SpMMPlu: A Compiler Plug-in with Sparse IR for Efficient Sparse Matrix Multiplication

Tao Yang[1,2,3,*], Yiyuan Zhou[1,2,*], Qidong Tang[1], Feng Xu[1,2], Hui Ma[1], Jieru Zhao[1] and Li Jiang[1,2,3,4]
[1]Shanghai Jiao Tong University, Shanghai, China, [2]Huawei Technologies Co. ,Ltd, [3]Shanghai Qi Zhi Institute, Shanghai, China
[4]MoE Key Lab of Artificial Intelligence, AI Institute, Shanghai Jiao Tong University

***Abstract*—Sparsity is becoming arguably the most critical dimension to explore for efficiency and scalability as deep learning models grow significantly larger. Particularly, pruning is a common method to reduce redundant computations in attention-based and convolution-based models. The induced sparse matrix multiplication (SpMM) normally requires domain-specific hardware architecture (DSA) to eliminate unnecessary zero-valued computations. However, generating an optimal kernel code for SpMM on general-purpose and ISA-based spatial accelerators without changing the hardware architecture is still an open problem.**

In this paper, we propose a compiler plug-in named SpMMPlu, which can extend the representation and optimization ability for SpMM in current deep learning compiler frameworks that only support dense matrix multiplication. The key of SpMMPlu is a flexible intermediate representation— Sparse IR, representing the SpMM with various sparsity patterns based on meta-ops with a multi-level structure. Meta-op takes abstraction of the hardware intrinsic as its minimum granularity, and the powerful optimizers of existing NN compiler backends (e.g., Auto-schedule in TVM, AKG in MindSpore) can be easily reused for its computational scheduling and code generation. Moreover, we propose a two-step (segmentation & grouping) method to achieve an efficient Sparse IR for each sparsity pattern. Only three passes are added in SpMMPlu to provide an automatic solution for SpMM kernel code generation. We embed SpMMPlu into MindSpore and do experiments on NVIDIA V100 GPU and Huawei Ascend 910 to verify its effectiveness and scalability. The results show that with SpMMPlu, MindSpore can support various sparsity patterns and deliver a $1.93\times$ (on V100 GPU) and $2.21\times$ (on AScend 910) speedup averagely compared to the dense counterpart.

***Index Terms*—Sparsity; Plug-in; DNN compiler; Intermediate representation; CNN; Transformer.**

I. INTRODUCTION

Deep neural networks (DNN) have been developing rapidly in many applications, such as image classification, object detection, and natural language processing [1]. As these models become larger, they are inevitably getting sparse to relieve the challenge of the tremendous computation and the large amounts of data transfer between off-chip memory and on-chip memory. For convolutional neural networks (CNN), weight pruning technology is a promising method to achieve sparsity while maintaining the model accuracy. According to the granularity, these pruning methods can be divided into two categories. Between them, the structured pruning method generates regular channel-wise, filter-wise, block-wise sparsity, etc. [2], while unstructured pruning generates fine-grained sparsity in weight matrix [3]. For transformer-based neural networks (e.g., BERT, RoBERTa, and the GPT family [4]), researches mainly focus on the sparsity in attention mechanism. There are a large number of redundant connections in the attention mechanism, and many works [5]–[8] have proposed to remove these redundant connections.

Meanwhile, general-purpose graphics processing units (GPGPU) and other spatial accelerators that employ specialized and hierarchical computation and memory resources, have recently shown great potential for tensor applications such as deep learning, scientific computing, and data mining [8], [9]. However, for the diverse sparsity patterns in CNNs and Transformers above, only a small part of them can be exploited by GPGPU directly [2], [10]. Most of these sparsity patterns need to cooperate with a dedicated domain-specific hardware architecture (DSA) in an algorithm-hardware codesign style [3], [7], [8] to utilize the sparsity. However, how to exploit these diverse sparsity patterns on typical spatial architectures like GPGPU conveniently is still an open problem. On the one hand, GPGPU and most of the other spatial accelerators do not inherently support sparsity. It is hard to extract a unified expression paradigm from various sparsity patterns suitable for the architecture characteristics of GPGPU and other spatial accelerators, which hinders the automatic generation of high-quality kernel code for sparse operators. On the other hand, the cost of algorithm-architecture codesign is too high, lacking of generality and scalability.

We therefore propose SpMMPlu, a compiler plug-in, which can expand the ability of existing DNN compilers to support the various sparsity patterns on GPGPU and other typical spatial accelerators. A novel intermediate representation (IR)— **Sparse IR** is proposed to efficiently represent sparse matrix multiplication (SpMM) with various sparse patterns using a unified paradigm— **meta-op**. The hardware-intrinsic abstraction is considered in meta-ops as the finest-grained workload for high resource utilization. Each meta-op aggregates lots of the finest-grained workloads, thus exploiting the widespread locality in SpMM. Our contributions could be summarized as follows:

- We propose a novel representation to describe the compute behavior of the SpMM on GPGPU and other spatial accelerators, named **Sparse IR**. Sparse IR is a multi-level IR representing different granularity of computations according to the multiple-level parallelism / sequentiality in spatial architecture. Each basic meta-op, the innermost Sparse IR, is essentially a sliced dense matrix multiplication (GEMM), for which we can get high-quality implementation code conveniently, benefiting from the powerful scheduling ability in existing DNN compilers.
- We propose a plug-in (named *SpMMPlu*) to provide an automatic compilation for SpMM in existing NN compilers. Specifically, In SpMMPlu, a novel *meta-op segmentation & grouping algorithm* is proposed to achieve efficient Sparse IR for a specific sparsity pattern. A *DFG transformation pass* will change the data flow graph (DFG) according to the Sparse IR in the front end of the compiler. Moreover, two passes (*layout transformation elimination pass* and *meta-ops fusion pass*) are proposed for kernel code auto-generation in the backend of the compiler.
- Experiments on a variety of tensor applications show that,

This work was partially supported by the National Key Reserch and Development Program of China (2018YFB1403400), National Natural Science Foundation of China (Grant No. 61834006), Shanghai Science and Technology Committee (No.18ZR1421400) and a project from Huawei.

This work was partially done by Yiyuan Zhou and Feng Xu during their internship in Huawei Mindspore department. The author Tao Yang is supported by Wu Wen Jun Honorary Doctoral Scholarship, AI Institute, Shanghai Jiao Tong University. Corresponding author: Li Jiang and Jieru Zhao.

*These authors contributed equally.

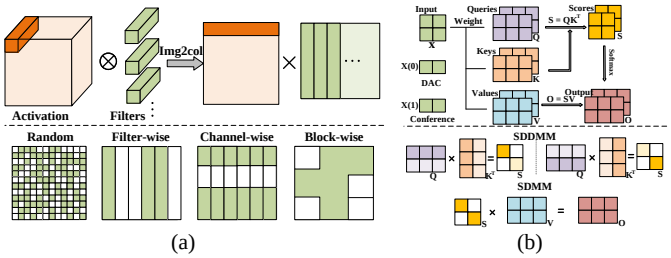

Fig. 1: Sparsity in CNN and Transformer: (a) Sparsity patterns caused by unstructured pruning and structured pruning method in CNN. (b) The calculation process of the attention mechanism, and the caused SDMM and SDDMM in sparse attention.

with SpMMPlu, MindSpore can achieve more than 1.93× and 2.21× speedup on NVIDIA V100 GPU and Huawei Ascend 910 with several widely-used NN models. The performance of SpMMPlu-Mind also exceeds the other popular NN compilers& frameworks (e.g., TVM [11], DeepSpeed [12] and Pytorch [13]) and kernel libraries (e.g., cuBLAS [9], cuSPARSE [14] and sputnik [15]) on NVIDIA V100 GPU.

II. BACKGROUND

A. Various forms of sparsity

Research on neural science and deep learning suggests that a deep neural network is sparse [16]. Various model compression algorithms are shown to construct sparse models with little accuracy degradation. In convolutional neural networks (CNN), As shown in Fig. 1(a), unstructured pruning [3] prunes model weights without a regular pattern, while other works prune DNN models in a structured granularity, such as in the filter, channel and block level in CNN [2]. Transformer models nowadays have achieved impressive performance in natural language processing and computer vision applications by effectively capturing contextual knowledge from the entire sequence. The attention mechanism is the key operation in Transformer models. Fig. 1(b) depicts the computational process of the self-attention (abbreviated as attention) mechanism. The main computations are the GEMM between query (Q) matrix and key (K) matrix and the GEMM between score (S) matrix and value (V) matrix. The attention mechanism is essentially a content-based similarity search. In general, most tokens in the sequence are irrelevant to the current query, which makes the attention mechanism inherently sparse, and the sparsity is mainly contained in the score matrix. This sparse score matrix is the output of $Q \times K^T$ and the input of $S \times V$, which turns the attention operation into sampled dense-dense matrix multiplication (SDDMM) and sparse-dense matrix multiplication (SDMM). There are also many works to adopt various sparsity patterns on the score matrix [7], [8].

B. Existing designs with sparsity

1) Domain specific architecture for sparsity acceleration: Many works co-design sparse algorithms with hardware architecture. Sparse Tensor core [17] proposes a vector-wise sparsity pattern to improve the workload balance and a micro-architecture optimization to adapt to this sparsity pattern. EIE [18] designs a new data encoding/decoding node and a new Processing Element (PE) to speed up sparse matrix-vector multiplication (SpMV). SCNN [19] designs another architecture of PE, which supports sparse convolution in a compressed format. SpAtten [7] proposes cascade token-level pruning in Transformer, and a dedicated hardware module for fetching and reordering the sparse tokens. Sanger [8] proposes a finer-grained pruning method on the score matrix in attention. Meanwhile, correspondingly, a reconfigurable systolic-format architecture is also designed in Sanger to support this special sparsity pattern. However, in real scenarios, we can not ideally design a chip with domain-specific architecture (DSA) for each algorithm and each sparse pattern, leading to the dilemma of excessively high cost. This motivates us to focus on efficient auto-implementations for various sparse models on more general-purpose architecture like GPGPU and other spatial accelerators.

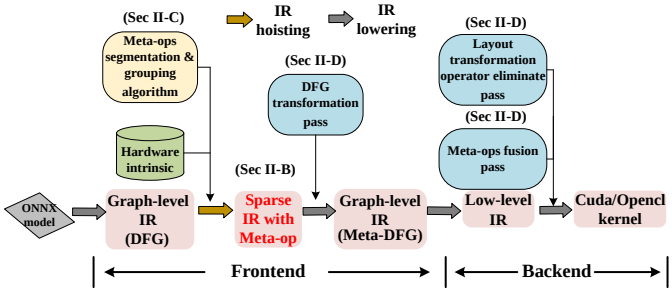

Fig. 2: The overall workflow of SpMMPlu.

2) Sparsity support in DNN frameworks and compilers: Deep learning frameworks like PyTorch [13] or compilers like TVM [11] exploit sparsity by vendor-specific libraries like cuSPARSE [14] or user-provided sparsity kernel templates [11]. However, these methods lack an understanding of the specific sparsity pattern, often leading to subpar performance. For example, cuSPARSE tends to apply a compressed sparse row (CSR) compression on any sparse tensors, which misses the message contained in specific sparsity patterns like channel and block level sparsity. As a result, cuSPARSE has been shown to underperform the dense matrix operation library cuBLAS [9], even when the sparsity of the matrices reaches 98%. This also motivates us to find a universal paradigm to generate high-quality kernel code for various sparsity schemes.

III. SPMMPLU DESIGN

A. SpMMPlu overview

We present the overall workflow of SpMMPlu as in Fig. 2. The input ONNX model is firstly lowered to graph-level IR, a data flow graph (DFG) with normal operators. The *meta-op segmentation & grouping algorithm* hoists the graph-level IR to our *Sparse IR*, which represents the specific sparse pattern using *meta-op*. During this step, the hardware-intrinsic is abstracted as a critical input for the algorithm. Then, we use a *DFG transformation pass* to lower the *Sparse IR* back to graph-level IR again. In this process, the GEMM nodes in the original DFG are replaced by meta-ops ("sliced" matmuls). The DFG we achieved in this step is named Meta-DFG. The Meta-DFG is further lowered to low-level IR like Halide IR [11] and AKG IR [20] in the backend of the compilers. *Layout transformation operator elimination pass* and *meta-ops fusion pass* are added to the backend to optimize the low-level IR and finally generate efficient kernel code for each SpMM.

B. SparseIR: An IR to represent sparsity patterns using meta-ops

A matrix multiplication (GEMM) operation has two input matrices and one output matrix. Depending on whether the sparsity is in the input matrix or the output matrix, the sparse matrix multiplication (SpMM) can be divided into two categories, *input sparsity SpMM* and *output sparsity SpMM*. Characteristically, in CNN, the sparsity is induced by the weight pruning technology, resulting in input sparsity SpMM (SDMM). As for the sparse attention module in Transformer, the sparse score matrix makes $Q \times K^T$ as an output sparsity SpMM (SDDMM). For both SDMM and SDDMM, we use a uniform method to map the sparsity of the two input matrices and one output matrix into a three-dimension computational space as shown in Fig. 3(a), which describes the computation properties of SpMM

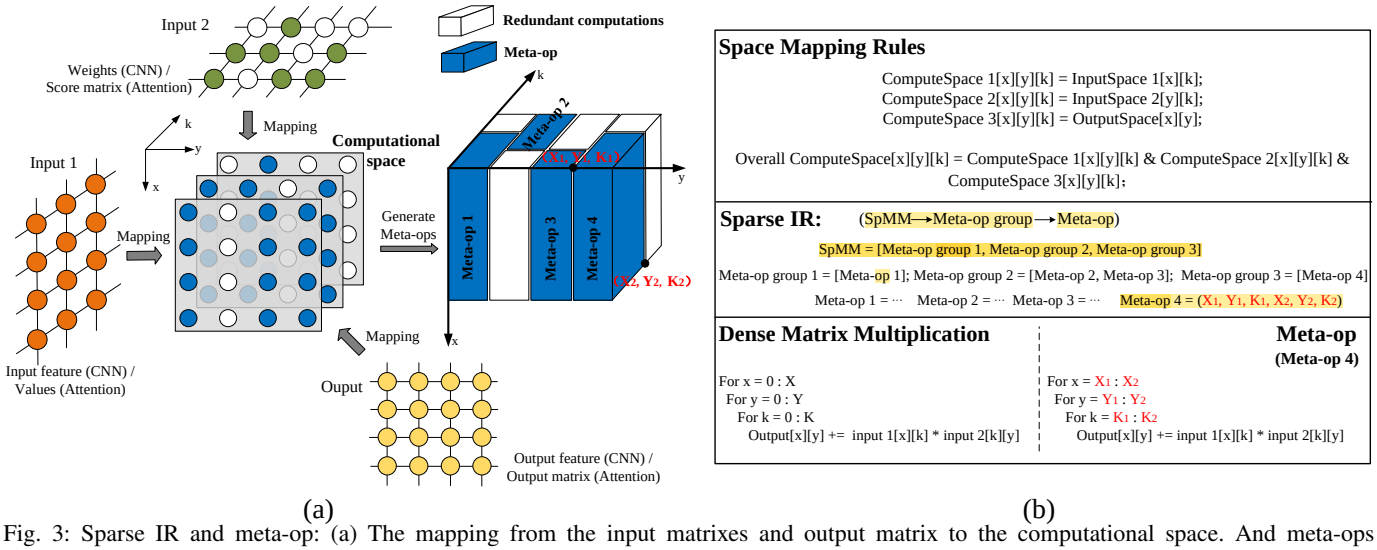

Fig. 3: Sparse IR and meta-op: (a) The mapping from the input matrixes and output matrix to the computational space. And meta-ops generation from the computational space. (b) The space mapping rules, the syntax for Sparse IR and the semantics of a meta-op.

more comprehensively. Specifically, according to the rules described in Fig. 3(b), we broadcast the 2-dimensional matrices in data space along the third dimension it does not have in the 3-dimensional computational space. Then we **and** the broadcasting results of the three matrices to get the final computational space mask. This rule also applies when more than one matrices are sparse.

In computational space, we gather adjacent non-zero elements in a cuboid as the basic computing unit in SpMMPlu, named meta-op. Take the computational space in Fig. 3(a) as an example, four meta-ops are generated in this space. Comparing the computational process of a meta-op (e.g., *meta-op 4* in Fig. 3(b)) with a dense GEMM, we can figure that a meta-op is essentially a "sliced" dense GEMM. They have **the same computational paradigm**, and the only difference lies at the loop bounds. This is a useful feature, which means we can easily reuse the powerful scheduling ability of the original NN compilers (e.g., Auto-schedule in TVM [11] or AKG in MindSpore [20]) to get a high-quality implementation for each meta-op.

To represent SpMM with meta-ops, we propose a novel *Sparse IR* as in Fig. 3(b). Sparse IR is a multi-level IR representing different granularity of computations according to the multiple-level parallelism / sequentiality in spatial architectures. Here, we use sub-IR to represent the finer-granularity computations in Sparse IR. For typical GPGPU, Sparse IR is organized in a three-level structure, i.e., SpMM operator sub-IR, meta-op group sub-IR, and meta-op sub-IR. Each innermost meta-op sub-IR consists of coordinates of two critical points (as the two points of the *meta-op 4* in Fig. 3(a)) in computational space to uniquely identify a specific meta-op. The coordinates of the two critical points directly determine the loop bounds for specific meta-op. The meta-op sub-IR matches primitive/thread-level parallelism (e.g., *wmma* in NVIDIA V100 GPU) in GPGPU, The meta-op group sub-IR represents the parallelism of meta-ops, which matches block-level parallelism in GPGPU (the meta-ops in a single meta-op group are mapped to different blocks in the same kernel). For most of the other spatial accelerators, the amount of calculation per meta-op is always too small for the massive parallel resources, so it is necessary to introduce this middle-level sub-IR for high resource utilization. The SpMM operator sub-IR represents the sequentiality among the meta-op groups, which matches the kernel-level serialization in GPGPU. In this way, The meta-op groups (cuda/opencl kernels in GPGPU) will be executed sequentially to complete the computations of a whole SpMM operator.

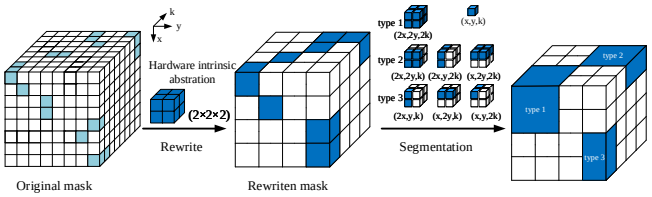

Fig. 4: Rewrite computational mask using hardware intrinsic abstraction. Then expand meta-ops in three type cases during the first iteration of the segmentation algorithm.

C. Strategy to generate efficient sparse IR

To achieve efficient Sparse IR for a specific sparsity pattern, we should make each granularity of the computations represented by each level of sub-IR to fully utilize the parallel resources. We design a hardware-intrinsic-aware strategy to find the most promising Sparse IR for each sparsity pattern. For typical GPGPU, the hardware intrinsic is usually abstracted as a cuboid in computational space. For example, we can abstract tensor core as a $16 \times 16 \times 16$ cuboid (the "wmma" primitive for tensor core in NVIDIA V100 GPU executes a $16 \times 16 \times 16$ GEMM each time). We use the hardware abstraction as the initial minimum size of a meta-op, which guarantees the high utilization of the finest granularity of hardware resources. We first rewrite the computational space mask according to the hardware abstraction as shown in Fig. 4. Then we implement the meta-op segmentation & grouping algorithm based on the rewritten mask as Alg. 1 to generate meta-ops & meta-op groups, respectively.

The segmentation algorithm is the process of expanding meta-ops from small to large. We prefer large meta-ops because larger meta-ops make better use of locality and are thus more computationally efficient. At first, each non-zero in the rewritten mask is a $1 \times 1 \times 1$ meta-op, we then iterate over each meta-op, combining as many as possible smaller meta-ops into a larger meta-op iteratively. For example, in the first iteration, the $1 \times 1 \times 1$ meta-op may be extended into a bigger meta-op size of $2 \times 2 \times 2$ (type 1 in Fig. 4), $2 \times 2 \times 1$ (type 2 in Fig. 4) and $2 \times 1 \times 1$ (type 3 in Fig. 4) according to the different aggregation degree of the around meta-ops. In practice, the aggregation degree thresholds of these three types are set empirically and should achieve a good trade-off between introducing redundant computation and improving computational efficiency.

Next, in the grouping algorithm, we handle input and output sparsity separately to obtain efficient meta-op grouping results. For

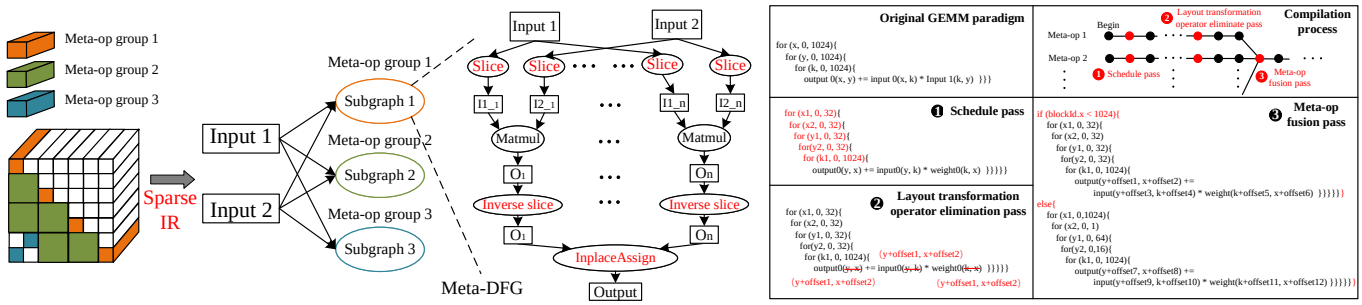

Fig. 5: The process of code generation: (a) A computational space and the resulted meta-op groups. (b) The Meta-DFG after DFG transformation pass. (c) The compilation process in the backend of the compiler. And the changes in kernel code with each critical pass.

Algorithm 1: Meta-Op Segmentation & Grouping

Input: $Mask[X][Y][K]$
Output: List of Meta-Op Group Lg

* ***Meta-op Segmentation*** *\
Use $S_{(x,y,k)}$ *to represent meta-op set with size* x, y, k
$S_{(1,1,1)}$ = Set ()
for $i:1 \to X,\quad j:1 \to Y,\quad l:1 \to K$ **do**
  if $Mask[i][j][l]$ *is True*
    $S_{(1,1,1)}$.insert($\{i, j, l\}$)
Q = Queue ($S_{1,1,1}$); \\Q stores all untraversed $S_{(x,y,k)}$
Ls = List of Meta-op Set
while $Q \neq \emptyset$ **do**
  $S_{(x,y,k)} = Q$.pop()
  for $\{i, j, l\} \in S_{(x,y,k)}$ **do**
    if *arounding meta-ops* ***meet*** *type 1 in Fig. 4*
      Remove these meta-ops from $S_{(x,y,k)}$
      $S_{(2x,2y,2k)}$.insert($\{i, j, l\}$)
    else if *arounding meta-ops* ***meet*** *type 2 in Fig. 4*
      Remove these meta-ops from $S_{(x,y,k)}$
      $S_{(*)}$.insert($\{i, j, l\}$) (*)=(2x,2y,k),(2x,y,2k) or (x,2y,2k)
    else if *arounding meta-ops* ***meet*** *type 3 in Fig. 4*
      Remove these meta-ops from $S_{(x,y,k)}$
      $S_{(*)}$.insert($\{i, j, l\}$) (*)=(2x,y,k), (x,2y,k) or (x,y,2k)
  if $S_{(x,y,k)} \neq \emptyset$
    Ls.append ($S_{(x,y,k)}$)
    $S_{(x,y,k)}$.clear()
  for $S_{(*)} \in \{ S_{(x',y',k')}\ (x' = x, 2x; y' = y, 2y; k' = k, 2k) \}$ **do**
    if $S_{(*)} \neq \emptyset$
      Q.push($S_{(*)}$)

* ***Meta-op Grouping*** *\
Lg = List of Meta-op Group
for $S_{(*)} \in Ls$ **do**
  if ***is*** *output_sparsity*
    $G_1 = S_{(*)}$
  else
    $G_1, \cdots, G_n$ = Group_With_No_Reduce ($S_{(*)}$)
  for $i:1 \to n$ **do**
    if $\#(G_i) <$ *Min_Group_Num*
      G_i = Combination_With_No_Additive()
  Append All $G_i (i:1 \to n)$ to Lg.

output sparsity, there is no additive relationship between any two of the meta-ops, thus we group the meta-ops with the same size together as one meta-op group, which is beneficial for achieving balanced workloads on massive parallel resources. As for input sparsity, there may be some meta-ops each producing a partial sum. So they need to be added to produce the corresponding part in the output matrix. For this case, we add a constraint: The meta-ops with additive relationships should not be gathered into one group. This way, the accumulations of these meta-ops are executed as we execute mate-op groups (kernels) sequentially on hardware. With this constraint, each kernel avoids the extremely expensive atomic addition operation, and it guarantees that the meta-ops in a meta-op group can be executed completely in parallel. However, this constraint also has the drawback that it may cause a small number of meta-ops in a group, inducing low utilization of the massive parallel resources. As compensation, we combine several groups with no additive relationship together to improve the hardware utilization.

D. Auto code generation

SpMMPlu offers three compilation passes to extend the capability for SpMM represented by our Sparse IR. We take the sparse computational space in Fig. 5(a) as an example to illustrate the code generation process on GPGPU, where three meta-op groups are obtained by our meta-op segmentation & grouping algorithm in Sec. III-C, and each meta-op group will eventually correspond to a kernel.

1) DFG transformation pass: DFG transformation pass transforms the original DFG to *Meta-DFG* according to the corresponding Sparse IR. Specifically, for the computational space in Fig. 5(a), DFG transformation pass will transform the original single graph representing the dense GEMM into a graph with three subgraphs, each of which corresponds to a meta-op group as shown in Fig. 5(b). In each subgraph, we insert *slice node* in the input side of the original DFG to offer the specific input data for each meta-op. After each matmul node, we insert *inverse slice node* to recover the sliced-out data to a global data format. At last, we insert a *inplace assign node* to gather all the output tensors of each meta-op together to achieve the final output tensor.

2) Layout transformation operator elimination pass: The Meta-DFG is lowered to low-level IR and then be compiled by a serials passes as shown in the *compilation process* part in Fig. 5(c). At beginning, the low-level IR of a single meta-op is shown as the *original GEMM paradigm* part in Fig. 5(c), which is a typical three-tier for-loop to represent the dense GEMM. Then this IR is processed by the ❶ *schedule pass* maintained originally in NN compilers like the Polyhedron pass in AKG [20] or the Auto-schedule pass in TVM [11]. The schedule pass explores the schedule space (e.g., loop split, loop reorder, vectorize, etc.) to get efficient computational paradigm for each meta-op on a specific hardware platform. Then, compilers often have passes (e.g., Flatten pass, Rewrite pass, etc.) to further optimize the computational paradigm.

After these paradigm optimization passes, the low-level IR will be processed by our added ❷ *layout transformation operator elimination pass*. The essence of this pass is that we find the layout transformation operators (e.g., slice, reshape, transpose, etc.) can be normally merged into computation-intensive operators by changing a little bit about the kernel code of the computation-intensive operators. Specifically, in the compilation process for SpMM in CNN and Transformer, the effect of this pass is reflected in two aspects. First, for the low-level IR after paradigm optimization passes, this pass will offset the indexes of the three tensors in IR according to the meta-op location message

Tab. I. Evaluated models with different sparsity patterns and corresponding sparsity ratios

Model	Type	Sparsity / Type	Sparsity ratio	Model	Type	Sparsity	Sparsity ratio
Transformer (NLP)	SDDMM (output sparsity)	**GST** [21] / Structured	92.5%	Transformer (NLP)	SDMM (input sparsity)	**GST** [21] / Structured	92.5%
		Longformer (**LF**) [5] / Structured	94.9%			Longformer (**LF**) [5] / Structured	94.9%
		Big Bird (**BD**) [6] / Other	85.4%			Big Bird (**BD**) [6] / Mixed	85.4%
		Random (**RD**) [8] / Unstructured	95.2%			Random (**RD**) [8] / Unstructured	95.2%
Resnet-18 (CV)	Conv (input sparsity)	filter-wise (**FW**) [2] / Sturctured	61.2%	Resnet-50 (CV)	Conv (input sparsity)	filter-wise (**FW**) [2] / Sturctured	63.9%
		channel-wise (**CW**) [2] / Structured	66.3%			channel-wise (**CW**) [2] / Structured	71.8%
		PCNN [22] / Other	86.4%			**PCNN** [22] / Other	89.9%
		Random (**RD**) [3] / Unstructured	94.5%			Random (**RD**) [3] / Unstructured	96.4%

from Sparse IR, as ❷ in Fig. 5(c). In this way, the added *slice node*, *inverse slice node* and *inplace assign node* in *DFG transformation pass* are merged into the matmul nodes, which reduce a big amount of the data movement and kernel launch overhead in these layout transformation operators. Second, in sparse CNN, we need *img2col* operators to transform the input data format, thus we can implement a sparse convolution operation using sparse matmul. The *img2col* operators are also layout transformation operators, and we merge them into the data loading part (from off-chip global memory to on-chip memory) of the matmul kernel code to further reduce the data movement and kernel launch overhead. Besides, this pass also supports merging successive layout transformation operators around the other computation-intensive operators (e.g., GEMV, reduce, etc.) iteratively in DFG. It means that the layout transformation operator elimination pass is a general-purpose optimization pass, which can optimize the performance in various scenarios other than SpMM compilation process.

3) Meta-ops fusion pass: This meta-ops fusion pass then merges the meta-ops in a meta-op group into a single kernel by adding the block condition judging statements in the kernel code as ❸. With this pass, we can gather multiple meta-ops to make full use of the massive parallel computing resources in hardware, while reducing the kernel launch overhead.

IV. EXPERIMENT

A. Experiment setup and benchmark

The main code of SpMMPlu includes the Sparse IR, and the three added passes in the front end and backend of the NN compiler. As a compiler Plug-in, SpMMPlu can be easily embedded in various deep learning compilers like TVM [11], MindSpore [23], and MLIR-based compilers [24] with only a little effort to modify the code with the default data structures in each compiler. In our experiments, we embed SpMMPlu into MindSpore (named **SpMMPlu-Mind**) as a full-stack solution for SpMM.

We evaluate SpMMPlu-Mind on one attention-based model and two convolutional-based models with four different sparsity patterns, respectively, on NVIDIA V100 GPU and Huawei Ascend 910. The specific sparse models across different task domains are shown in Table I. The sparsity ratio we used in each case is calculated according to the experimental results in the corresponding paper, ensuring acceptable accuracy loss in these sparse models. We compare SpMMPlu-Mind with eight representative solutions, including two popular dense DNN compilers: TVM (0.9.dev0) [11] and MindSpore [23]. One sparse DNN compiler: TVM-Sparse [11]. Two popular deep learning frameworks: DeepSpeed [12] (specially optimized for sparse Transformer model) and PyTorch 1.13 [13]. To evaluate the state-of-the-art dense/sparse kernel libraries in an end-to-end model, we create implementations by wrapping cuBLAS [9] (cuBLAS-Mind), cuSPARSE [14] (cuSPARSE-Mind) and sputnik [15] (sputnik-Mind) in MindSpore, separately.

B. Overall performance

1) Performance on GPU: The first row of Fig. 6 shows the inference latency of the attention-based Transformer with the four

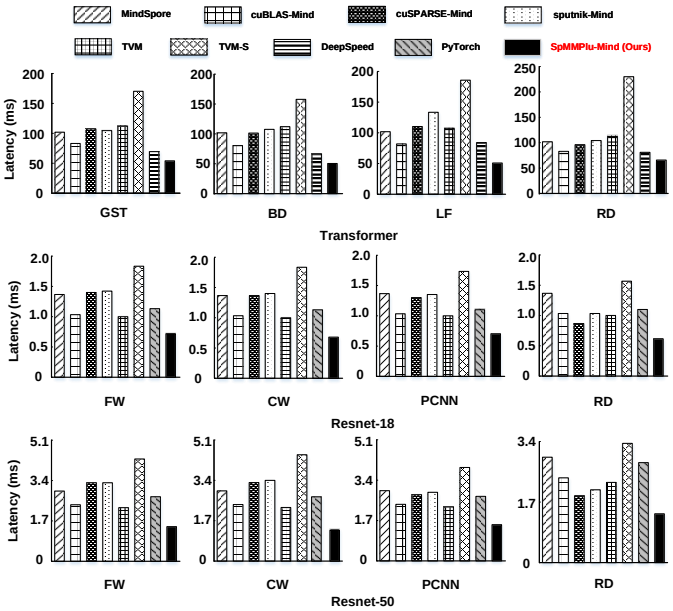

Fig. 6: Inference latency on NVIDIA V100 GPU.

sparsity pattern on NVIDIA V100 GPU. For sparse attention in cuSPARSE, sputnik, TVM-S, DeepSpeed and SpMMPlu-Mind, $Q \times K^T$ is SDDMM (output sparsity), and $S \times V$ is SDMM(input sparsity). However, PyTorch, MindSpore, TVM and cuBLAS treat $Q \times K^T$ and $S \times V$ as dense matrix multiplications. For all four cases, TVM-S has no advantage over the corresponding dense TVM, because the kernel templates it uses cannot effectively support different sparse patterns. The SOTA sparse kernel sputnik and cuSPARSE perform worse than the dense cuBLAS on the four patterns. The reason is that the sparsity ratios in these three patterns are too small. However, sputnik and cuSPARSE usually can outperform cuBLAS only when the sparsity ratio reaches very high values (at least 96% in our experiments). DeepSpeed performs the best among them, in which there are some targeted optimizations for these widely used sparsity patterns in Transformer. Compared with DeepSpeed, our SpMMPlu-Mind performs $1.28\times$, $1.31\times$, $1.38\times$ and $1.22\times$ speedup with these four sparsity patterns, separately.

The second and the third row of Fig. 6 shows the inference latency of the Resnet-18 and Resnet-50 with another four widely used sparsity patterns on NVIDIA V100 GPU. For the two structured pruning patterns filter-wise and channel-wise sparsity, SpMMPlu-Mind not only performs best among all these works, but also has enormous performance improvement compared with existing works ($1.48\times$ and $1.54\times$ for Resnet-18, $1.56\times$ and $1.69\times$ for Resnet-50 compared with the best TVM). For the other two patterns, PCNN and RD sparsity, TVM and cuSPARSE perform best among the existing works. SpMMPlu-Mind also performs $1.35\times$ and $1.39\times$ with PCNN, $1.31\times$ and $1.35\times$ with RD on Resent-18 and Resnet-50 compared with the best existing implementations.

The performance gain of SpMMPlu-Mind comes from three aspects. One is that our Sparse IR is excellent at catching the locality feature in SpMM, which is particularly significant in various rule-

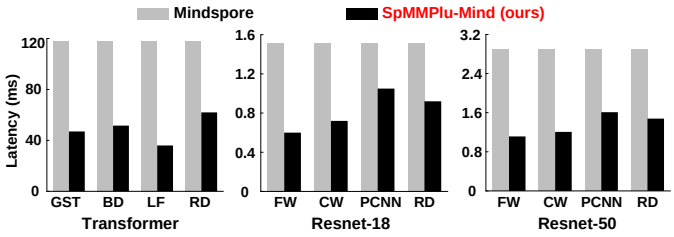

Fig. 7: Inference latency on Huawei Ascend 910.

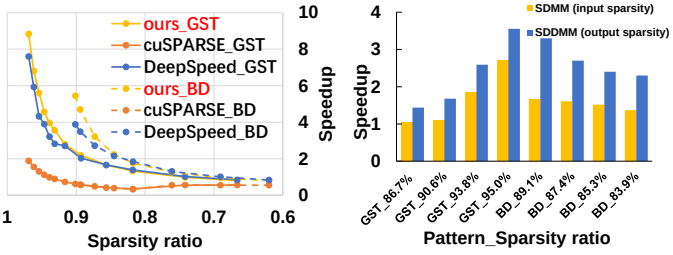

Fig. 8: Ablation study of sparsity ratio and sparsity location: (a) The speed up of the three designs on different sparsity ratios. (b) The speed up comparisons between input sparsity and output sparsity.

based sparsity patterns like GST, LF, BD sparsity in Transformer and FW, CW sparsity in CNN. The second is that the meta-ops in our Sparse IR can be well scheduled easily through reusing the powerful schedule ability in the embedded compilers, bringing high-quality kernel code. The third is that with our segmentation & grouping algorithm in Sec. III-C, the achieved three-level-structured Sparse IR fully takes the abstraction of the hardware intrinsic and multiple-level parallelism / sequentiality in architectures into consideration, which ensure a full utilization of the massive parallel resources.

2) Performance on Huawei Ascend 910: We also implement SpMMPlu-Mind on Huawei Ascend 910 to prove the scalability on various hardware platforms. The results are shown in Fig. 7. Compared with the original compiler MindSpore only supporting dense operations, SpMMPlu-Mind Speedup $2.47\times$, $1.96\times$ and $2.20\times$ averagely on Transformer, Resnet-18 and Resnet-50. By abstracting hardware intrinsic as the minimum unit in segmentation & grouping algorithm, and mapping the different levels of parallelism / sequentiality in hardware architecture to different levels in Sparse IR, SpMMPlu can be easily extended to other spatial accelerators.

C. Influence of sparsity ratio and sparsity location.

We provide ablation studies to demonstrate the impact of sparsity ratio (Fig. 8 (a)) and the location of sparsity (Fig. 8 (b)) on inference latency. We choose two sparsity patterns BD, LF, and the common sequence length (4096) in Transformer, as a case study for these two factors. Fig. 8 (a) shows the speedups of the three sparse matrix operation libraries on different sparsity ratios compared with the original dense MindSpore on GPU. We can figure that all the three implements suffer from low performance when the sparsity ratio is below 68%. However, For the sparsity ratio higher than 85%, SpMMPlu offers significantly higher speedup compared with the other two implementations. Besides, the sparsity location also affects the inference performance. We show the speedups of SDMM (input sparsity) and SDDMM (output sparsity) in various sparsity *pattern_ratio*s compared with their dense counterpart MindSpore as in Fig. 8 (b). It can be figured that, for the same sparsity pattern and sparsity ratio, the SDDMM always give a higher speedup performance than SDMM. This is because, for input sparsity, there is an additional constraint in the meta-op segmentation & grouping algorithm that the meta-ops with additive relationships should not be grouped into one meta-op group. With this constraint, more meta-op groups (also means more cuda kernels) are generated. We combine several groups with no additive relationship together to improve hardware utilization, however, inducing unbalanced workloads because the meta-ops in combined groups may have different sizes.

V. CONCLUSION

SpMMPlu is a compiler plug-in for existing NN compilers to extend the compilation ability for SpMM. In SpMMPlu, a three-level-structured Sparse IR is proposed using meta-op to represent SpMM in a hardware-aware method, in which the parallelism and sequentiality in hardware structure are well matched with different granularity of the computations in SpMM. Experimental results show that with SpMMPlu, SpMMPlu-Mind achieves noticeable speedup on various hardware platforms.

REFERENCES

[1] M. R. Minar *et al.*, "Recent advances in deep learning: An overview," *CoRR*, 2018. [Online]. Available: http://arxiv.org/abs/1807.08169

[2] W. Wen *et al.*, "Learning structured sparsity in deep neural networks," in *NIPS*, 2016.

[3] S. Han *et al.*, "Deep compression: Compressing deep neural network with pruning, trained quantization and huffman coding," in *ICLR*, 2016.

[4] D. W. Otter *et al.*, "A survey of the usages of deep learning for natural language processing," *TNNLS*, 2021.

[5] I. Beltagy *et al.*, "Longformer: The long-document transformer," *arXiv:2004.05150*, 2020.

[6] M. Zaheer *et al.*, "Big bird: Transformers for longer sequences," *Advances in Neural Information Processing Systems*, vol. 33, 2020.

[7] Wang *et al.*, "Spatten: Efficient sparse attention architecture with cascade token and head pruning," in *HPCA*, 2021.

[8] L. Lu *et al.*, "Sanger: A co-design framework for enabling sparse attention using reconfigurable architecture," in *MICRO*, 2021.

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

[10] S. Anwar *et al.*, "Structured pruning of deep convolutional neural networks," *J. Emerg. Technol. Comput. Syst.*, 2017.

[11] T. Chen *et al.*, "Tvm: An automated end-to-end optimizing compiler for deep learning." in *OSDI*, 2018.

[12] J. Rasley *et al.*, "Deepspeed: System optimizations enable training deep learning models with over 100 billion parameters," in *KDD*, 2020.

[13] A. Paszke *et al.*, "Pytorch: An imperative style, highperformance deep learning library." 2019.

[14] "cusparselt: A high-performance cuda library for sparse matrix-matrix multiplication." 2021. [Online]. Available: https://docs.nvidia.com/cuda/cusparselt/index.html

[15] T. Gale *et al.*, "Sparse gpu kernels for deep learning," in *SC*, 2020.

[16] T. Yoshida *et al.*, "Natural images are reliably represented by sparse and variable populations of neurons in visual cortex," *Nature Communications*, vol. 11, 2020.

[17] M. Zhu *et al.*, "Sparse tensor core: Algorithm and hardware co-design for vector-wise sparse neural networks on modern gpus," in *MICRO*, 2019.

[18] S. Han and others., "Eie: Efficient inference engine on compressed deep neural network," in *ISCA*, 2016.

[19] A. Parashar and others., "Scnn: An accelerator for compressed-sparse convolutional neural networks," in *ISCA*, 2017.

[20] J. Zhao *et al.*, "Akg: Automatic kernel generation for neural processing units using polyhedral transformations," in *PLDI*, 2021.

[21] R. Child *et al.*, "Generating long sequences with sparse transformers," *arXiv preprint arXiv:1904.10509*, 2019.

[22] Z. Tan *et al.*, "Pcnn: Pattern-based fine-grained regular pruning towards optimizing cnn accelerators," in *DAC*, 2020.

[23] Huawei, "Mindspore." 2020. [Online]. Available: https://www.mindspore.cn/en

[24] C. Lattner *et al.*, "Mlir: Scaling compiler infrastructure for domain specific computation," in *CGO*, 2021.