29 Function PropOneNode($N$, $I_s$, $O_s$):
30     $I_{updated} = O_{updated} = \emptyset$;
31     foreach RegisteredPropRule do
32         $I_{proped}$, $O_{proped}$ = RegisteredPropRule($N$, $I_s$, $O_s$);
33         $I_s$.Update($I_{proped}$); $O_s$.Update($O_{proped}$);
34         $I_{updated}$.Update($I_{proped}$);
35         $O_{updated}$.Update($O_{proped}$);
36     return $I_{updated}$, $O_{updated}$;
```

gI_j's TeSA, which has the same shape and value of that of I_j.

We use Operator $Y = AX + B$ as an example to illustrate output-to-input and input-to-input propagation, where Y is output tensor, and A, B, and X are input tensors. To derive the TeSA of A, B, X, we take the derivative of the operator with respect to A, B, and X, *i.e.,* $gA = gY \times X^T$, $gB = gY$, $gX = A^T \times gY$, respectively. The sparsity propagation from output tensor Y to input tensor A uses $gA = gY \times X^T$. gY has the same TeSA as Y. Given the TeSA of Y and X, gA's TeSA can be inferred using either TeSA algebra or Tensor Scrambling. The propagation from X to A also uses this backward computation, which is input-to-input propagation. Similarly, the TeSA of B and X can be inferred from Y using $gB = gY$ and $gX = A^T \times gY$, respectively. It is obvious from $gB = gY$ that B has the same TeSA as Y.

The propagation rule of pruned elements can be realized in line 11 of Algorithm 2. Every input/output TeSA of node N is computed (line 14), *i.e.,* propagating the sparsity in other TeSAs (*i.e.,* $(I_s \cup O_s) \setminus T$) to this TeSA (*i.e.,* T). This function returns the updated input and output TeSAs separately (*i.e.,* line 17). Note that here Tensor Scrambling can be replaced with tensor algebra.

Quantization rule. For propagation of quantization attributes, the key is to find tensors with unnecessarily high quantization precision. SparTA defines a quantization rule (line 18 of Algorithm 2) that borrows the idea of knowledge distillation [45,46] to identify such tensors. That is, to identify whether the information passed through an operator can be 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.



```
void matmul_block_sparse_uint8(
    float *A,float *B, float *C){
… …
}
```

```
void matmul_sparse_float32(
    float *A,float *B, float *C){
… …
}
```

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

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 8bit quantization and one fine-grained element uses 32bit. 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 profiled 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.

```
   Data: $N$: An $m$ inputs single output operator to be transformed; $P$: A
         transformation policy.
37 Function TransformOp($N$):
       /* HasBudget determines the number of
          transformation options that can be searched */
38     while $P$.HasBudget() and $P$.PerfNotSatisfied() do
           /* loop body is one transformation option */
39         $S = []$;
40         $I_1,\dots,I_m$, $O$ = InputsOutputsTeSAOf($N$);
41         foreach $i \in 1,\dots,m$ do
42             $T_i = P$.TransformTeSA($I_i$);
43         $T_o = P$.TransformTeSA($O$);
44         foreach $\langle t_1,\dots,t_m,t_o\rangle \in T_1 \times \cdots \times T_m \times T_o$ do
45             $N_{sub}$ = SpecializeOp(op=$N$, in=$t_1,\dots,t_m$, out=$t_o$);
46             $S$.Append($N_{sub}$);
47         $P$.RecordPerf($G$=ComposeToGraph($S$), Profile($G$));
48     return $P$.BestTransformation();
49 Function $P$::TransformTeSA($T$):
       /* return a new transformation option per run */
50     $T_{transformed} = []$;
       /* BitOption: (i) 4 and 8, (ii) only 8, if
          hardware natively supports 4-bit and 8-bit */
51     $T_1,\dots,T_k$ = self.BitRounding($T$, self.SampleBitOption());
52     foreach $i \in 1,\dots,k$ do
53         $T_i^1,\dots,T_i^j$ = self.WeightedBlockCover($T_i$);
54         $T_{transformed}$.Extend($T_i^1,\dots,T_i^j$);
55     return $T_{transformed}$;
56 Function $P$::WeightedBlockCover($T$):
57     $B_{chosen} = []$;
       /* covering non-pruned elements to blocks using
          every available block size, to produce $B$ */
58     $B$ = self.AllCoveredBlocks($T$, self.AvailableBlockSizes());
59     while not AllElementsCovered($T$, $B_{chosen}$) do
60         $B_{cost}$ = self.UpdateBlockCost($B$);
61         $b$ = BlockWithMinCost($B_{cost}$);
62         $B_{chosen}$.Append($b$);
63         $B = B - b$;
       /* decompose $T$ to the TeSAs with different block
          sizes based on $B_{chosen}$ */
64     return DecomposedTeSAs($T$, $B_{chosen}$);
```

`TransformTeSA` (line 49) implements the logic of transforming a TeSA. It first decomposes TeSA in `BitRounding`, based on both the quantization bit width that the TeSA contains and the possible quantization bit width supported by the hardware. For example, if the hardware supports both 4-bit and 8-bit instructions, there are at least two rounding options: (i) rounding to 4-bit and 8-bit accordingly, (ii) all rounding

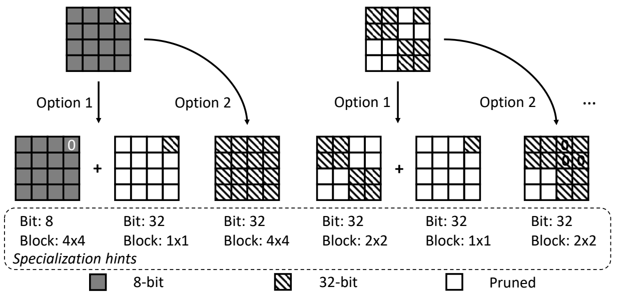


Figure 6: Multiple transformation plans produced by a transformation policy. The specialization hints are used by the second pass compilation for code specialization.

to 8-bit. For each TeSA returned by `BitRounding`, function `WeightedBlockCover` chooses one or multiple proper block sizes to cover the non-pruned elements, which we treat as a *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.,* $\frac{block\_weight}{num\_covered\_elements}$), until all non-pruned elements are covered (line 56).

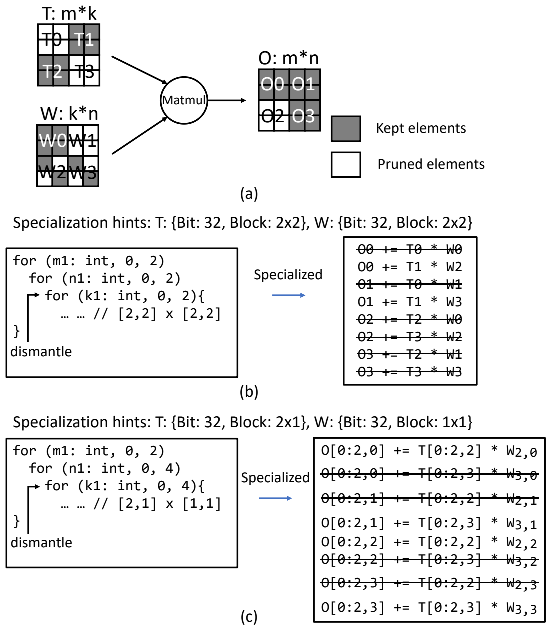
Transformation policy P can be further customized to incorporate new optimizations (*e.g.,* supporting Sparse Tensor Cores [1] detailed in §5.3).

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

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.,* `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 `dismantle` that jointly performs loop unrolling and DCE. When `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). `dismantle` is applied on the third loop, thus the top three loops are unrolled, generating eight small Matmuls (*i.e.,* [2,2]x[2,2]). According to the sparsity pattern in Figure 7(a), six of them are dead computation and can be eliminated. In essence, `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]x[1,1]), which enables


T: m*k
T0 T1
T2 T3
W: k*n
W0 W1
W2 W3
Matmul
O: m*n
O0 O1
O2 O3
Kept elements
Pruned elements
(a)
Specialization hints: T: {Bit: 32, Block: 2x2}, W: {Bit: 32, Block: 2x2}
dismantle
Specialized
(b)
Specialization hints: T: {Bit: 32, Block: 2x1}, W: {Bit: 32, Block: 1x1}
dismantle
Specialized
(c)

```
for (m1: int, 0, 2)
  for (n1: int, 0, 2)
    for (k1: int, 0, 2){
      … … // [2,2] x [2,2]
}
```

~~O0 += T0 * W0~~
O0 += T1 * W2
~~O1 += T0 * W1~~
O1 += T1 * W3
~~O2 += T2 * W0~~
~~O2 += T3 * W2~~
~~O3 += T2 * W1~~
~~O3 += T3 * W3~~

```
for (m1: int, 0, 2)
  for (n1: int, 0, 4)
    for (k1: int, 0, 4){
      … … // [2,1] x [1,1]
}
```

O[0:2,0] += T[0:2,2] * $W_{2,0}$
~~O[0:2,0] += T[0:2,3] * $W_{3,0}$~~
~~O[0:2,1] += T[0:2,2] * $W_{2,1}$~~
O[0:2,1] += T[0:2,3] * $W_{3,1}$
O[0:2,2] += T[0:2,2] * $W_{2,2}$
~~O[0:2,2] += T[0:2,3] * $W_{3,2}$~~
~~O[0:2,3] += T[0:2,2] * $W_{2,3}$~~
O[0:2,3] += T[0:2,3] * $W_{3,3}$

Figure 7: Sparsity-aware code specialization, leveraging specialization hints generated during execution plan transformation. (a) is a sparse Matmul, (b) and (c) are its specialized kernel code with different transformation plans. $T[x:y,z]$ denotes the elements on row x to y and column z, $W_{x,y}$ denotes the value on row x and column y of W.

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.,* cuSPARSE [3], 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.,* `DP4A` [13] or `wmma` [5] for 8bit, `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 `wmma` is the Matmul [16,16]x[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 `DP4A` with a shape [1,4]x[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 on NVIDIA GPU, AMD GPU, and Intel CPU. Overall, our key findings include:

- SparTA significantly reduces the inference latency of sparse DNN models with less memory consumption. The speedup is up to 10.6x, 5.0x, 7.5x, 20.1x, 5.8x, 5.6x, 1.7x over PyTorchJIT, TensorRT, TVM, TVM sparse[2], Rammer, Rammer sparse[3], and OpenVINO (CPU), respectively. The average speedup is 3.8x, 2.6x, 4.2x, 8.4x, 3.0x, 3.2x, 1.7x. (§5.1)
- 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)

[2]Excluding cases where kernel tuning failed for TVM and TVM sparse.
[3]The state-of-the-art sparse kernels wrapped in Rammer.

Model	Type	Sparsity	Ratio	Acc (%)
BERT [34]	NLP {*Matmul*}	Structured [87]	95%	89.7->88.49
		Unstructured [43]	95%	89.7->88.67
		Structured+8bit [52]	95%	89.7->88.03
		Mixed Sparsity [48, 84]	94.99%	89.7->88.63
MobileNet [49]	CV {*Conv*}	Structured [94]	60%	78.27->75.62
		Unstructured [43]	95%	78.27->64.15
		Structured+8bit [52]	60%	78.27->75.13
HuBERT [50]	Speech {*Conv*, *Matmul*}	Structured [56, 62]	80%	95.61->95.1
		Unstructured [43]	95%	95.61->95.55
		Structured+8bit [52]	80%	95.61->94.3

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.

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]) applied 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, 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.

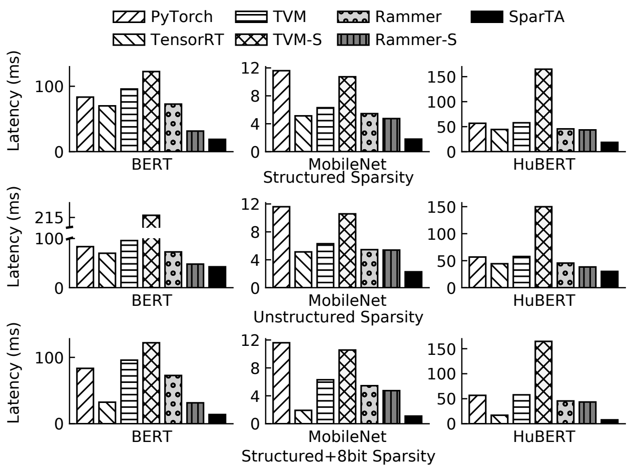


Figure 8: Inference latency of different models with three sparsity patterns on NVIDIA 2080 Ti.

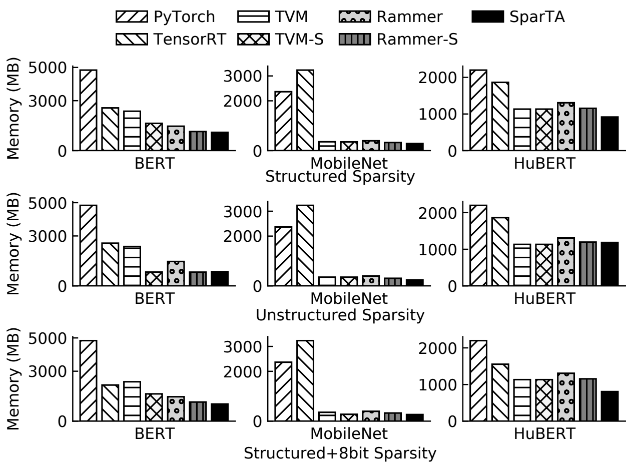


Figure 9: GPU memory usage of different models with three sparsity patterns on NVIDIA 2080 Ti.

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, 1.5x faster than TensorRT on BERT, MobileNet, HuBERT, respectively. Rammer-S still uses Sputnik. SparTA outperforms Rammer-S by 1.13x, 2.4x, 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-

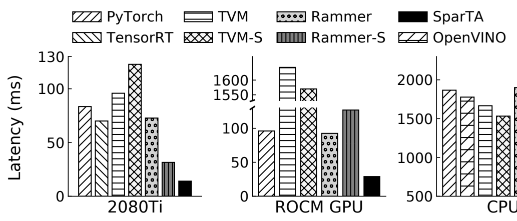


Figure 10: Mixed Sparsity end-to-end latencies.

sorRT performs much better than PyTorch, TVM, and Rammer, because it supports low-bit computations with Tensor Cores. SparTA outperforms TensorRT by 2.3x, 1.7x, 2.2x on BERT, MobileNet, HuBERT, respectively. This is because, besides leveraging the hardware instruction `wmma` of Tensor Core, SparTA further combines the optimization for structured sparsity. Compared to Rammer-S (also using Sputnik), SparTA is 2.3x, 4.3x, 5.6x faster on BERT, MobileNet, HuBERT, respectively, because Rammer-S has limited optimization (*e.g.,* missing low-bit instructions) for each single sparse operator, while SparTA does holistic optimizations on the model, *e.g.,* sparsity propagation, operator transformation and specialization. The speedup shows the combined gain from structured sparsity (*i.e.,* the first row of Figure 8) and low-bit instructions. The memory usage of SparTA (*i.e.,* the third row of Figure 9) is the lowest: the memory saving comes from both low-bit values and pruned elements. This highlights the benefit of SparTA and in particular TeSA that uses a unified abstraction for pruning and quantization to make such a joint optimization possible.

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 2080Ti. 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 large 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

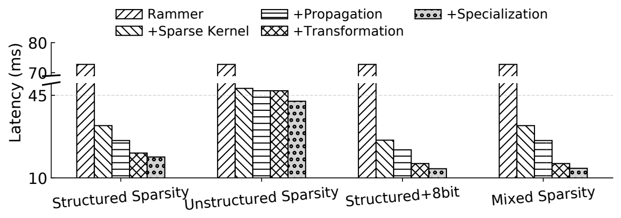


Figure 11: Performance breakdown of SparTA for different sparsity patterns of BERT on 2080 Ti. Each bar shows the result of applying the additional optimization labeled on this bar from the previous one.

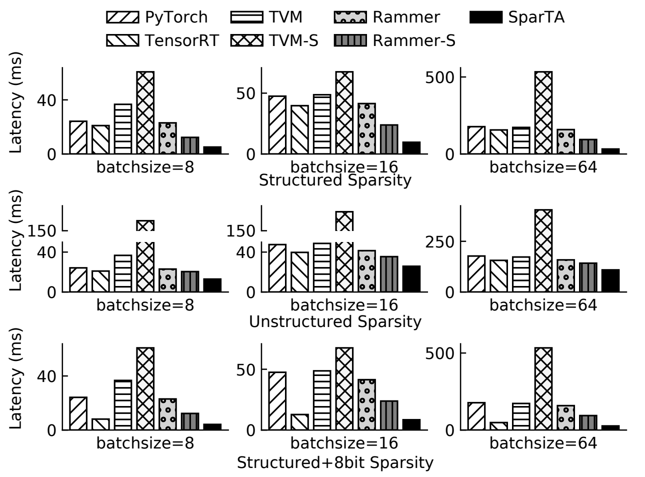


Figure 12: Inference latency of BERT under different batch sizes on NVIDIA 2080Ti.

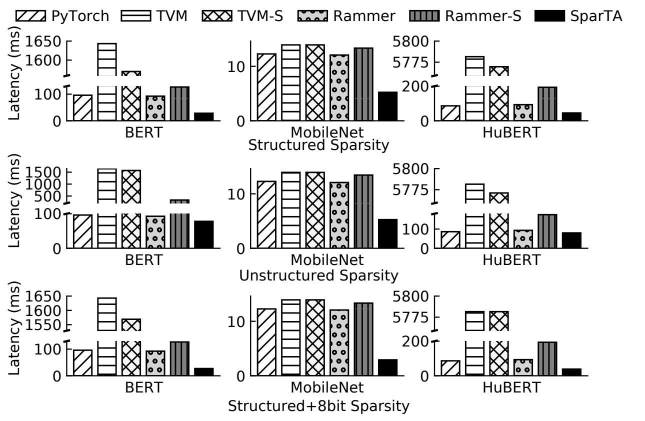


Figure 13: Inference latency of different models with three sparsity patterns on AMD Radeon VII.

have provided reasonably good default kernel schedules for some popular DNN models including MobileNet. Compared to Rammer, SparTA is up to 3.4x, 4.1x, 2.4x faster for BERT, MobileNet, and HuBERT, respectively, on the three sparsity patterns. Rammer-S uses hipSPARSE on ROCm GPU, the speedup of SparTA over Rammer-S is up to 4.7x, 4.6x, 5.0x for BERT, MobileNet, and HuBERT, respectively.

For mixed sparsity of BERT shown in the middle of Figure 10, SparTA is 3.3x, 56.7x, 54.1x, 3.2x, 4.4x faster than 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

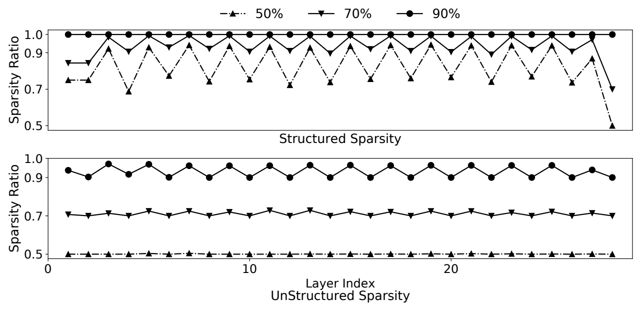


Figure 14: Propagated sparsity across the layers for different sparsity patterns on MobileNet

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-

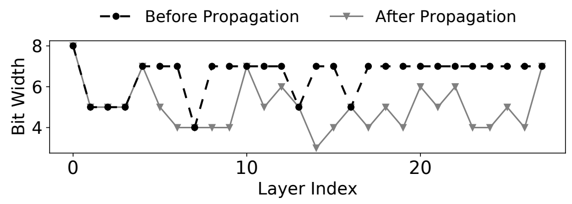


Figure 15: The quantization (bit width) of each layer in MobileNet before and after propagation.

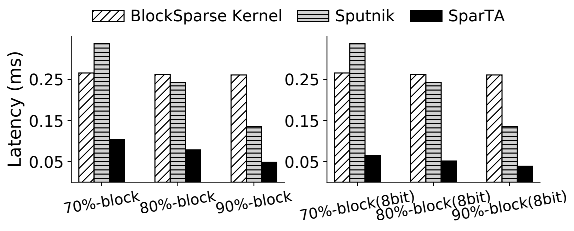


Figure 16: The performance of execution plan transformation for mixed sparsity patterns. B is sparsified for the matrix multiplication $A \times B$ (1024x1024x1024). "X%-block" means X% block sparsity mixed with 1% unstructured sparsity.

ing to explore the bit width on each weight and activation tensor. The candidate bit width is between 0 to 8. After exploring 50 different configurations of bit width, we pick the best one, whose accuracy on ImageNet is 64.6%. The propagation rule is then applied to that configuration. The experiment result is shown in Figure 15: 18 out of 28 layers reduces its bit width (from 7bit to around 4bit), while the model accuracy after propagation only drops slightly, from 64.6% to 64.2%. From another point of view, our propagation rule for quantization is complementary to the search algorithm (*e.g.,* reinforcement learning, simulated annealing [58]) on quantization bits. A proper combination of them could improve search efficiency, which is an interesting future work.

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 pattern, SparTA transforms the operator into two sub-operators with structured sparsity and unstructured sparsity, respectively. After transformation, SparTA becomes 2.5x, 3.3x, 5.4x faster than BlockSparse on the three sparsity ratios, respectively. The performance of BlockSparse has little change, because the blocks to cover those unstructured elements construct the major of the blocks in the computation. The speedup of SparTA over Sputnik is 3.2x, 3.1x, 2.8x, respectively. Sputnik performs the worst on 70%-block, because it treats each block as 1,024 unstructured elements, missing optimization opportunities. For the second sparsity pattern, the performance gain of SparTA is much higher, *i.e.,* 4.1x, 5.1x, 6.7x faster than BlockSparse, 5.2x, 4.7x, 3.5x faster than Sputnik. This is because SparTA further leverages low-bit instructions for the computation of those 32x32 blocks.

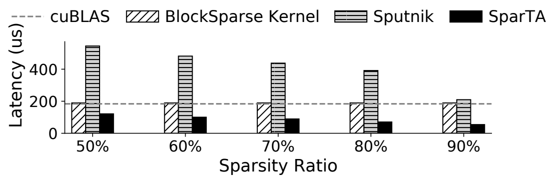


Figure 17: The performance of execution plan transformation leveraging Sparse Tensor Cores. B is sparsified with different sparsity ratios for $A \times B$ (1024x1024x1024).

The transformation can effectively leverage special hardware like Sparse Tensor Core [1]. Sparse Tensor Core has a strict requirement on tensor's sparsity pattern, *e.g.,* one element should be pruned in a $[1 \times 2]$ tile (50% sparsity ratio). To leverage Sparse Tensor Cores for the generic unstructured sparsity, we develop a new transformation policy to decompose an unstructured sparse tensor into two: one follows the sparsity requirement of Sparse Tensor Cores, the other contains the elements not included in the first one. The first one uses Sparse Tensor Cores, while the other uses our specialized sparse kernel. To evaluate the transformation, we randomly generate an unstructured sparse tensor whose sparsity ratio ranges from 50% to 90% in one input of Matmul. The experiment runs on NVIDIA A100, the result is shown in Figure 17. SparTA performs better than both BlockSparse and Sputnik, as it leverages cuSPARSELt [6], a library optimized for Sparse Tensor Cores, for sub-Matmul that costs around 40 us. The other sub-Matmul has 12.5%, 8%, 4.5%, 2.0%, 0.5% sparsity ratio respectively. BlockSparse shows a similar performance to cuBLAS. As the sparsity is randomly introduced, it actually computes a dense Matmul.

Sparsity Pattern	1	2	3	4	5
Origin Latency(ms)	1.293	0.361	2.808	1.263	0.189
SparTA Latency(ms)	0.436	0.191	0.599	0.569	0.101
Best Block Size	32x128	128x32	32x128	32x64	128x64

Table 4: Block size transformation

During the transformation, SparTA also finds the best 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.

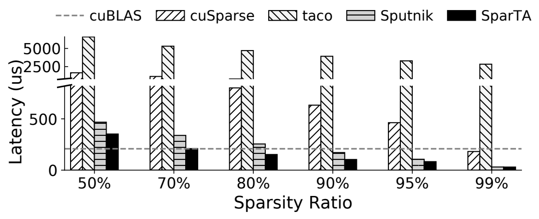


Figure 18: Comparison of cuSPARSE, taco, Sputnik, and SparTA on matrix multiplication (1024x1024x1024) with fine-grained sparsity under different sparsity ratios. B is sparse for $A \times B$.

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.

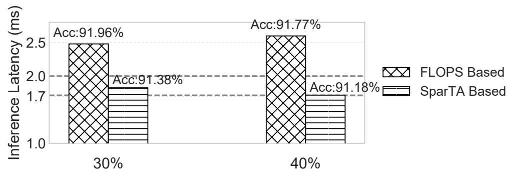


Figure 19: The comparison of using real latency or FLOPS as metric to explore sparse models by Simulated Annealing.

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 pretrain 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

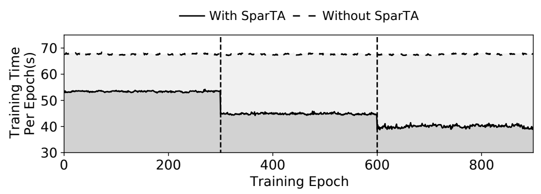


Figure 20: The improvement on exploration time when using SparTA-accelerated sparse model.

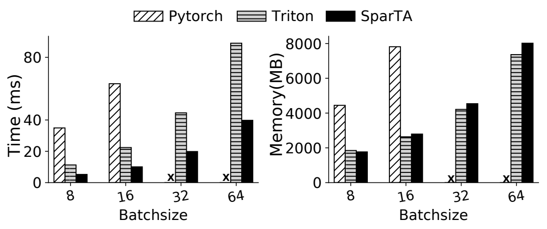


Figure 21: The time of training one batch and GPU memory consumption of 3DNA on NVIDIA 2080Ti.

batch sizes, mainly because SparTA specializes the block sizes (*e.g.,* 32x64, 32x32) covered on non-zero values in the matrix multiplications of 3DNA. The memory usage of SparTA and Triton are similar. As the SparTA-based version currently relies on PyTorch for the management of some intermediate tensor, it is possible to further improve the memory usage by moving it to the SparTA side.

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

[1] Accelerating inference with sparsity using the nvidia ampere architecture and nvidia tensorrt. https://developer.nvidia.com/blog/accelerating-inference-with-sparsity-using-ampere-and-tensorrt/, 2021.

[2] The api reference guide for cublas, the cuda basic linear algebra subroutine library. https://docs.nvidia.com/cuda/cublas/index.html, 2021.

[3] The api reference guide for cusparse, the cuda sparse matrix library. https://docs.nvidia.com/cuda/cusparse/index.html, 2021.

[4] Cloud tpu: Train and run machine learning models faster than ever before. https://cloud.google.com/tpu, 2021.

[5] Cuda c++ programming guide. https://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html#wmma, 2021.

[6] cusparselt: A high-performance cuda library for sparse matrix-matrix multiplication. https://docs.nvidia.com/cuda/cusparselt/index.html, 2021.

[7] Einstein notation. https://en.wikipedia.org/wiki/Einstein_notation, 2021.

[8] Geforce rtx 2080 ti. https://www.nvidia.com/en-us/geforce/graphics-cards/rtx-2080-ti/, 2021.

[9] Graphcore. https://www.graphcore.ai/, 2021.

[10] Intel advanced vector extensions 512 (intel avx512). https://www.intel.com/content/www/us/en/architecture-and-technology/avx-512-overview.html, 2021.

[11] Intel fpgas and programmable devices. https://www.intel.com/content/www/us/en/products/programmable.html, 2021.

[12] Intel oneapi math kernel library. https://www.intel.com/content/www/us/en/developer/tools/oneapi/onemkl.html, 2021.

[13] Mixed-precision programming with cuda 8. https://developer.nvidia.com/blog/mixed-precision-programming-cuda-8/, 2021.

[14] Open neural network exchange. https://onnx.ai/, 2021.

[15] Openvino: Deploy high-performance, deep learning inference. https://www.intel.com/content/www/us/en/developer/tools/openvino-toolkit/overview.html, 2021.

[16] Reproducing oopsla 2020 results. https://github.com/tensor-compiler/taco/tree/oopsla2020, 2021.

[17] Rocm sparse marshalling library. https://github.com/ROCmSoftwarePlatform/hipSPARSE, 2021.

[18] The sdk for high-performance deep learning inference. https://docs.nvidia.com/deeplearning/tensorrt/, 2021.

[19] Set cover problem. https://en.wikipedia.org/wiki/Set_cover_problem, 2021.

[20] The world's first 7nm gaming gpu. https://www.amd.com/en/products/graphics/amd-radeon-vii, 2021.

[21] Triton. https://github.com/openai/triton.git, 2021.

[22] Tvm sparsity code. https://github.com/apache/tvm/blob/254563a3140cf63fe77a46058688209de3aa213c/python/tvm/topi/cuda/sparse.py#L96, 2021.

[23] Martín Abadi, Paul Barham, Jianmin Chen, Zhifeng Chen, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Geoffrey Irving, Michael Isard, et al. Tensorflow: A system for large-scale machine learning. In *12th {USENIX} symposium on operating systems design and implementation ({OSDI} 16)*, pages 265–283, 2016.

[24] Nathan Bell and Michael Garland. Efficient sparse matrix-vector multiplication on cuda. Technical report, Citeseer, 2008.

[25] Tom B Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared Kaplan, Prafulla Dhariwal, Arvind Neelakantan, Pranav Shyam, Girish Sastry, Amanda Askell, et al. Language models are few-shot learners. *arXiv preprint arXiv:2005.14165*, 2020.

[26] Aydin Buluç, Jeremy T Fineman, Matteo Frigo, John R Gilbert, and Charles E Leiserson. Parallel sparse matrix-vector and matrix-transpose-vector multiplication using compressed sparse blocks. In *Proceedings of the twenty-first annual symposium on Parallelism in algorithms and architectures*, pages 233–244, 2009.

[27] Brad Calder, Peter Feller, Alan Eustace, et al. Value profiling and optimization. *Journal of Instruction Level Parallelism*, 1(1):1–6, 1999.

[28] Shijie Cao, Chen Zhang, Zhuliang Yao, Wencong Xiao, Lanshun Nie, Dechen Zhan, Yunxin Liu, Ming Wu, and Lintao Zhang. Efficient and effective sparse lstm on fpga with bank-balanced sparsity. In *Proceedings of the 2019 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, pages 63–72, 2019.

[29] Tianqi Chen, Thierry Moreau, Ziheng Jiang, Lianmin Zheng, Eddie Yan, Haichen Shen, Meghan Cowan, Leyuan Wang, Yuwei Hu, Luis Ceze, et al. {TVM}: An automated end-to-end optimizing compiler for deep learning. In *13th {USENIX} Symposium on Operating Systems Design and Implementation ({OSDI} 18)*, pages 578–594, 2018.

[30] Stephen Chou, Fredrik Kjolstad, and Saman Amarasinghe. Automatic generation of efficient sparse tensor format conversion routines. In *Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 823–838, 2020.

[31] Matthieu Courbariaux, Itay Hubara, Daniel Soudry, Ran El-Yaniv, and Yoshua Bengio. Binarized neural networks: Training deep neural networks with weights and activations constrained to+ 1 or-1. *arXiv preprint arXiv:1602.02830*, 2016.

[32] Luis Damas and Robin Milner. Principal type-schemes for functional programs. In *Proceedings of the 9th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 207–212, 1982.

[33] Jia Deng, Wei Dong, Richard Socher, Li-Jia Li, Kai Li, and Li Fei-Fei. Imagenet: A large-scale hierarchical image database. In *2009 IEEE conference on computer vision and pattern recognition*, pages 248–255. Ieee, 2009.

[34] Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. Bert: Pre-training of deep bidirectional transformers for language understanding. *arXiv preprint arXiv:1810.04805*, 2018.

[35] Johannes Doerfert and Hal Finkel. Compiler optimizations for openmp. In *International Workshop on OpenMP*, pages 113–127. Springer, 2018.

[36] Ahmed Elthakeb, Prannoy Pilligundla, FatemehSadat Mireshghallah, Amir Yazdanbakhsh, Sicuan Gao, and Hadi Esmaeilzadeh. Releq: An automatic reinforcement learning approach for deep quantization of neural networks. In *NeurIPS ML for Systems workshop, 2018*, 2019.

[37] William Fedus, Barret Zoph, and Noam Shazeer. Switch transformers: Scaling to trillion parameter models with simple and efficient sparsity. *arXiv preprint arXiv:2101.03961*, 2021.

[38] Jeffrey S Foster, Manuel Fähndrich, and Alexander Aiken. A theory of type qualifiers. *ACM SIGPLAN Notices*, 34(5):192–203, 1999.

[39] Trevor Gale, Matei Zaharia, Cliff Young, and Erich Elsen. Sparse GPU kernels for deep learning. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis, SC 2020*, 2020.

[40] Amir Gholami, Sehoon Kim, Zhen Dong, Zhewei Yao, Michael W Mahoney, and Kurt Keutzer. A survey of quantization methods for efficient neural network inference. *arXiv preprint arXiv:2103.13630*, 2021.

[41] Song Han, Xingyu Liu, Huizi Mao, Jing Pu, Ardavan Pedram, Mark A Horowitz, and William J Dally. Eie: efficient inference engine on compressed deep neural network. *ACM SIGARCH Computer Architecture News*, 44(3):243–254, 2016.

[42] Song Han, Huizi Mao, and William J Dally. Deep compression: Compressing deep neural networks with pruning, trained quantization and huffman coding. *arXiv preprint arXiv:1510.00149*, 2015.

[43] Song Han, Jeff Pool, John Tran, and William J Dally. Learning both weights and connections for efficient neural networks. *arXiv preprint arXiv:1506.02626*, 2015.

[44] Yang He, Ping Liu, Ziwei Wang, Zhilan Hu, and Yi Yang. Filter pruning via geometric median for deep convolutional neural networks acceleration. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, pages 4340–4349, 2019.

[45] Yihui He, Ji Lin, Zhijian Liu, Hanrui Wang, Li-Jia Li, and Song Han. Amc: Automl for model compression and acceleration on mobile devices. In *Proceedings of the European Conference on Computer Vision (ECCV)*, pages 784–800, 2018.

[46] Geoffrey Hinton, Oriol Vinyals, and Jeff Dean. Distilling the knowledge in a neural network. *arXiv preprint arXiv:1503.02531*, 2015.

[47] Torsten Hoefler, Dan Alistarh, Tal Ben-Nun, Nikoli Dryden, and Alexandra Peste. Sparsity in deep learning: Pruning and growth for efficient inference and training in neural networks. *arXiv preprint arXiv:2102.00554*, 2021.

[48] Changwan Hong, Aravind Sukumaran-Rajam, Israt Nisa, Kunal Singh, and P Sadayappan. Adaptive sparse tiling for sparse matrix multiplication. In *Proceedings of the 24th Symposium on Principles and Practice of Parallel Programming*, pages 300–314, 2019.

[49] Andrew G Howard, Menglong Zhu, Bo Chen, Dmitry Kalenichenko, Weijun Wang, Tobias Weyand, Marco Andreetto, and Hartwig Adam. Mobilenets: Efficient convolutional neural networks for mobile vision applications. *arXiv preprint arXiv:1704.04861*, 2017.

[50] Wei-Ning Hsu, Benjamin Bolte, Yao-Hung Hubert Tsai, Kushal Lakhotia, Ruslan Salakhutdinov, and Abdelrahman Mohamed. Hubert: Self-supervised speech representation learning by masked prediction of hidden units. *arXiv preprint arXiv:2106.07447*, 2021.

[51] Shankar Iyer, Nikhil Dandekar, Kornél Csernai, et al. First quora dataset release: Question pairs. *data. quora. com*, 2017.

[52] Benoit Jacob, Skirmantas Kligys, Bo Chen, Menglong Zhu, Matthew Tang, Andrew Howard, Hartwig Adam, and Dmitry Kalenichenko. Quantization and training of neural networks for efficient integer-arithmetic-only inference. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pages 2704–2713, 2018.

[53] Fredrik Kjolstad, Shoaib Kamil, Stephen Chou, David Lugato, and Saman Amarasinghe. The tensor algebra compiler. *Proceedings of the ACM on Programming Languages*, 1(OOPSLA):1–29, 2017.

[54] Yann LeCun, John S Denker, and Sara A Solla. Optimal brain damage. In *Advances in neural information processing systems*, pages 598–605, 1990.

[55] Namhoon Lee, Thalaiyasingam Ajanthan, and Philip HS Torr. Snip: Single-shot network pruning based on connection sensitivity. *arXiv preprint arXiv:1810.02340*, 2018.

[56] Hao Li, Asim Kadav, Igor Durdanovic, Hanan Samet, and Hans Peter Graf. Pruning filters for efficient convnets. *arXiv preprint arXiv:1608.08710*, 2016.

[57] Darryl Lin, Sachin Talathi, and Sreekanth Annapureddy. Fixed point quantization of deep convolutional networks. In *International conference on machine learning*, pages 2849–2858. PMLR, 2016.

[58] Ning Liu, Xiaolong Ma, Zhiyuan Xu, Yanzhi Wang, Jian Tang, and Jieping Ye. Autocompress: An automatic dnn structured pruning framework for ultra-high compression rates. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 34, pages 4876–4883, 2020.

[59] Zhuang Liu, Jianguo Li, Zhiqiang Shen, Gao Huang, Shoumeng Yan, and Changshui Zhang. Learning efficient convolutional networks through network slimming. In *Proceedings of the IEEE International Conference on Computer Vision*, pages 2736–2744, 2017.

[60] Lingxiao Ma, Zhiqiang Xie, Zhi Yang, Jilong Xue, Youshan Miao, Wei Cui, Wenxiang Hu, Fan Yang, Lintao Zhang, and Lidong Zhou. Rammer: Enabling holistic deep learning compiler optimizations with rtasks. In *14th {USENIX} Symposium on Operating Systems Design and Implementation ({OSDI} 20)*, pages 881–897, 2020.

[61] Huizi Mao, Song Han, Jeff Pool, Wenshuo Li, Xingyu Liu, Yu Wang, and William J Dally. Exploring the regularity of sparse structure in convolutional neural networks. *arXiv preprint arXiv:1705.08922*, 2017.

[62] Paul Michel, Omer Levy, and Graham Neubig. Are sixteen heads really better than one? *arXiv preprint arXiv:1905.10650*, 2019.

[63] Sharan Narang, Eric Undersander, and Gregory Diamos. Block-sparse recurrent neural networks. *arXiv preprint arXiv:1711.02782*, 2017.

[64] Keiron O'Shea and Ryan Nash. An introduction to convolutional neural networks. *arXiv preprint arXiv:1511.08458*, 2015.

[65] Angshuman Parashar, Minsoo Rhu, Anurag Mukkara, Antonio Puglielli, Rangharajan Venkatesan, Brucek Khailany, Joel Emer, Stephen W Keckler, and William J Dally. Scnn: An accelerator for compressed-sparse convolutional neural networks. *ACM SIGARCH Computer Architecture News*, 45(2):27–40, 2017.

[66] Eunhyeok Park, Dongyoung Kim, and Sungjoo Yoo. Energy-efficient neural network accelerator based on outlier-aware low-precision computation. In *2018 ACM/IEEE 45th Annual International Symposium on Computer Architecture (ISCA)*, pages 688–698. IEEE, 2018.

[67] Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, Alban Desmaison, Andreas Kopf, Edward Yang, Zachary DeVito, Martin Raison, Alykhan Tejani, Sasank Chilamkurthy, Benoit Steiner, Lu Fang, Junjie Bai, and Soumith Chintala. Pytorch: An imperative style, high-performance deep learning library. In H. Wallach, H. Larochelle, A. Beygelzimer, F. d'Alché-Buc, E. Fox, and R. Garnett, editors, *Advances in Neural Information Processing Systems 32*, pages 8024–8035. Curran Associates, Inc., 2019.

[68] Ali Pinar and Michael T Heath. Improving performance of sparse matrix-vector multiplication. In *SC'99: Proceedings of the 1999 ACM/IEEE Conference on Supercomputing*, pages 30–30. IEEE, 1999.

[69] Ram Rangan, Mark W Stephenson, Aditya Ukarande, Shyam Murthy, Virat Agarwal, and Marc Blackstein. Zeroploit: Exploiting zero valued operands in interactive gaming applications. *ACM Transactions on Architecture and Code Optimization (TACO)*, 17(3):1–26, 2020.

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

[71] Mark Stephenson and Ram Rangan. Pgz: automatic zero-value code specialization. In *Proceedings of the 30th ACM SIGPLAN International Conference on Compiler Construction*, pages 36–46, 2021.

[72] Yi Tay, Mostafa Dehghani, Dara Bahri, and Donald Metzler. Efficient transformers: A survey. *arXiv preprint arXiv:2009.06732*, 2020.

[73] Naftali Tishby, Fernando C Pereira, and William Bialek. The information bottleneck method. *arXiv preprint physics/0004057*, 2000.

[74] Frederick Tung and Greg Mori. Clip-q: Deep network compression learning by in-parallel pruning-quantization. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pages 7873–7882, 2018.

[75] Frederick Tung and Greg Mori. Deep neural network compression by in-parallel pruning-quantization. *IEEE transactions on pattern analysis and machine intelligence*, 42(3):568–579, 2018.

[76] Hanrui Wang, Zhekai Zhang, and Song Han. Spatten: Efficient sparse attention architecture with cascade token and head pruning. In *2021 IEEE International Symposium on High-Performance Computer Architecture (HPCA)*, pages 97–110. IEEE, 2021.

[77] Kuan Wang, Zhijian Liu, Yujun Lin, Ji Lin, and Song Han. Haq: Hardware-aware automated quantization with mixed precision. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, pages 8612–8620, 2019.

[78] Ying Wang, Yadong Lu, and Tijmen Blankevoort. Differentiable joint pruning and quantization for hardware efficiency. In *European Conference on Computer Vision*, pages 259–277. Springer, 2020.

[79] Ziheng Wang. Sparsert: Accelerating unstructured sparsity on gpus for deep learning inference. In *Proceedings of the ACM International Conference on Parallel Architectures and Compilation Techniques*, pages 31–42, 2020.

[80] Samuel Williams, Leonid Oliker, Richard Vuduc, John Shalf, Katherine Yelick, and James Demmel. Optimization of sparse matrix-vector multiplication on emerging multicore platforms. In *SC'07: Proceedings of the 2007 ACM/IEEE Conference on Supercomputing*, pages 1–12. IEEE, 2007.

[81] Chenfei Wu, Jian Liang, Lei Ji, Fan Yang, Yuejian Fang, Daxin Jiang, and Nan Duan. NÜWA: Visual synthesis pre-training for neural visual world creation. *arXiv preprint arXiv:2111.12417*, 2021.

[82] Yuanzhong Xu, HyoukJoong Lee, Dehao Chen, Blake Hechtman, Yanping Huang, Rahul Joshi, Maxim Krikun, Dmitry Lepikhin, Andy Ly, Marcello Maggioni, et al. Gspmd: General and scalable parallelization for ml computation graphs. *arXiv preprint arXiv:2105.04663*, 2021.

[83] Haichuan Yang, Shupeng Gui, Yuhao Zhu, and Ji Liu. Automatic neural network compression by sparsity-quantization joint learning: A constrained optimization-based approach. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, pages 2178–2188, 2020.

[84] Jie Amy Yang, Jianyu Huang, Jongsoo Park, Ping Tak Peter Tang, and Andrew Tulloch. Mixed-precision embedding using a cache. *arXiv e-prints*, pages arXiv–2010, 2020.

[85] Shu-wen Yang, Po-Han Chi, Yung-Sung Chuang, Cheng-I Jeff Lai, Kushal Lakhotia, Yist Y Lin, Andy T Liu, Jiatong Shi, Xuankai Chang, Guan-Ting Lin, et al. Superb: Speech processing universal performance benchmark. *arXiv preprint arXiv:2105.01051*, 2021.

[86] Tien-Ju Yang, Andrew Howard, Bo Chen, Xiao Zhang, Alec Go, Mark Sandler, Vivienne Sze, and Hartwig Adam. Netadapt: Platform-aware neural network adaptation for mobile applications. In *Proceedings of the European Conference on Computer Vision (ECCV)*, pages 285–300, 2018.

[87] Zhewei Yao, Linjian Ma, Sheng Shen, Kurt Keutzer, and Michael W Mahoney. Mlpruning: A multilevel structured pruning framework for transformer-based models. *arXiv preprint arXiv:2105.14636*, 2021.

[88] Zhuliang Yao, Shijie Cao, Wencong Xiao, Chen Zhang, and Lanshun Nie. Balanced sparsity for efficient dnn inference on gpu. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 33, pages 5676–5683, 2019.

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

of neurons in visual cortex. *Nature communications*, 11(1):1–19, 2020.

[90] Yiren Zhao, Xitong Gao, Daniel Bates, Robert Mullins, and Cheng-Zhong Xu. Focused quantization for sparse cnns. *arXiv preprint arXiv:1903.03046*, 2019.

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

[92] Shuchang Zhou, Yuxin Wu, Zekun Ni, Xinyu Zhou, He Wen, and Yuheng Zou. Dorefa-net: Training low bitwidth convolutional neural networks with low bitwidth gradients. *arXiv preprint arXiv:1606.06160*, 2016.

[93] Maohua Zhu, Tao Zhang, Zhenyu Gu, and Yuan Xie. Sparse tensor core: Algorithm and hardware co-design for vector-wise sparse neural networks on modern gpus. In *Proceedings of the 52nd Annual IEEE/ACM International Symposium on Microarchitecture*, pages 359–371, 2019.

[94] Michael Zhu and Suyog Gupta. To prune, or not to prune: exploring the efficacy of pruning for model compression. *arXiv preprint arXiv:1710.01878*, 2017.

[95] Ling Zhuo and Viktor K Prasanna. Sparse matrix-vector multiplication on fpgas. In *Proceedings of the 2005 ACM/SIGDA 13th international symposium on Field-programmable gate arrays*, pages 63–74, 2005.

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

```
# get the source code
git clone -b sparta_artifact https://github.com/microsoft
      /SparTA.git
cd SparTA/image
# build the docker image
sudo docker build . -t artifact
# start a docker instance
sudo docker run -it --gpus all --shm-size 16G artifact

# Execute following commands in the docker
      instance
# install the sparta
mkdir workspace && cd workspace
git clone https://github.com/microsoft/SparTA && cd
      SparTA && git checkout sparta_artifact
conda activate artifact
python setup.py develop
# initialize the environment
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

```
# go into the script directory
cd script
# for 2080Ti
bash run_all_2080ti.sh
# for A100
bash run_all_a100.sh
```