A New Approach to Deep-Learning Model Sparsity via Tensor-with-Sparsity-Attribute

Ningxin Zheng
Microsoft Research

Bin Lin
Tsinghua University

Quanlu Zhang
Microsoft Research

Lingxiao Ma
Microsoft Research

Yuqing Yang
Microsoft Research

Fan Yang
Microsoft Research

Mao Yang
Microsoft Research

Lidong Zhou
Microsoft Research

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 a new approach to model sparsity via a new abstraction called *Tensor-with-Sparsity-Attribute* (or *TeSA*) to augment the default Tensor abstraction, which 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 stages, and used to create highly efficient, specialized operators, taking into account any special sparsity support from the underlying hardware. The resulting SparGen framework is flexible in accommodating more than 10 popular sparsity schemes, is efficient in delivering more than 8x speedup compared to existing solutions like TVM and cuSPARSE, and is extensible to incorporate new innovations through new sparsity attributes, new propagation rules, new optimized sparse operators, or new sparsity-aware accelerators.

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 [65]. A DNN model is usually modeled as a data flow graph (DFG), where each node in the DFG is an operator with one or multiple input and output tensors. *Model sparsity* involves introducing some specific 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) part of some tensors to zero (*e.g.,* block sparsity [42, 44] or fine-grained sparsity [29, 35, 36]); or to apply the combination of quantization and pruning to a model. With careful quantization and pruning, a DNN model can be compressed into a smaller memory footprint without losing too much accuracy. With DNN operators customized for the sparsity patterns, the resulting model will, hopefully, come with a lower inference latency.

Unfortunately, deep learning systems are not yet effective in exploiting sparsity: the increase in sparsity does not translate into corresponding gains in efficiency for a variety of reasons. First, the computation kernels for general sparse operations remain far from optimal. For example, cuSPARSE [2], 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%, albeit at a significant smaller memory footprint. Second, as DNN computation tends to take multiple stages, the sparsity patterns 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 SparGen, 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 supports *iterative exploration* of different sparsity patterns to find the best.

At the core of SparGen 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, zero weights, and block sparsity. A set of *TeSA propagation rules* guides the forward and backward propagation of sparsity attributes for end-to-end coverage. Some of those rules can be generated automatically (*e.g.,* by inferring how all zero weights are propagated via operators); others can be specified directly using domain knowledge.

With the sparse attributes in TeSA, SparGen has the opportunity to generate the best execution plan, taking into account factors such as sparsity-aware hardware acceleration and special sparse operators/kernels in certain sparsity patterns and conditions. For each operator, SparGen can perform *code specialization* to generate efficient kernels for individual cases, instead of resorting to generic sparse kernels that are known to be inefficient. This is how SparGen achieves extreme efficiency without sacrificing generality.

SparGen allows a set of *transformation rules* to be specified to produce promising execution plans for consideration. Those rules enable SparGen to decompose complex sparsity attributes into a combination of simple ones with known effective optimizations, and then to make optimal decisions across the stack by evaluating execution plans.

Because SparGen covers the entire stack all the way to codegen on accelerators, SparGen is able to provide the ground-truth performance metrics 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.

SparGen 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 hardware accelerators into transformation rules.

We have used SparGen to implement 19 popular model sparsity schemes proposed in the deep learning community. Our experiments on both Nvidia and AMD GPUs show that SparGen achieves more than 8x speedup, compared to existing solutions such as TVM and TensorRT. We plan to open source SparGen to bring the community together in this extensible and unified framework to accelerate innovations on model sparsity.

2 Background and Motivation

Various forms of sparsity. Deep learning model sparsity is an active and extensively studied research topic. Currently, there are various sparsity patterns being studied. Coarse-grained sparsity, including channel-granularity sparsity and block sparsity [37, 40, 42, 44], involves pruning a channel or a sub-block of tensors (e.g., weight or activation tensor) associated with some operator. With fine-grained sparsity, any element of a tensor [29, 35, 36] might be pruned. Quantization algorithms represent models at different levels of precision (*e.g.,* binarized models [20], 8-bit models [33, 68]), and even with different, mixed precision across neural network layers [24, 38, 55] or within a single tensor [47, 62]. Some research further combines pruning and quantization in order to achieve high accuracy under the strict latency and memory constraints [28, 53, 54, 57, 61, 66]. Overall, pruning and quantization have been shown effective in reducing the size and latency of certain deep learning models, sometimes by more than 10 times, without losing much accuracy [28, 59].

Table 1. Speed of matrix multiplication (1024*1024*1024) in cuSPARSE and cuBLAS (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

The myth of FLOPS. Model sparsity does not translate directly into performance benefits. The practice of using "proxy metric" (e.g., FLOPS, or Floating point operations per second) to evaluate the effect of their proposal 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 actually model inference latency may even become higher with a default sparse kernel.

One of the reasons is that the current generic sparse kernel implementation is sub-optimal. A sparse kernel tends to set a threshold (e.g., 90%) to decide whether or not a tensor or a row/column is sparse or dense [2, 34]. Such a coarse-grained sparsity assumption provides limited optimization opportunities to a sparse kernel. For example, one can only use sparse encoding (e.g., Compressed Sparse Row [15]) to reduce memory usage. As a result, a generic sparse kernel library like cuSPARSE [2] can outperform cuBLAS, its dense counterpart [1], only in some extreme sparse case (98%), as shown in Table 1.

The diminishing end-to-end return. Sparsity algorithms often focus on exploring the sparsity of a certain DNN operator (e.g., convolution [45]). 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, which may introduce sophisticated sparsity patterns that are difficult to understand or optimize, leading to diminishing end-to-end return from sparsity.

As shown in Figure 1, the tensor W_2 illustrates a fine-grained sparsity pattern (63% sparsity) Even without further complication, the 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 end-to-end.

Across-stack sparsity innovations in silos. Model sparsity innovations tend to work in silos due to lack of a common foundation to build on. Machine learning practitioners often have to implement their sparsity algorithms end-to-end manually [28, 55]. Such individual solutions are hard to be extended to or combined with other proposals. It is also

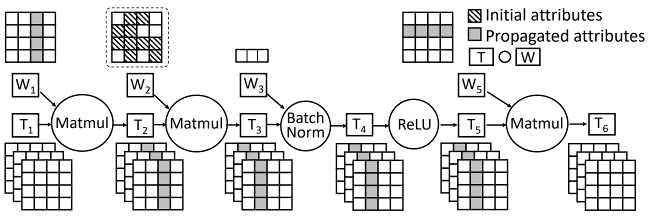


Figure 1. The sparsity attribute of one tensor can be propagated along the deep learning network.

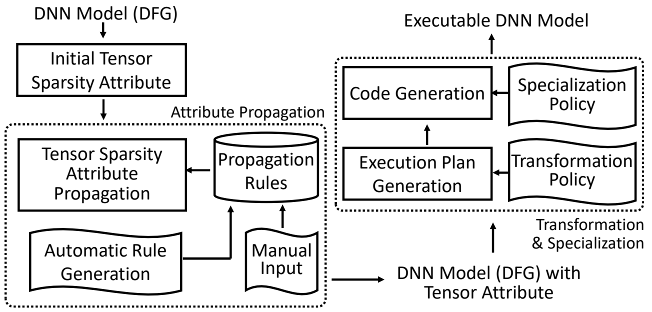

DNN Model (DFG)
Initial Tensor Sparsity Attribute
Attribute Propagation
Tensor Sparsity Attribute Propagation
Propagation Rules
Automatic Rule Generation
Manual Input
DNN Model (DFG) with Tensor Attribute
Executable DNN Model
Code Generation
Specialization Policy
Execution Plan Generation
Transformation Policy
Transformation & Specialization

Figure 2. The system architecture of SparGen.

difficult for them to be further enhanced with new special hardware primitives (e.g., the DP4A feature in NVIDIA GPU that supports mix-precision computation [7]).

3 Design

We design SparGen, a deep learning system that provides full-stack support to model sparsity. Figure 2 summarizes the overall architecture of SparGen. At the core of SparGen is the TeSA abstraction, which augments the existing tensor abstraction with *sparsity attribute* (§3.1). An algorithm designer can specify arbitrary sparsity patterns in any tensor of a deep learning model by setting the sparsity-attribute values. These sparsity attributes set by the algorithm designer serve as the input to SparGen (marked as Initial Tensor Sparsity Attribute in Figure 2).

Given the initial sparsity attribute, SparGen will perform *attribute propagation* to infer the sparsity attributes of all other tensors in the deep learning model, according to the propagation rules defined automatically or by domain specific knowledge (§3.2). Sparsity attribute propagation exposes more optimization opportunities across the model than the original sparse tensor, as shown, for example, in Figure 1.

When the sparsity attribute of all tensors in the DNN model is derived, SparGen will run a multi-pass compilation process to generate efficient end-to-end kernel code. Compared to a traditional DNN compiler, there are two additional compiling passes SparGen conducts to exploit model sparsity fully. The first pass performs execution-plan transformation (§3.3). When generating an execution plan for the DNN model, SparGen may transform the original execution plan into a new one with the given sparsity pattern. For example, it may decouple one tensor into two, each with a different sparsity attribute to use different quantization schemes (e.g., 16-bit vs. 8-bit quantization). Correspondingly, the decoupled tensor will require rewriting the original operator into two new operators, each leveraging different hardware for efficient computation.

With the transformed execution plan, SparGen will further run a compilation pass to perform sparsity-aware code specialization (§3.3). The awareness on the sparsity pattern of any tensor gives SparGen an opportunity to generate highly customized code specifically tailored for the observed sparsity pattern. For example, any pruned element in a tensor could lead to dead code elimination for the corresponding part of computation involving the particular element.

Finally, with the generated end-to-end DNN code, the sparsity algorithm designer is able to profile the DNN model to obtain the trustworthy performance feedback, including memory consumption and inference latency for the whole or a particular part of the DNN model. Given the feedback, the algorithm designer may further update the sparsity attribute in some tensors and repeat this process iteratively. Thus SparGen provides a feedback loop to the end user so as to facilitate the innovation of model sparsity.

In the rest of this section, we will elaborate TeSA and introduce each component in SparGen in more details.

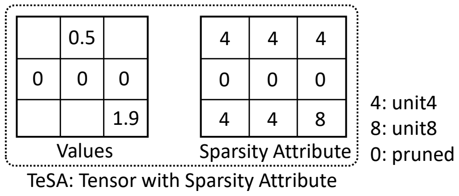


Figure 3. An example of TeSA abstraction. Sparsity Attribute denotes the quantization scheme, 4 means `uint4`, 8 means `uint8`, and 0 means the element is pruned.

3.1 The TeSA abstraction

TeSA is a simple yet powerful abstraction. Figure 3 shows an example of TeSA. In addition to the traditional tensor, TeSA further provides Sparsity Attribute, which is an additional tensor where each element represents the sparsity attribute of the corresponding element in the original tensor. This allows a user to specify an arbitrary sparsity pattern in a tensor. For example, one can use 8-bit to represent the bottom-right element and prune the second row in the tensor. Algorithm designer is required to specify at least some sparsity attribute of some TeSAs before running the system. Therefore, it is possible for SparGen to understand the sparsity pattern at compile time, which enables further optimizations.

3.2 Sparsity Attribute Propagation

Since the number of tensors in a deep learning model is usually large, a user may only set the values of the sparsity attribute for a subset of tensors. To maximize model sparsity, SparGen will perform attribute propagation to derive the sparsity attribute of all tensors in the DNN model.

In SparGen, the attribute propagates bidirectionally. Given an initial sparsity attribute, it may propagate both downstream and up-stream along the data flow graph of the DNN model, in the granularity of operators. Taking the operator `Matmul` (matrix multiplication) in Figure 4 as an example. In Figure 4(a), the sparsity attribute shows that tensor W_2's third row and second column are pruned. With attribute propagation, the second column of the downstream output tensor W_3 has been pruned. Meanwhile, one column in W_1 is pruned, due to W_2's sparsity attribute. Figure 4(b) shows that the sparsity attribute of the downstream tensor W_3 can be propagated back to the upstream tensor W_2. Eventually, a sparsity attribute in a tensor may impact the whole deep learning model. As will be shown in §3.3, such sparsity attribute can be used to optimize code in the compilation stage. For example, the entire codes that compute the second column of W_2 in Figure 4 can be removed (dead code elimination).

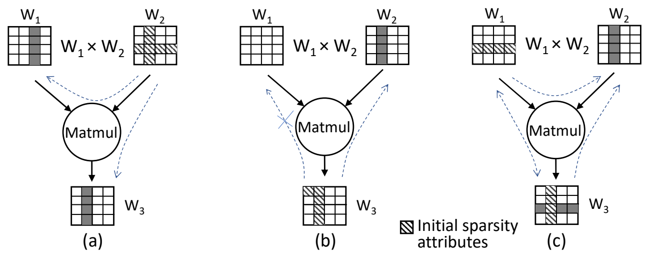


Figure 4. The propagation of sparsity attribute. The gray blocks are propagated sparsity attributes.

Propagation rules. How a sparsity attribute is propagated could be operator specific. For example, in Figure 4(b), the sparsity attribute of a pruned element [0,0] in tensor W_3 cannot propagate to W_1 and W_2 through the operator `Matmul`, while it does propagate through element wise operators like `ReLU` and `BatchNorm`.

The propagation rule could also be algorithm specific. For example, empirically, an algorithm designer could specify that the length of the quantization bit of an output tensor should not be larger than twice of that of its input tensor.

SparGen allows user to register new propagation rules through the interface `register_rule` defined in Figure 5. And Figure 5 also shows an example propagation rule defined for the operator `ReLU`. In this rule, the input `tensor` is of the type TeSA. No matter whether `tensor` is input or output, the propagation rule is the same; i.e., the sparsity attribute passes through without any change.

```
# interface of propagation rules
def register_rule(op_type, attr_prop_func)

#  propagation rule for ReLU operator
def prop_relu(tensor: TeSA):
    attr_tensor = TeSA()
    for ele in tensor:
        attr_tensor.attr[ele.loc] = ele.attr
    return attr_tensor
register_rule('ReLU', prop_relu)
```

Figure 5. Propagation-rule interface: the ReLU example.

Attribute propagation may result in conflicts. For example, in Figure 4(c), the sparsity attributes in W_1 and W_3 are set with conflicting initial values. Such conflicts are detected during the propagation and resolved through conflict resolution rules. Currently, for pruned elements, the resolved tensor is the union of all the pruned elements. For quantization, SparGen chooses the lower precision. Like propagation rules, the conflict resolution rule can also be customized.

Automatic rule generation for attribute propagation. The propagation rules of some operators could be more complex than `ReLU` (e.g., `Matmul`). It is a burden to define propagation rules for every operator. To alleviate this issue, SparGen adopts *Tensor Scrambling* to generate propagation rules automatically for all operators where the associated tensors contain pruned elements.

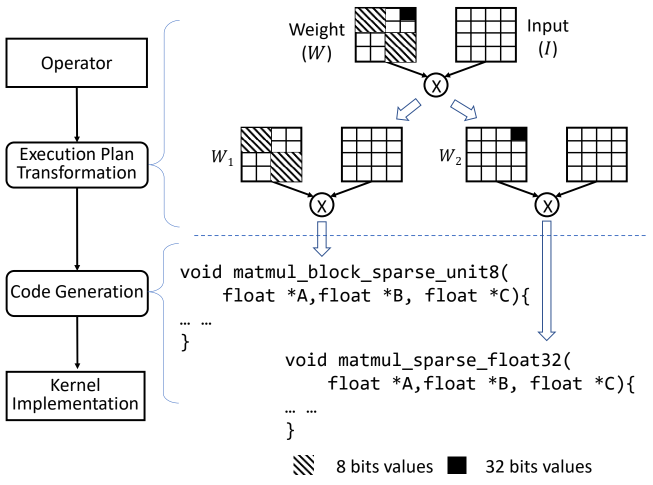
Essentially, *Tensor Scrambling* detects the invariant elements of a tensor by scrambling the values of other related tensors. Specifically, if an input tensor of an operator contains pruned elements (denoted by its sparsity attribute), SparGen infers the sparsity attribute of the output tensor as follows. It assigns random values to the input tensor while keeping the value of pruned elements to 0, and runs the operator to obtain its output tensor (assuming at least the dense version of the operator is available). By repeating this process sufficiently large amount of times, SparGen detects the invariant elements across all the output tensors (i.e., 0), these elements are then pruned by marking the corresponding sparsity attribute of the output tensor.

The approach above could derive the sparsity attribute propagation rule from input to output tensor. As for propagation rule from one input to another input, and from output to input, SparGen leverages the auto differentiation (AD) of DNN models to infer the rule. An operator usually has its counterpart backward operator (for back-propagation in the AD) . Let $I_1 ... I_n$ and $O_1 ... O_n$ to denote an operator's inputs and outputs respectively. Its backward operator's inputs are $I_1 ... I_n$, $gO_1 ... gO_n$, and outputs are $gI_1 ... gI_n$. The prefix g means the corresponding tensor is a gradient. To infer the sparsity attribute from I_2 to I_1, SparGen again uses tensor scrambling. It assigns random values to all the inputs of the backward operator, and the pruned elements in I_2 are set to 0. Then, the system detects invariant elements in gI_1, which are I_1's sparsity attribute. Similarly, for inferring sparsity from O_1 to I_1, we assign the pruned elements in gO_1 to 0 (the sparsity of O_1 and gO_1 are the same) while setting the value of other elements randomly. The invariant (zero-value) elements in gI_1 are the pruned one of I_1.

With the propagation of sparsity attribute, SparGen can derive certain sparsity patterns across the whole deep learning model. Next, we show how such sparsity information enables SparGen to perform sparsity-aware optimizations.

3.3 Generating Efficient TeSA Code

The sparsity pattern of a tensor could be complex [28, 38, 53, 62] with a mixture of different sparsity patterns, making it challenging to generate an efficient, customized operator kernel for a tensor with a specific sparsity pattern. SparGen therefore first transforms the tensors with a complex sparsity pattern to the combination of simpler sparsity patterns, and then generates an efficient kernel for each simplified sparsity pattern. Correspondingly, the execution plan of the operator needs to be rewritten to accommodate the new operators to compute the newly transformed tensors. Finally, SparGen performs code generation for the transformed execution plan, with sparsity-aware specialization. The overall two-pass compilation process is shown in Figure 6.



```
void matmul_block_sparse_unit8(
    float *A,float *B, float *C){
… …
}
        void matmul_sparse_float32(
            float *A,float *B, float *C){
        … …
        }
```

Figure 6. The two-pass compilation process to generate an efficient kernel implementation for an operator.

Execution-plan transformation. In the first pass, SparGen leverages a traditional deep learning compiler (*e.g.,* TVM [18]) to generate an execution plan that is sparsity unaware. SparGen then inspects the sparsity attribute on each given tensor. If the sparsity pattern is "irregular", SparGen transforms the tensor into the sum of multiple tensors, each with a simpler, more regular sparsity pattern. For example, the weight tensor W in Figure 6 is a mix-precision tensor, where two coarse-grained blocks use 8-bit quantization and one fine-grained element uses 32-bit. SparGen transforms W into W_1 and W_2, each using its own quantization scheme. Consequently, SparGen introduces two operators to handle $W_1 \times I$ and $W_2 \times I$, respectively, each with the proper hardware instruction leveraging the corresponding quantization scheme. As a result, the original execution plan with one tensor operation has been transformed into a new one, where two more tensor operations are required.

SparGen supports different policies for execution-plan transformation. It also allows policy customization. Figure 7 illustrates three example policies. In the figure, the sparsity attribute in each element of a tensor can be 32-bit, 8-bit, or zero (pruned). The left tensor contains only one 32-bit element and the rest are all 8-bit. In this case, SparGen transforms it into two tensors, one is an 8-bit dense tensor (except that the value of one element is always zero), the other is a 32-bit sparse tensor. SparGen can therefore use different code specialization policies to handle the dense and sparse tensor differently (e.g., less loop unrolling for the dense version). SparGen can also compute the 8-bit tensor with the 8-bit hardware instruction if available.

The middle tensor in Figure 7 has a mixture of block sparsity and fine-grained sparsity patterns. Similarly, the tensor is decomposed to two tensors for the code generation pass to leverage block sparsity and fine-grained sparsity respectively.

Finally, the right tensor in Figure 7 assimilates the "minority" elements to construct a regular sparsity pattern. In this case, although the top element of the tensor is pruned, its sparsity attribute is changed back to 32-bit quantization so as to construct a more regular block sparsity pattern.

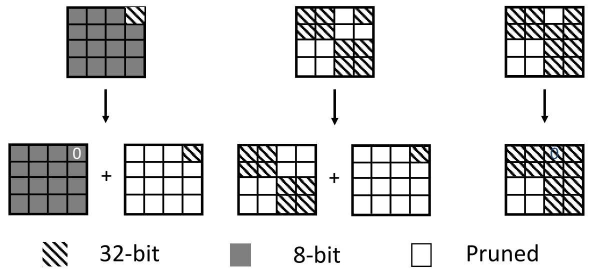


Figure 7. Examples of transformation policies.

In practice, given a sparsity pattern, SparGen may use different transformation policies to generate multiple execution plans, perform code generation, and select the best result. Figure 8 shows an example to transform the execution plan on matrix multiplication (`matmul_transform` in line 11). To generate a good execution plan for the matrix multiplication of two tensors A and B ($A \times B$), for each tensor A and B, SparGen first generates several candidate transformation schemas using `tensor_transform`. This function first decomposes a tensor into several tensors, each of which has only one quantization precision (*i.e.,* `bit_decompose` in line

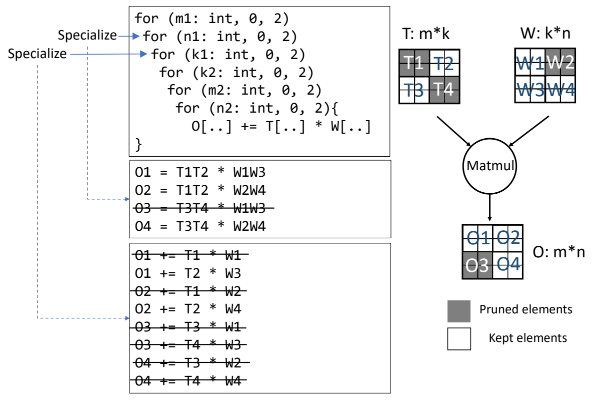
3), aligned with the underlying hardware-supported low-precision instructions. It further decomposes each resulting tensor by matching block sparsity and fine-grained sparsity (*i.e.,* `sparsity_decompose` in line 6). `tensor_transform` returns a set of schemas, where the sum of the tensors in each schema is the input tensor (A or B). The matrix multiplication $A \times B$ can then be transformed to $A_1 \times B_1 + A_1 \times B_2 + A_2 \times B_1 + A_2 \times B_2$, if $A's$ schema is $A_1 + A_2$ and B's is $B_1 + B_2$. By comparing the results of different schemas, the best plan is decided and returned.

```
1  def tensor_transform(T, bits):
2      trans_schemas = []
3      bit_schemas = bit_decompose(T, bits)
4      for schema in bit_schemas:
5          for tensor in schema:
6              sparse_schema = sparsity_decompose(tensor)
7              schema.update(tensor, sparse_schema)
8          trans_schemas.append(schema)
9      return trans_schemas
10
11 def matmul_transform(A, B):
12     specialized_impl = null
13     best_latency = MIN
14     ta_schemas = tensor_transform(A)
15     tb_schemas = tensor_transform(B)
16     for a in ta_schemas:
17         for b in tb_schemas:
18             implement = specialize_matmul(a, b)
19             latency = profile(implement)
20             if latency < best_latency:
21                 specialized_impl = implement
22     return best_latency, specialized_impl
```

Figure 8. An example transformation policy for matrix multiplication.

TeSA code specialization. With the transformed execution plan and the sparsity attributes, SparGen can now understand and leverage the sparsity patterns in a DNN model. Specifically, SparGen uses the sparsity attributes to specialize the code during code generation and performs sparsity-aware optimizations to generate efficient, custom kernel code for a given sparsity pattern. With the TeSA abstraction, SparGen knows exactly which element's computation can be eliminated (for zero values) or which tensor operation might benefit from special hardware instructions (*e.g.,* DP4A for 8-bit computation), as specified in *specialization policies*.

SparGen implements *sparse_specialize*, a specialization policy for efficient kernel code generation for sparsity-aware DNN operators. It could generally specialize various sparsity patterns produced by both pruning and quantization into efficient kernel implementations. *sparse_specialize* assumes that a deep learning operator (*e.g.,* Matmul and Conv2d) is implemented with multi-level tiling loops [18, 67] following an efficient tiling strategy, which can be derived by a traditional DNN compiler (unaware of the sparsity pattern). An example tiling loops of Matmul is shown in Figure 9, where there are six levels of loops, each axis is tiled into two axes (*e.g.,* m to $m1$ and $m2$). With TeSA, SparGen is aware of the pruned elements and low precision elements. *sparse_specialize* then specializes pruned elements by unrolling loops and applying dead code elimination (DCE). It specializes low precision elements by replacing loops with hardware instructions.



```
for (m1: int, 0, 2)
 for (n1: int, 0, 2)
  for (k1: int, 0, 2)
   for (k2: int, 0, 2)
    for (m2: int, 0, 2)
     for (n2: int, 0, 2){
       O[..] += T[..] * W[..]
}
```

O1 = T1T2 * W1W3
O2 = T1T2 * W2W4
~~O3 = T3T4 * W1W3~~
O4 = T3T4 * W2W4

~~O1 += T1 * W1~~
O1 += T2 * W3
~~O2 += T1 * W2~~
O2 += T2 * W4
~~O3 += T3 * W1~~
~~O3 += T4 * W3~~
~~O4 += T3 * W2~~
~~O4 += T4 * W4~~

Figure 9. Sparsity-aware code specialization: loop unrolling and dead code elimination.

The idea of specializing pruned elements is that, as the multi-level loops tile the computation into different sized computation blocks hierarchically, DCE can be applied to different levels of computation blocks (loops). For the example in Figure 9, if the top two loops are unrolled, there are four computation blocks, among which the third computation block can be eliminated as $O3$ in the output tensor is pruned. If the top three loops are unrolled, there are eight smaller computation blocks. As it matches pruning granularity, DCE can be more efficiently applied. Specifically, six computation blocks are eliminated, based on the pruned elements in both input and output tensors, denoted by sparsity attribute. Note that specializing pruned elements is complementary with existing kernel implementations, as the computation block after kernel specialization can still be implemented with those implementations. For example, the computation block $O1 = T1T2 * W1W3$ can be implemented with dense matrix multiplication or CSR based sparse matrix multiplication. As seen in the example, there exist multiple choices to specialize a kernel code. User is allowed to implement custom specialization policy to evaluate different options.

Unlike specializing pruned elements, which is applied from the top loop down, specializing low-precision elements works in a bottom-up manner. Based on the sparsity attribute, the system picks the corresponding hardware instruction and applies the instruction starting from the innermost loop. For example, if the instruction is a simple element multiplication, it will replace the code line within the innermost loop. If the instruction is multiplication and sum of four elements (*e.g.,* DP4A), it will replace the innermost loop. If the instruction is a small matrix multiplication (*e.g.,* mma_sync of Tensor Cores), it will replace the two innermost loops.

The kernel code specialization is also affected by the operator-tiling strategy (*i.e.,* the term *schedule* used in TVM). For example, if the tiling is not aligned with the block size of block sparsity, code specialization may become sub-optimal. Similarly, if the tiling does not align the innermost loops with the computation granularity of hardware instruction, specializing low-precision element would not work. In practice, SparGen treats the joint tiling and kernel code specialization as a search problem; *i.e.,* using a certain search policy to find a reasonable solution.

4 Implementation

We implemented SparGen using DenseGen [41], an open-source, light-weight, and flexible DNN compiler that generates high-performance executable. Specifically, SparGen takes a TeSA-based deep learning model, propagates the sparsity attribute across the model, and automatically generates kernels for sparse operators. SparGen feeds a DNN model graph in the ONNX [8] format to DenseGen, and adds an additional compiler optimization pass for execution-plan transformation, which induces necessary graph rewrite, and injects the generated kernels into DenseGen's kernel db. After DenseGen completes all the optimization passes, including common compiler optimizations, such as kernel fusion and memory planning, an executable binary is produced for model inference.

Kernel code specialization is implemented following the design of kernel *schedule* in TVM. SparGen abstracts the kernel specialization into two new schedule primitives: `dismantle` and `comp_unit`. Primitive `dismantle` is for dead code elimination, enabled by the sparsity-aware specialization. It is applied to a certain level of loop (*i.e.,* axis). The usage of `dismantle` is similar to the `split` primitive in a TVM schedule. It means that loop and all the loops out of that loop should be unrolled, where the dead computation blocks indicated by the sparsity attribute are eliminated accordingly. Loop unrolling sometimes introduces a large amount of kernel code. This problem can be mitigated if degree of sparsity is high. Another way to reduce kernel code is to order loops (*i.e.,* axes) properly. For example, for sparse-dense matrix multiplication, setting the sparse tensor's axes as outer loops, the sparsity-aware specialization can be done by unrolling only those outer loops. `comp_unit` is for specializing hardware instructions in kernel implementation. It is similar to `Tensorize` in TVM. We wrap hardware instructions, which reorganize data if necessary, as a basic computation block, and rely on a tiling strategy to align a certain level of tiling blocks to such basic blocks.

SparGen has supported 19 model sparsity algorithms, including both pruning and quantization. Those algorithms can run on SparGen with little code modifications, and benefit from SparGen not only on sparsity exploration but also on model fine tuning, which will be demonstrated in §5.3.

5 Evaluation

We conduct extensive experiments to evaluate SparGen under various DNN sparsity patterns using NVIDIA and AMD GPUs, and compare SparGen's performance against the state-of-the-art DNN inference engines like TVM [18], a general purpose DNN compiler, and TensorRT [10], a highly optimized vendor-specific DNN library. The key findings from the experiments are the following:

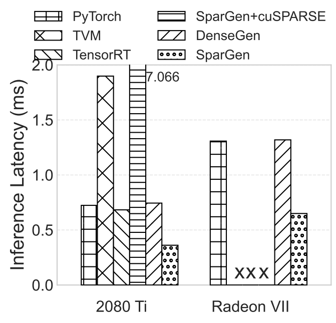
- SparGen can significantly improve the inference performance for TeSA-augmented DNN models, achieving up to 8.7x speedup compared to TVM and up to 3.7x speedup compared to TensorRT.
- SparGen can generate high-performance kernel implementations that efficiently exploit diverse sparsity patterns expressed by TeSA, including fine-grained and coarse-grained sparsity, different precision, mixed sparsity and precision.
- SparGen greatly facilitates the development and exploration on modern model sparsity algorithms, helps producing DNN models with lower inference latency and/or higher accuracy.

5.1 Speeding up TeSA-augmented DNN Models

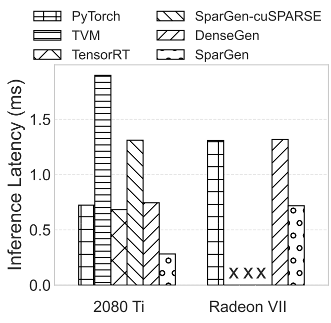
We first evaluate how SparGen can reduce inference latency for sparse DNN models. We apply the following four sparsity patterns to three popular DNN models, Multi-Layer Perceptrons (MLP) [43], MobileNet [32], BERT [22], respectively.

- Coarse-grained sparsity: Model weights are randomly pruned in the granularity of column, row, and channel [30, 37, 40] to reach 60% model sparsity.
- Fine-grained sparsity: Model weights are randomly pruned in the granularity of elements [29, 35, 36] to reach 95% model sparsity.
- Coarse-grained sparsity + 8-bit quantization: Applying 8-bit quantization to a coarse-grained sparse model with 60% model sparsity [56].
- Block sparsity + 8-bit quantization: Applying 8-bit quantization to a model being pruned with block granularity (60% sparsity) [57].

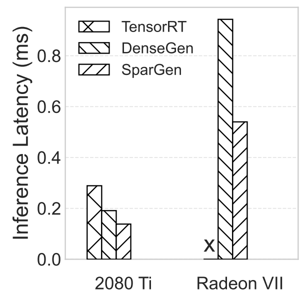
We compare SparGen to three popular deep learning tools: PyTorch (v1.7) [48] (we use its JIT version), TVM (v0.8) [18], and TensorRT (v7.2) [10]. For TVM, each kernel is tuned with 1,000 trials. To evaluate the end-to-end performance of cuSPARSE, NVIDIA's sparse kernel library [2], we construct a baseline SparGen-cuSPARSE, which replaces SparGen's own kernel with cuSPARSE. To understand the performance gain from SparGen, we introduce another baseline DenseGen, which uses the same code base as SparGen, but uses the dense kernels instead of the custom sparse kernels. We run the experiments on two different GPUs: Nvidia GeForce RTX 2080 Ti [4] and AMD Radeon VII [11] to show that SparGen can be easily extended to different accelerators. Note that TVM provides a sparse kernel implementation for matrix



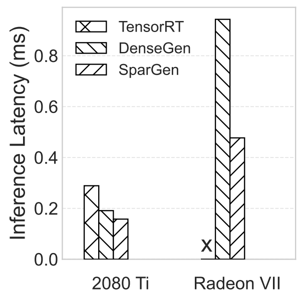
(a) Coarse-grained sparsity



(b) Fine-grained sparsity

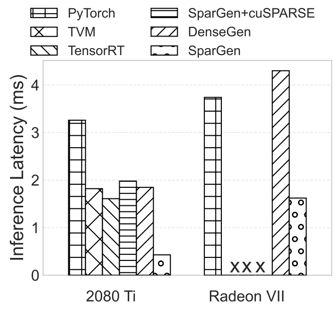


(c) Coarse-grained sparsity + 8-bit

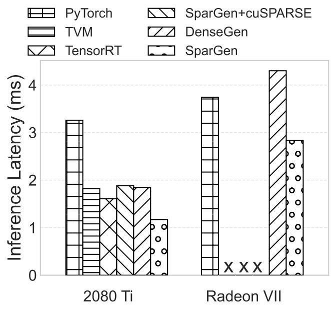


(d) Block sparsity + 8-bit

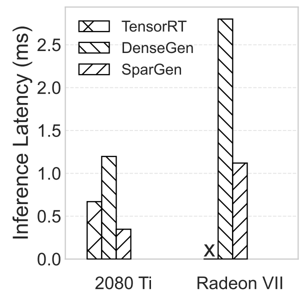
Figure 10. The end-to-end inference latency of MLP with four different sparsity patterns.



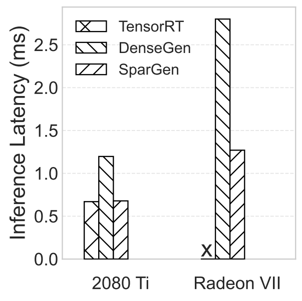
(a) Coarse-grained sparsity



(b) Fine-grained sparsity

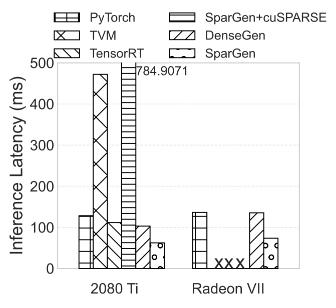


(c) Coarse-grained sparsity + 8-bit

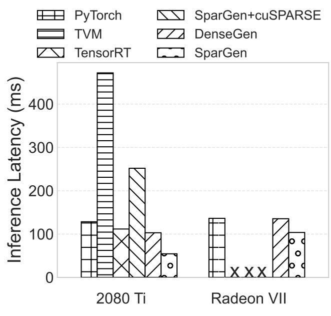


(d) Block sparsity + 8-bit

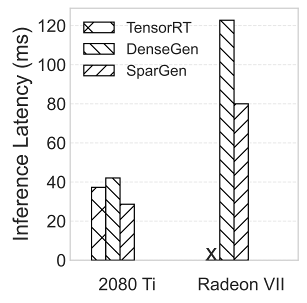
Figure 11. The end-to-end inference latency of MobileNet with four different sparsity patterns.



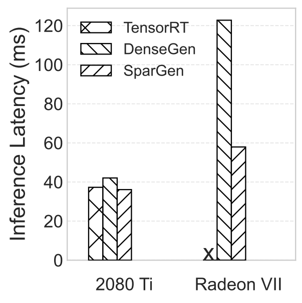
(a) Coarse-grained sparsity



(b) Fine-grained sparsity



(c) Coarse-grained sparsity + 8-bit



(d) Block sparsity + 8-bit

Figure 12. The end-to-end inference latency of BERT with four different sparsity patterns.

multiplication [12]. However, the kernel always crashes in our experiments. We fix the implementation and find that the resulting kernel performs worse than cuSPARSE (by 100%). Hence we decide not to include the result in this paper. Note that TVM does not yet provide a sparse library to support general sparse models, not to mention the capability to automatically generate custom kernel implementation for a specific sparsity pattern.

Experimental results. Figure 10 shows the experiment results on MLP. SparGen always performs the best on the four sparsity patterns, achieving up to 2.6x, 6.8x, 2.4x, 2.6x speedup compared to DenseGen, TVM, TensorRT, and PyTorch respectively. DenseGen incurs a latency similar to that of TensorRT and PyTorch for coarse-grained and fine-grained sparsity. This shows the performance gain of SparGen comes from the sparsity-aware optimizations, enabled by TeSA. TVM performs the worst for coarse-grained sparsity because the kernels it tuned performs worse than that in PyTorch and TensorRT. SparGen-cuSPARSE performs the worst for fine-grained sparsity, because the general sparse kernel in cuSPARSE incurs higher latency than its dense counterpart.

For mixed sparsity (*i.e.,* coarse-grained + 8-bit, block sparsity + 8-bit), SparGen performs the best because it leverages both sparsity and hardware instruction; *i.e.,* Tensor Cores `mma_sync` in 2080 Ti and `amd_mixed_dot` in Radeon VII. We do not show other baselines in Figure 10c and Figure 10d, because they do not support 8-bit on GPU (or cannot run successfully). Falling back to 32-bit kernels would make it perform much worse than any bar in these two figures. DenseGen also performs better than TensorRT, because our automatically tuned kernel fits well on the tensor shape and the hardware (*i.e.,* fully occupying an SM on GPU), even without sparsity-aware optimizations.

The numbers on Radeon VII are similar, except that some baselines are cannot run successfully on Radeon VII, which also shows SparGen's flexibility on generating optimized code on different accelerators.

Figure 11 shows that SparGen also performs the best on MobileNet, achieving up to 4.3x, 4.2x, 3.7x, and 7.8x speedup compared to DenseGen, TVM, TensorRT, and PyTorch respectively. For coarse-grained sparsity (Figure 11(a)), each layer in MobileNet (*i.e.,* Conv1x1, DepthwiseConv, and Linear) is applied with 60% sparsity; while the speedup of SparGen against DenseGen is higher than the sparsity ratio, *i.e.,* 4.3x. This is because the sparsity attribute propagation further increases each layer's sparsity, achieving a global model sparsity around 87%. PyTorch performs the worst. Because MobileNet is a small model, which amplifies the runtime overhead of PyTorch framework. For fine-grained sparsity, SparGen-cuSPARSE has similar performance to DenseGen, because MobileNet has only one Linear layer that is able to use cuSPARSE (which does not support other operators like conv or depthwise conv). For mixed sparsity, DenseGen shows a higher latency than TensorRT, because TensorRT fully optimizes MobileNet on 8-bit, while DenseGen generated the 8-bit kernels on Tensor Cores on its own. SparGen outperforms TensorRT by up to 1.9x, due to the sparsity-aware optimizations, including specialization to use low-precision instructions. On Radeon VII, SparGen consistently outperforms DenseGen by up to 3.8x.

Figure 12 shows the results on BERT. Similar to MLP, BERT model's main building operators are matrix multiplication. Thus, the trend of the results is similar to that of MLP. The speedup of SparGen is up to 1.9x, 8.7x, 2.0x, and 2.4x compared to DenseGen, TVM, TensorRT, and PyTorch, respectively.

Performance breakdown. To understand where the performance gain of comes from, we break down the performance number by applying the optimization techniques in SparGen one by one; *i.e.,* specialization for hardware instruction, specialization for sparsity (e.g., DCE), and sparsity attribute propagation. Figure 13 shows the performance breakdown. For coarse-grained sparsity + 8-bit, the latency decreases when each optimization technique is applied, the resulting reduction is 35%, 34%, and 13%, respectively. Block sparsity is more difficult to optimize than coarse-grained sparsity, thus latency reduction of block sparsity + 8-bit is less than that of coarse-grained sparsity + 8-bit. For fine-grained sparsity, the first two bars are the same, because it uses `float32` instead of `int8`. Specialization and sparsity attribute propagation bring 22% and 14% latency reduction, respectively. The breakdown on Radeon VII shows a similar trend, hence being omitted.

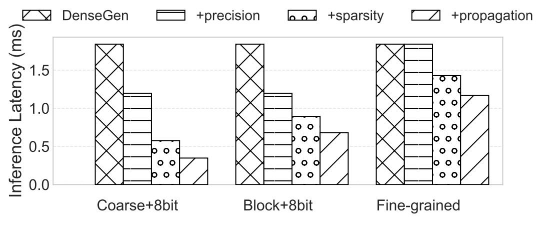


Figure 13. Performance breakdown of SparGen for different sparsity patterns of MobileNet on 2080 Ti. Each bar shows the result of applying the additional optimization labeled on this bar from the previous one.

5.2 Micro-benchmarks

We use micro-benchmark to illustrate the details of code specialization and sparsity attribute propagation in SparGen.

Effectiveness of TeSA code specialization. We evaluate SparGen's specialized matrix multiplication kernel under different fine-grained sparsity ratio, ranging from 50% to 99%. We compare the specialized kernels with cuSPARSE and TACO [9, 34], Sparse GPU kernels [26], SparseRT [58]. The result is shown in Figure 14. 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, though it has a lower memory footprint due to encoded sparse tensors. TACO performs worse than cuSPARSE, which is consistent with the numbers reported in their paper [34]. It is 15.6x slower than cuSPARSE for 99% sparsity; the slowdown is reduced to 4.0x when the sparsity is 50%. SparGen is up to 6.01x faster than cuSPARSE. It outperforms cuBLAS when the sparsity is only 70%.

Effectiveness of execution plan transformation. The sparsity-aware execution plan transformation in SparGen is powerful in exploiting complex sparsity patterns. We construct two types of sparsity patterns: (1) Mixed precision, where a tensor has both `int8` elements and `float32` elements [62], the ratio of `float32` elements is varied from 5% to 0%. (2) Mix of block sparsity and fine-grained sparsity [31, 34], there are 1% fine-grained elements, and the block sparsity ratio varies from 70% to 90%. Figure 15a shows that decomposing the sparse tensor into one 8-bit tensor and one

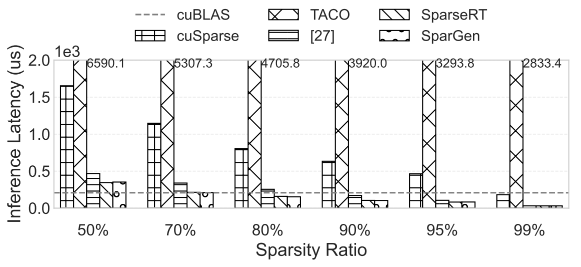
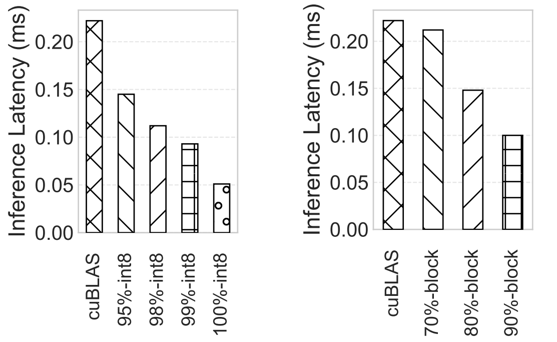


Figure 14. Comparison of cuSPARSE, TACO, and SparGen on matrix multiplication (1024x1024x1024) with fine-grained sparsity under different sparsity ratios. The sparsity is applied on B for $A * B$.

32-bit tensor could speed up the matrix multiplication, as the 8-bit tensor leverages Tensor Cores. The higher the sparsity ratio of `float32` elements is, the faster the operator could be. Figure 15b shows that decomposing the tensor into the sum of one block sparse tensor and one fine-grained sparse tensor could outperform the dense implementation. Note the system can also treat the tensor as a pure fine-grained sparsity pattern (i.e., assimilating), where the transformation does not happen. As stated in §3.3, SparGen may generate multiple execution plans and select the best one based on evaluation results.



(a) Mixed precisions **(b)** Mixed sparsity patterns

Figure 15. The performance of the execution plan transformation in SparGen for mixed precision and sparsity patterns. B is sparsified for the matrix multiplication $A * B$ (1024x1024x1024). "X%-block" means X% block sparsity mixed with 1% fine-grained sparsity.

Effectiveness on sparsity attribute propagation. We also study how sparsity attribute propagation in SparGen works on an end-to-end, real-world deep learning model, under different sparsity patterns.

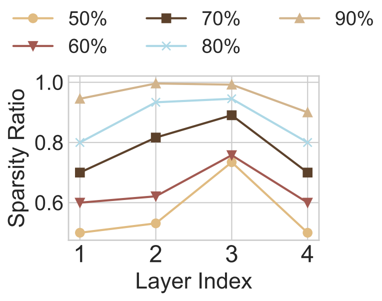
In the experiment, we set the four layers in MLP with the same sparsity ratio and varies the sparsity ratio from 50% to 98%. Also, we experiment with three types of sparsity pattern: block sparsity, fine-grained sparsity, and coarse-grained sparsity. The sparsity is applied to model weights. Figure 16 shows the experiment results on the MLP model. When randomly setting the layers with block sparsity (Figure 16a), the two middle layers have a higher sparsity ratio than the pre-configured one due to sparsity attribute propagation. The sparsity of layers 1 and 4 increases only a little because they only accept propagated sparsity attributes from one direction. Fine-grained sparsity (in Figure 16b), on the other hand, has a weaker propagation ability. The probability that an entire column or row in weight tensor is pruned is much lower than block sparsity. Even so, 3.8% of the sparse attribute is still propagated to layer 3. The propagation ability of coarse-grained sparsity, as shown in Figure 16c, is the best, because coarse-grain sparsity prunes tensors in the granularity of rows and columns. In this experiment, we prune only rows (no columns) from model weights, the sparsity propagates in only one direction, *i.e.,* forward propagation. Thus, the latter three layers have much higher sparsity ratio than the pre-configured ratio: the ratio is increased by up to 50%.

Due to page limit, we omit detailed results for the other two models and only briefly describe their high-level result. Since BERT's model architecture is similar to MLP, the result is also similar. MobileNet has a higher chance to propagate a sparsity attribute across the model (as indicated in Figure 11a). Thus the propagation increases the overall sparsity from 60% to 87%.

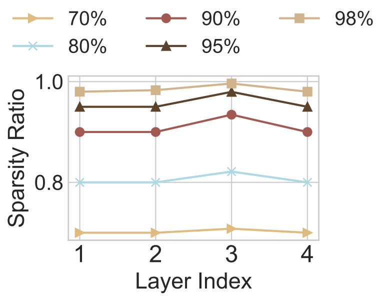
Sparsity attribute propagation is not limited to pruned elements (*i.e.,* zero values). We use a micro-benchmark to illustrate how quantization, another type of sparsity, can be propagated. Specifically, we first run the Simulated Annealing algorithm [39], which is a model compression algorithm that searches the sparsity ratio of each layer. We run the SA algorithm for 20 iterations to find each layer's best quantization setting (*i.e.,* to be either 8 bits or 32 bits), which we call "Before-propagation". Then we customize a propagation rule for quantization, following the paper [38]. Specifically, given a layer's quantization bit width b and the number of elements in this layer's weight w, its neighbor layer's bit width is $b_n = bw/w_n$, where w_n is the number of elements in the neighbor layer's weight. If b_n is larger than 16, which is an empirical threshold, the neighbor layer's bit width is set to 32; otherwise, it is set to 8. The experiment result is shown in Figure 17. After propagation, many layer's bit-width is decreased from 32 to 8. Accordingly, inference latency decreases from 0.85ms to 0.70ms. The accuracy of the two models are the same, *i.e.,* 93.02% on Cifar10.

5.3 Facilitating Exploration of Model Sparsity

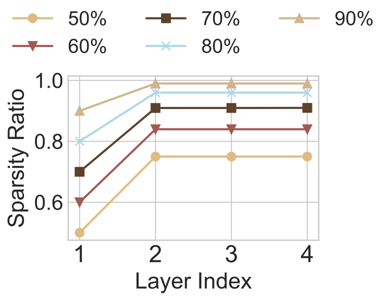
SparGen, as a full-stack solution for model sparsity, facilitates the exploration of existing model sparsity algorithms. In this section, we demonstrate this from following three aspects.



(a) Block sparsity



(b) Fine-grained sparsity



(c) Coarse-grained sparsity

Figure 16. Propagated sparsity across the layers for different sparsity patterns on the MLP model.

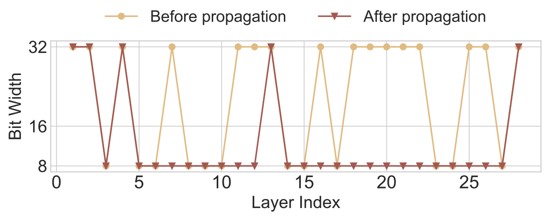


Figure 17. The quantization (bit width) of each layer in MobileNet before and after propagation. They have the same accuracy.

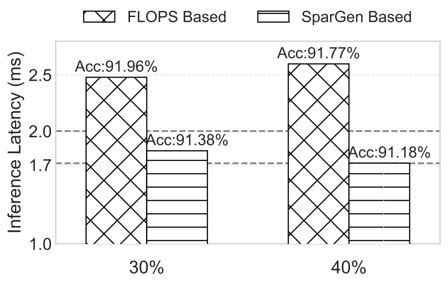


Figure 18. The comparison of using real latency or FLOPS as metric to explore sparse models by Simulated Annealing.

Real latency facilitates sparsity exploration. In this experiment, we use Simulated Annealing to prune MobileNet to reduce 30% and 40% inference latency respectively. Our baseline uses FLOPS as the metric to filter out disqualified models: the model whose FLOPS is larger than 70% of the original FLOPS. In contrast, SparGen uses the real latency to filter models. The result is shown in Figure 18. The best sparse models found by the two approaches have similar accuracy. However, the model found by FLOPS does not meet the latency target, with 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 SparGen are guaranteed to meet the latency requirement.

Propagation aware sparsity exploration. Sparsity attribute propagation can discover the sparsity correlation between different DNN layers. Such discovery sometimes shows that two layers should be jointly pruned in order to obtain the best trade-off between accuracy and inference latency. For example, if two `Conv2d` operators' outputs are summed together and fed into another `Conv2d`, and the output channels of the first two `Conv2d` are pruned with the same sparsity pattern, which will be propagated to the last `Conv2d`. However, if the pruned channels are interleaved, sparsity cannot be propagated (Dictated by `Conv2d`'s propagation rule). With sparsity attribute propagation, SparGen exposes all the layers involved in the propagation as a *layer group*. When two layer groups have a common aggregation layer (*i.e.,* `add`, `concat`), the two layer-groups should be jointly pruned to make sure the sparsity can pass through the aggregation layer to the most extent. We evaluate AutoCompress [39] algorithm on pruning MobileNet and ResNet18 in two modes: (1) being aware of layer groups, (2) pruning layers independently. The result in Figure 19 shows that, given the same inference latency, the former obtains higher accuracy. The accuracy gap becomes larger under a more strict latency requirement.

Speeding up sparsity exploration. With high-performance sparse kernel implementation, SparGen can speed up the exploration process of a sparsity algorithm, which usually searches for a sparsity pattern iteratively [39, 63]. 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 SparGen, the model can be accelerated before fine-tuning. Figure 20 runs Simulated Annealing algorithm, which

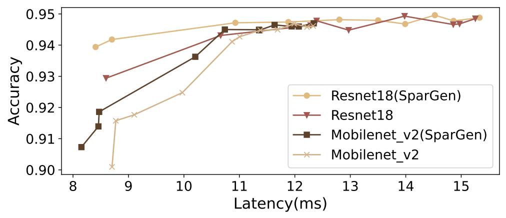


Figure 19. The performance comparison of being aware of sparsity propagation and not being aware.

is an iterative sparsity algorithm, on ResNet50. the algorithm prunes 50% of the remaining weights and fine-tune 300 epochs in each iteration. SparGen reduces 31.8% exploration time, compared to the baseline that always uses the original dense model.

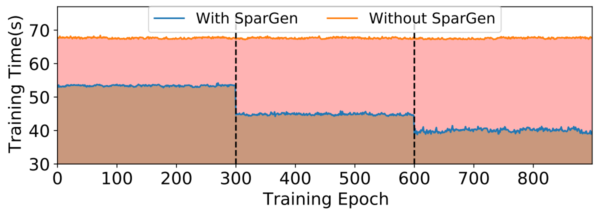


Figure 20. The exploration time when using SparGen-accelerated sparse model vs. not using the accelerated model.

6 Related Works

Sparsity support in DNN frameworks and compilers. The mainstream deep learning frameworks and compilers, including PyTorch [48], TensorFlow [13], TVM/Ansor [18, 67], all treat model sparsity as an afterthought. They either rely on vendor-specific libraries like cuSPARSE to provide sparse kernel implementations [2] or user-provided sparsity kernel templates [18]. They do not understand the specific sparsity pattern of a given sparse model. In contrast, SparGen treats sparsity as a first-class citizen. It proposes TeSA, a new tensor abstraction to express arbitrary sparsity patterns and enable various sparsity-aware optimization to generate efficient sparse kernels.

With TeSA, SparGen is able to incorporate several classic compiler techniques. For example, sparsity attribute propagation is similar to type qualifiers [25] and type inference [21]. OpenMP [23] also leverages attribute propagation in a different problem domain with a different mechanism. Code specialization based on value profiling [16] is also a well-known technique. Zeroploit [50] and PGZ [52] also use a similar idea, but focus on gaming applications. SparGen does not rely on values, but on attributes for code specialization, which is more general. And SparGen offers a complete framework for DNN model sparsity.

Sparsity acceleration of DNN models. Sparse matrix multiplication has been studied for decades in scientific computing [49, 60]. With the emerging accelerators (*e.g.,* GPU [4, 11], TPU [3], FPGA [6], GraphCore [5]), some research optimizes sparse matrix multiplication for a certain type of hardware [14, 15, 26, 60, 70]. 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 [19, 34, 51] generalizes various sparse data formats with an unified expression. It can generate sparse kernel code based on each specific data format. One data format can express any sparsity pattern, but with different efficiency on different sparsity patterns. Unlike TACO, SparGen proposes to specialize sparse kernel code by fully leveraging a specific sparsity pattern and even concrete tensor values. Though it cannot deal with dynamically changed sparsity pattern (or changed sparse tensor values), it pushes the sparsity optimizations to the extreme, while being widely applicable in DNN model inference scenario.

To optimize sparse kernels on GPU, SparseRT [58] proposes to embed sparse weight values into kernel code rather than stored in a sparse data format. It can be seen as a special case of code specialization in SparGen, *i.e.,* unrolling all the loops. Hong et. al [31] reorders elements in a sparse tensor and uses an adaptive tiling strategy to enhance the performance of sparsity matrix multiplication. These optimizations are complementary to SparGen.

Another line of research [17, 64] co-design sparsity algorithm with kernel optimization, which balances sparsity in a tensor for efficient parallel execution on a GPU. Similar design has been incorporated into Nvidia GPU, called Sparse Tensor Core [69]. There are other works that design a new hardware to accelerate sparse operators. EIE [27] designs a new data encoding/decoding node and a new Processing Element (PE) to speed up matrix-vector multiplication. SCNN [46] designs another architecture of PE, which supports sparse convolution in a compressed format. SparGen can easily leverage these new accelerators by adding new transformation and specialization policies.

Sparsity exploration on DNN models. Both the neural science and deep learning communities find lots of clues [35, 65] implying that a deep neural network is (highly) sparse. Meanwhile, in the deep learning community, many model compression algorithms have been proposed to construct sparse models with little accuracy degradation. Unstructured pruning (also called fine-grained pruning) prunes weights without following any specific pattern [29, 35, 36]. The pruned model is hard to be optimized on hardware accelerators (*i.e.,* GPU). To make a pruned model easier to be accelerated, many research works propose structured pruning, which prunes DNN models in a regularized granularity, such as in filter [30], channel [37, 40] in CNN, and block level [42, 44]. Quantization is another way to sparsify a model, such as single-precision quantization [20, 33, 68], mixed-precision among layers [24, 38, 55], and mixed-precision within each tensor [47, 62]. Recent works explore mixing pruning and quantization to make a DNN model even smaller [28, 53, 54, 57, 61, 66]. SparGen is a general framework for model sparsity, its TeSA could express the sparsity patterns in all these papers and generate efficient codes for the sparse model.

7 Conclusion

SparGen takes a principled system approach to model sparsity in deep learning, centered on the new TeSA abstraction. SparGen 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. SparGen 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] 2021. The API Reference guide for cuBLAS, the CUDA Basic Linear Algebra Subroutine library. https://docs.nvidia.com/cuda/cublas/index.html

[2] 2021. The API reference guide for cuSPARSE, the CUDA sparse matrix library. https://docs.nvidia.com/cuda/cusparse/index.html

[3] 2021. Cloud TPU: Train and run machine learning models faster than ever before. https://cloud.google.com/tpu

[4] 2021. GEFORCE RTX 2080 Ti. https://www.nvidia.com/en-us/geforce/graphics-cards/rtx-2080-ti/

[5] 2021. GraphCore. https://www.graphcore.ai/

[6] 2021. Intel FPGAs and Programmable Devices. https://www.intel.com/content/www/us/en/products/programmable.html

[7] 2021. Mixed-Precision Programming with CUDA 8. https://developer.nvidia.com/blog/mixed-precision-programming-cuda-8/

[8] 2021. Open Neural Network Exchange. https://onnx.ai/

[9] 2021. Reproducing OOPSLA 2020 Results. https://github.com/tensor-compiler/taco/tree/oopsla2020

[10] 2021. The SDK for high-performance deep learning inference. https://docs.nvidia.com/deeplearning/tensorrt/

[11] 2021. THE WORLD'S FIRST 7nm GAMING GPU. https://www.amd.com/en/products/graphics/amd-radeon-vii

[12] 2021. TVM sparsity code. https://github.com/apache/tvm/blob/254563a3140cf63fe77a46058688209de3aa213c/python/tvm/topi/cuda/sparse.py#L96

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

[14] Nathan Bell and Michael Garland. 2008. *Efficient sparse matrix-vector multiplication on CUDA*. Technical Report. Citeseer.

[15] Aydin Buluç, Jeremy T Fineman, Matteo Frigo, John R Gilbert, and Charles E Leiserson. 2009. 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*. 233–244.

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

[17] Shijie Cao, Chen Zhang, Zhuliang Yao, Wencong Xiao, Lanshun Nie, Dechen Zhan, Yunxin Liu, Ming Wu, and Lintao Zhang. 2019. 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*. 63–72.

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

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

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

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

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

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

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

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

[26] Trevor Gale, Matei Zaharia, Cliff Young, and Erich Elsen. 2020. Sparse GPU Kernels for Deep Learning. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis, SC 2020*.

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

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

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

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

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

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

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

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

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

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

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

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

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

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

[41] Lingxiao Ma, Zhiqiang Xie, Zhi Yang, Jilong Xue, Youshan Miao, Wei Cui, Wenxiang Hu, Fan Yang, Lintao Zhang, and Lidong Zhou. 2020. Rammer: Enabling Holistic Deep Learning Compiler Optimizations with rTasks. In *14th {USENIX} Symposium on Operating Systems Design and Implementation ({OSDI} 20)*. 881–897.

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

[43] Fionn Murtagh. 1991. Multilayer perceptrons for classification and regression. *Neurocomputing* 2, 5-6 (1991), 183–197.

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

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

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

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

[48] 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. 2019. PyTorch: An Imperative Style, High-Performance Deep Learning Library. In *Advances in Neural Information Processing Systems 32*, H. Wallach, H. Larochelle, A. Beygelzimer, F. d'Alché-Buc, E. Fox, and R. Garnett (Eds.). Curran Associates, Inc., 8024–8035. http://papers.neurips.cc/paper/9015-pytorch-an-imperative-style-high-performance-deep-learning-library.pdf

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

[50] Ram Rangan, Mark W Stephenson, Aditya Ukarande, Shyam Murthy, Virat Agarwal, and Marc Blackstein. 2020. Zeroploit: Exploiting Zero Valued Operands in Interactive Gaming Applications. *ACM Transactions on Architecture and Code Optimization (TACO)* 17, 3 (2020), 1–26.

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

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

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

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

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

[56] Tianzhe Wang, Kuan Wang, Han Cai, Ji Lin, Zhijian Liu, Hanrui Wang, Yujun Lin, and Song Han. 2020. Apq: Joint search for network architecture, pruning and quantization policy. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*. 2078–2087.

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

[58] Ziheng Wang. 2020. SparseRT: Accelerating Unstructured Sparsity on GPUs for Deep Learning Inference. In *Proceedings of the ACM International Conference on Parallel Architectures and Compilation Techniques*. 31–42.

[59] Zhekai Zhang Wang, Hanrui and Song Han. 2018. SpAtten: Efficient Sparse Attention Architecture with Cascade Token and Head Pruning. In *IEEE International Symposium on High-Performance Computer Architecture*.

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

[61] Haichuan Yang, Shupeng Gui, Yuhao Zhu, and Ji Liu. 2020. 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*. 2178–2188.

[62] Jie Amy Yang, Jianyu Huang, Jongsoo Park, Ping Tak Peter Tang, and Andrew Tulloch. 2020. Mixed-Precision Embedding Using a Cache. *arXiv e-prints* (2020), arXiv–2010.

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

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

[65] Takashi Yoshida and Kenichi Ohki. 2020. Natural images are reliably represented by sparse and variable populations of neurons in visual cortex. *Nature communications* 11, 1 (2020), 1–19.

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

[67] Lianmin Zheng, Chengfan Jia, Minmin Sun, Zhao Wu, Cody Hao Yu, Ameer Haj-Ali, Yida Wang, Jun Yang, Danyang Zhuo, Koushik Sen, et al. 2020. Ansor: Generating high-performance tensor programs for deep learning. In *14th {USENIX} Symposium on Operating Systems*

Design and Implementation ({OSDI} 20). 863–879.

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

[69] Maohua Zhu, Tao Zhang, Zhenyu Gu, and Yuan Xie. 2019. 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*. 359–371.

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