SparseTIR: Composable Abstractions for Sparse Compilation in Deep Learning

Zihao Ye*
University of Washington
Seattle, WA, USA
zhye@cs.washington.edu

Ruihang Lai†
Carnegie Mellon University
Pittsburgh, PA, USA
ruihangl@cs.cmu.edu

Junru Shao
OctoML
Seattle, WA, USA
jshao@octoml.ai

Tianqi Chen‡
Carnegie Mellon University
Pittsburgh, PA, USA
tqchen@cmu.edu

Luis Ceze‡
University of Washington
Seattle, WA, USA
luisceze@cs.washington.edu

ABSTRACT

Sparse tensors are rapidly becoming critical components of modern deep learning workloads. However, developing high-performance sparse operators can be difficult and tedious, and existing vendor libraries cannot satisfy the escalating demands from new operators. Sparse tensor compilers simplify the development of operators, but efficient sparse compilation for deep learning remains challenging because a single sparse format cannot maximize hardware efficiency, and single-shot compilers cannot keep up with latest hardware and system advances. In this paper, we observe that the key to addressing both these challenges is to leverage composable formats and composable transformations. We propose SparseTIR, a sparse tensor compilation abstraction that offers composable formats and composable transformations for deep learning workloads. SparseTIR constructs a search space over these composable components for performance tuning. With these improvements, SparseTIR obtains consistent performance speedups vs vendor libraries on GPUs for single operators: 1.20-2.34x for GNN operators, 1.05-2.98x for sparse attention operators, and 0.56-7.45x for sparse convolution operators. SparseTIR also accelerates end-to-end GNNs by 1.08-1.52x for GraphSAGE training, and 4.20-40.18x for RGCN inference.

CCS CONCEPTS

• **Software and its engineering** → **Domain specific languages**.

KEYWORDS

Sparse Computation, Tensor Compilers, Code Generation and Optimizations, Scheduling, Vectorization, Tensor Cores, Kernel Fusion

*Part of this work was done during internship at OctoML.
†Part of this work was done at Shanghai Jiao Tong University.
‡Also with OctoML.

ASPLOS '23, March 25–29, 2023, Vancouver, BC, Canada

ACM ISBN 978-1-4503-9918-0/23/03.
https://doi.org/10.1145/3582016.3582047

ACM Reference Format:
Zihao Ye, Ruihang Lai, Junru Shao, Tianqi Chen, and Luis Ceze. 2023. SparseTIR: Composable Abstractions for Sparse Compilation in Deep Learning. In *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 3 (ASPLOS '23), March 25–29, 2023, Vancouver, BC, Canada.* ACM, New York, NY, USA, 19 pages. https://doi.org/10.1145/3582016.3582047

1 INTRODUCTION

Sparsity is becoming ubiquitous in deep learning due to the application of deep learning to graphs and the need for more efficient backbone models. Graph neural networks (GNNs) [39, 52, 92] have made substantial progress in modeling relations in social networks, proteins, point clouds, etc., using highly sparse matrices. Sparse transformers [6, 15, 20] reduce both the time and space complexity of transformers [91] by making the attention mask sparse using manually designed and moderately sparse matrices. Network Pruning [40, 55, 76] prunes the network weight to sparse matrix to reduce model size, the pruned weights are moderately sparse and stored in various formats depending on the pruning algorithm.

Existing libraries, such as cuSPARSE [24], dgSPARSE [28], Sputnik [37] and Intel MKL [95], support only a few sparse operators. As such, they fail to accelerate rapidly evolving emerging workloads such as GNNs on heterogeneous graphs [48, 77, 97] and hypergraphs [35]. Manually optimizing sparse operators can be difficult and tedious. Sparse matrices are stored in compressed formats, and programmers must write manual code to compress or decompress coordinates to access non-zero elements. Furthermore, the compressed sparse formats vary, and operators designed for one format cannot generalize to others. Therefore, *we need a more scalable and efficient approach to developing optimized sparse operators.*

Sparse tensor compilers, such as MT1 [9] and TACO [54], greatly simplify the development of sparse operators by decoupling format specification and format-agnostic computation descriptions. However, applying sparse compilation to deep learning must overcome two major challenges. First, *modern deep learning workloads are quite diverse,* making them hard to fit into a single sparse format pattern provided by existing solutions. Second, *harware backend are evolving and becoming heterogenous,* making it hard for single-shot compilers to keep up with the latest hardware and system advances.

Our key observation is that we can resolve all challenges by introducing two forms of composability:

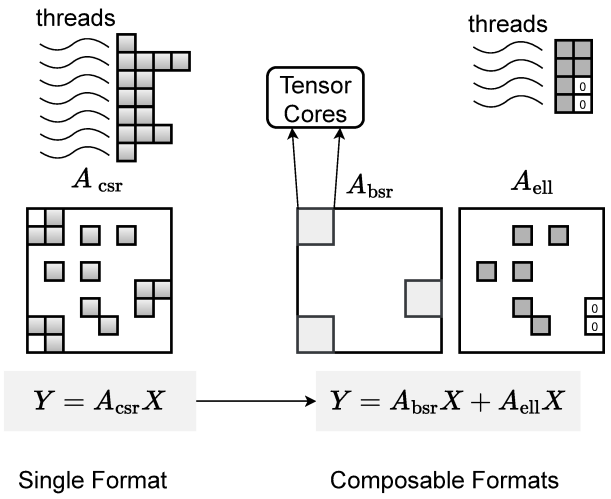


Figure 1: Format composability enables us to leverage multiple formats for different parts in sparse pattern we face in deep learning, and maximize the use of underlying hardware resources.

Format composability. We propose to go beyond the single format option provided by most existing solutions to composable formats (Figure 1) that store different parts of a sparse matrix in the different formats that best fit their local patterns. The compilation process decomposes the original computations into subcomputation routines to enable efficient executions on each local pattern that better match the characteristics of the corresponding deep learning workloads.

Transformation composability. We reconfigure the single-shot sparse tensor program compilation process into a composable set of program transformations. Additionally, we enable a design that incorporates existing loop-level abstractions in dense tensor compilers. This design lets us define our own transformations for sparse data while reusing hardware-specific optimizations (such as tensorization and GPU mapping) from existing solutions, increasing our overall efficiency to incorporate advances in hardware backends.

Combining both forms of composability, we propose SparseTIR, an abstraction that generates efficient sparse operators for deep learning. Our contributions include the following.

- We propose an intermediate representation (IR) with *composable formats* and *composable transformations* to accelerate sparse operators by decomposing formats and specifying schedules.
- We build a performance-tuning system that searches over the parameter space of possible composable formats and composable transformations.
- We evaluate SparseTIR generated kernels on several important sparse deep learning workloads.

SparseTIR offers consistent speedup for single operators relative to vendor libraries on GPUs: 1.20-2.34x for GNN operators and 1.05-2.98x for sparse transformer operators. SparseTIR also accelerates end-to-end GNNs by 1.08-1.52x for GraphSAGE [39] training and by 4.20-40.18x for RGCN [77] inference, 0.56-7.45x for Sparse Convolution [23] operators.

2 SYSTEM OVERVIEW

This section provides an overview of SparseTIR. Figure 2 summarizes our overall design and compares it with existing approaches. The figure's left side shows the design of most existing sparse tensor compilers [79]. Their inputs are (1) tensor expressions, (2) format annotations/specifications that allow only a single format for each matrix, and (3) user-defined schedules. Schedules are applied to high-level IRs such as provenance graph, and then lowered to target device code; we refer to such compilation flow as *single-shot compilation*. These high-level IRs do not reflect low-level information such as loop structures, memory access regions, and branches. However, optimizations such as tensorization[1] requires loop-level AST matching and replacement, which is not exposed in high-level IR. Though tensor compilers such as Halide [70] and TVM [16] implement schedule primitives and code generation on multiple backends, it is difficult to re-use these infrastructures in previous sparse compilers because of the discrepancy of provenance graph and loop-level IR of existing tensor compilers.

SparseTIR builds on top of these previous approaches and introduces a design that enables composable formats and composable transformations. It contains three IR stages. The first stage presents computation in coordinate space, where we describe sparse tensor computations; like in previous work, we decouple format specification and computations. Unlike a single-shot sparse compiler that accepts a single format for each sparse tensor, SparseTIR lets users specify composable formats. The second stage characterizes computation in position space, where the position refers to the index of non-zero elements in the compressed sparse data structure. The concepts of "coordinates" and "positions" were first proposed in Vivienne et al. [85] and then used in Senanayake et al. [79]. The last stage of SparseTIR is a loop-level IR in existing tensor compilers, such as TVM [16], AKG [107] and the affine dialect in MLIR [90]. We design two passes on the IR, namely, sparse iteration lowering and sparse buffer lowering, to transform code from stage I to stage II and stage II to stage III, respectively.

Instead of single-shot compilation, all schedules in SparseTIR are performed as composable program transformations (which do not change the stage of the IR) on the IR instantly. The composable design lets user transform the IR step-by-step and stage-by-stage. To manipulate the coordinate space computation in stage I IR, we can define new schedules as composable transformations applied to the stage I (i.e., stage I schedules). For stages compatible with target loop-level IR, we can apply schedules defined for backend tensor compilers (i.e., stage II/III schedules). Notably, format decomposition can also be formulated as a program transformation at stage I (see §3.2.1).

SparseTIR constructs a joint search space of composable formats and composable transformations for performance tuning of sparse operators. Users can customize the parameterized search

[1]We use this term to describe rewriting the program to use Matrix-Multiply Units such as Tensor Cores in GPU and MXU in TPU.

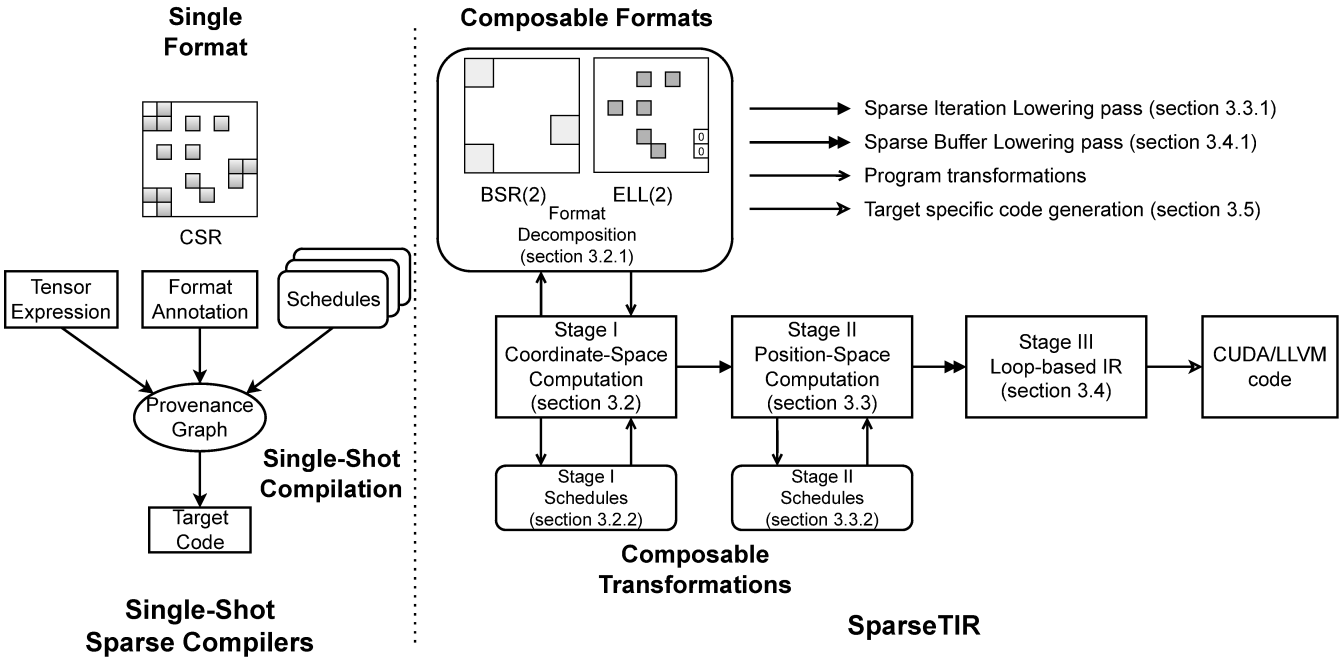


Figure 2: Single-shot sparse compilers vs SparseTIR. The composable formats and composable transformations enable us to create optimizations that fit into broader range of deep learning workloads and leverage more advances in hardware backends.

space by specifying format and schedule templates based on their domain-specific knowledge about the operator and sparse tensor characteristics. When the sparse structure is present at compile-time, we can search for the best formats and schedules that achieve optimal runtime performance in advance. Though the compilation might take some time due to the large search space, the overhead can be amortized because the compiled operator will be re-used many times during training or inference for a fixed sparse structure (as is typical in deep learning).

The rest of the paper is organized as follows. We introduce the SparseTIR design of each stage and compiler passes in Section 3. In Section 4 we evaluate our system in real world sparse deep learning workloads. Section 5 positions SparseTIR relative to related work. Finally, we discuss future work in Section 6 and conclude our work in Section 7.

3 OUR APPROACH

In this section, we introduce the language constructs in SparseTIR, then describe each compilation stage and transformations in the order they appeared in the flow.

3.1 Language Constructs

The SparseTIR language has three major components: axes, sparse buffers and sparse iterations.

Axes. An *axis* is a data structure that defines sparse iteration spaces, which generalize the idea of abstraction levels in previous work [21]. Each axis in SparseTIR has two orthogonal attributes, dense/sparse and fixed/variable, denoting whether the index of non-zero elements in the axis is contiguous or not and whether the number of non-zero elements in the axis is fixed or not. Variable axes are associated with a `indptr` (short for "index pointer") field that points to the address of the indices pointer array; sparse axes are associated with an `indices` field that points to the address of the indices array. Each axis has a parent field that directs to the axis it depends on; a dense-fixed axis has no dependency, and its parent field is always set to none. Axis metadata includes its indices' data

Axis declarations

```
I = dense_fixed(m, "int32")
J = sparse_variable(I, (n, nnz),\
      (j_indptr, j_indices), "int32")
J_ = dense_fixed(n, "int32")
K = dense_fixed(feat_size, "int32")
```

Sparse buffer declarations

```
A = match_sparse_buffer(a, (I, J), "float32")
B = match_sparse_buffer(b, (J_, K), "float32")
C = match_sparse_buffer(c, (I, K), "float32")
```

Sparse iteration declarations

```
with sp_iter([I, J, K], "SRS", "spmm") as [i, j, k]:
    with init():
        C[i, k] = 0.0
    C[i, k] = C[i, k] + A[i, j] * B[j, k]
```

Figure 3: Language constructs in the SpMM operator. Users specify axis dependencies and metadata to create axes. The `match_sparse_buffer` defines sparse buffers and binds them to pointers to their value, and `sp_iter` creates a sparse iteration structure, where "S" and "R" indicate whether the iterator is for spatial or reduction purposes, "spmm" is the name of the sparse iteration as a reference for scheduling.

type, maximum length, number of accumulated non-zeros in this dimension (if variable), and number of non-zeros per row in this dimension (if fixed).

Sparse buffers. A sparse buffer is SparseTIR's data structure for a sparse matrix. We use defined axes to compose the format specification of sparse matrices. We split sparse structure-related auxiliary data and values: axes store auxiliary data, and sparse buffers store only values. Such design lets Two sparse buffers can re-use auxiliary data if they share the sparse layout. Figure 4 shows the decoupled storage of sparse buffers/axes in the SpMM (Sparse-Dense Matrix Multiplication) operator. The composition of axes is expressive to describe various sparse formats, including Compressed Sparse Row/Column (CSR/CSC) format [31], Block Compressed Sparse Row (BSR) format [75], Diagonal Format (DIA) [74], ELLPACK (ELL) format [30, 44, 64], Ragged Tensor [26], Compressed Sparse Fiber (CSF) [82] etc, please refer to Duff et al. [31] for an overview of sparse formats.

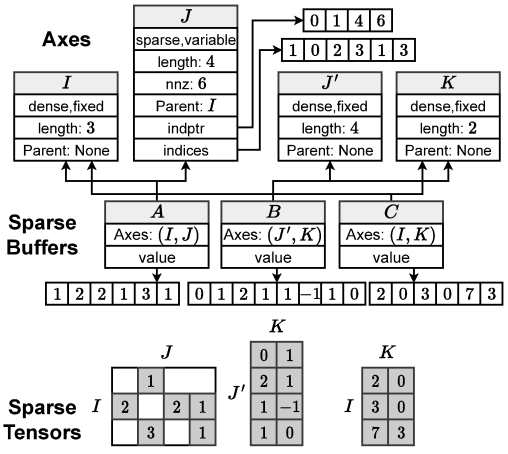


Figure 4: Internal storage of axes and sparse buffers in SpMM: $C_{ik} = A_{ij}B_{jk}$. Sparse buffers store their axes' composition and pointers to their value; axes store dense/sparse and fixed/variable attributes, metadata, their dependent axes, and pointers to indices and indptr arrays.

Sparse iterations. Sparse iterations generates iterators over the space composed of a sparse axes array and a body containing statements describing tensor computations and orchestrating data movements. Notably, unlike TACO [54] which only allows the iterator variables to be used as indices to access sparse data structures (e.g. `A[i, j]` where `i` and `j` are iterator variables), SparseTIR supports affine indices (e.g. `A[i * m + j, k]`) and non-affine indices such as integer values loaded from another buffer (e.g. `B[eid[i], j * n + k]`). This enhances the capabilities of the SparseTIR, allowing for more complex operations such as convolution. SparseTIR enables multiple sparse iterations within a single program and even allows for nested sparse iterations within the body of another iteration, enabling branching and decomposing computation.

Figure 3 shows how to define these constructs in SparseTIR for the SpMM operator.[2] In SparseTIR, axes are used to construct both sparse buffers and sparse iterations. This design lets us iterate over a sparse iteration space that is not bound to any sparse buffers.

3.2 Stage I: Coordinate Space Computation

In the first stage of SparseTIR, we define sparse computations within sparse iterations, where we iterate over non-zero elements and access sparse buffers in coordinate space. During this stage, we are able to define program transformations, such as format decomposition and sparse iteration fusion, that manipulate the three types of constructs in SparseTIR.

3.2.1 Format Decomposition. *Format decomposition* is a transformation that decomposes computations for composable formats (introduced in Section 1). The transformation accepts a list of format descriptions and rewrites the IR according to these formats. Figure 5 shows the generated IR for the Sparse Matrix-Matrix multiplication (SpMM) operation after decomposing the computation in the CSR format to a computation in the BSR format, with block size 2 and an ELL format with 2 non-zero columns per row. In addition to SpMM computations on the new formats, another two sparse iterations that copy data from original to new formats are generated, as well. When the sparse matrix to decompose is stationary, we can perform data copying at pre-processing step to avoid the overhead of run-time format conversion.

The information used to create new sparse buffers: `indptr_bsr`, `indices_bsr` and `indices_ell` need to be pre-computed and specified by user as input arguments. Each format decomposition rule in SparseTIR needs to be registered as a function $F : (x, i) \to (x', i')$, where x, i refers to original SparseTIR program and indices/index pointer information, and x', i' are transformed ones. Figure 5 describes the IR transformation from x to x', and the conversion between i to i' need to be implemented by user manually. We have wrapped all format decomposition rules used in this paper as standard APIs, for new composable formats, user can use existing sparse libraries such as Scipy [93] to ease the implementation of indices inference. SparseTIR leaves the flexibility of integrating with existing systems such as Chou et al. [22] for automatic indices inference.

3.2.2 Stage I Schedules. We define two schedule primitives at stage I, `sparse_reorder` and `sparse_fuse`:

Sparse reorder. The order of sparse axes in the sparse iteration influences the order of generated loops in stage II. This primitive enables manipulation of the order of sparse axes.

Sparse fuse. This schedule primitive fuses several iterators in a given sparse iteration into one. It is helpful when we want a single loop rather two nested loops that iterate over all non-zero elements, such as in the SDDMM [63].

Figure 6 shows how stage I schedules transform the IR.

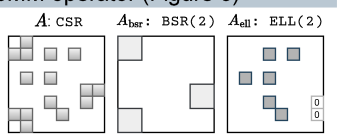
3.3 Stage II: Position Space Computation

In the second stage, SparseTIR introduces loop structures and removes the sparse iteration constructs and restructuring them as

[2]The SparseTIR has round-trip compatibility with Python, and this paper presents only its Python form.

Stage I IR of SpMM operator (Figure 3)

Decompose A to A_{bsr} and A_{ell} with rule `[BSR(2), ELL(2)]`



Generated new axes and sparse buffers

```
# Generated axes for BSR(2)
IO = dense_fixed(m_bsr, "int32")
II = dense_fixed(2, "int32")
JO = sparse_variable(IO, (n_bsr, nnz_bsr),
    (indptr_bsr, indices_bsr), "int32")
JI = dense_fixed(2, "int32")
# Generated axes for ELL(2)
I2 = dense_fixed(m_ell, "int32")
J2 = sparse_fixed(I2, (n_ell, 2), indices_ell, "int32")
# Generated sparse buffers
A_bsr = match_sparse_buffer(a_bsr, [IO, JO, II, JI], "float32")
A_ell = match_sparse_buffer(a_ell, [I2, J2], "float32")
```

Generated sparse iterations

```
# Generated sparse iteration for copying data to BSR(2)
with sp_iter([IO, II, JO, JI], "SSSS", "copy_bsr_2") as [
    oi, ii, jo, ji]:
    A_bsr[io, jo, ii, ji] = A[io * 2 + ii, jo * 2 + ji]
# Generated sparse iteration for copying data to ELL(2)
with sp_iter([I2, J2], "SS", "copy_ell_2") as [i, j]:
    A_ell[i, j] = A[i, j]
# Generated sparse iteration for BSR(2)
with sp_iter([IO, II, JO, JI, K], "SSSSR", "spmm_bsr_2") as [
    io, ii, jo, ji, k]:
    with init():
        C[io * 2 + ii, k] = 0.0
    C[io * 2 + ii, k] = C[io * 2 + ii, k] +\
        A_bsr[io, jo, ii, ji] * B[jo * 2 + ji, k]
# Generated sparse iteration for ELL(2)
with sp_iter([I2, J2, K], "SRS", "spmm_ell_2") as [i, j, k]:
    with init():
        C[i, k] = 0.0
    C[i, k] = C[i, k] + A_ell[i, j] * B[j, k]
```

Figure 5: Format decomposition for SpMM Stage I IR in Figure 3. New axes and sparse buffers are created for decomposed formats BSR and ELL. New sparse iterations are generated to copy data from original to new formats and for computations on these new formats.

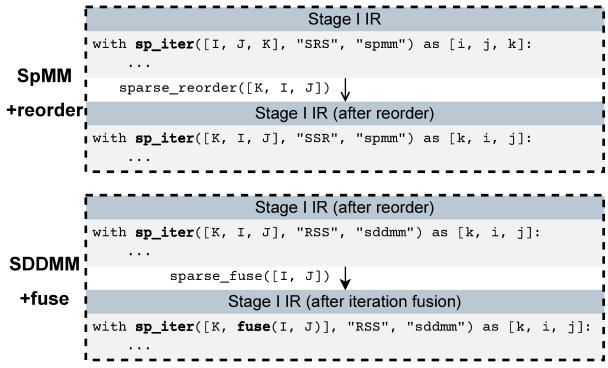


Figure 6: Stage I schedules sequentially applied to stage I IR.

nested loops. To achieve this, we extend TensorIR [34] with sparse buffer support as our stage II IR. Unlike in stage I where we access sparse buffers in *coordinate space*, in stage II access sparse buffers in *position space*, with the "position" referring to an element's non-zero index. The difference between coordinate and position applies to "sparse" dimensions: if the coordinate of the first 4 non-zero elements in a sparse row A is $\{1, 3, 9, 10\}$, the position of coordinate 9 is 2 (assuming the index is 0-based), and we use $A[9]$ to access the element in coordinate space and $A[2]$ to access the element in position space.

3.3.1 Sparse Iteration Lowering. This pass transforms stage I IR to stage II IR. It consists of the following 5 steps.

Step 1: Auxiliary buffer materialization. Pointers to the indices pointer array and indices array are specified as arguments when creating axes. In stage II we need to declare these auxiliary buffers explicitly to access their value when determining loop range and translating coordinates. Figure 7 shows how the materialization works. In addition to auxiliary buffers, we also create hints that indicate the domain of buffer values; these are used for integer set analysis in stage II when performing schedules.

Stage I IR

```
I = dense_fixed(m, "int32")
J = sparse_variable(I, (n, nnz), (j_indptr, j_indices), "int32")
```

Stage II IR

```
I = dense_fixed(m, "int32")
J = sparse_variable(I, (n, nnz), (j_indptr, j_indices), "int32")
J_dense = dense_variable(I, (n, nnz), j_indptr, "int32")
J_indptr = match_sparse_buffer(j_indptr, (I,), "int32")
J_indices = match_sparse_buffer(j_indices, (I, J_dense), "int32")
assume_buffer_domain(J_indptr, [0, nnz])
assume_buffer_domain(J_indices, [0, n1])
```

Figure 7: Example of auxiliary buffer materialization. Sparse buffers storing auxiliary information are created.

Step 2: Nested loop generation. This step restructures sparse iterations in stage I as nested loops in stage II: we emit one loop per axis in the sparse iteration. The generated loops start from 0, and the extent is determined by whether the axis is fixed or variable. They are separated by TensorIR's `block` constructs, which establish boundaries to prevent cross-block loop reordering. Additionally, We add a `block` inside the innermost generated loop and place the body of original sparse iterations inside of it. Figure 8 shows the emitted nested loop structures of different sparse iterations. In the first case, the loops I and J cannot not be reordered in stage II because they are separated by a block; in the second case, we fuse I, J and emit only one loop (`ij`). Currently SparseTIR do not support emitting co-iterations like TACO [54].

Step 3: Coordinate translation. This step rewrites the indices used to access sparse buffers from coordinate space to non-zero position space to bridge the semantic gap between stages I and II. See Figure 9 for an example. Suppose $\{\mathbf{A}_i^{(\text{iter})}\}_{i=1}^M$ is the array of axes used in sparse iterations, $\{\mathbf{v}_i^{(c)}\}_{i=1}^M$ is the array of iterator variables in coordinate space (before translation) and $\{\mathbf{v}_i^{(p)}\}_{i=1}^M$ is the array of loop variables in position space (after translation). For a sparse buffer access to be translated, suppose the buffer is composed of axes $\{\mathbf{A}_j^{(\text{buffer})}\}_{j=1}^N$, and the indices can be viewed as an array of functions $\{\mathbf{I}_j^{(\text{coord})}\}_{j=1}^N$ that maps iterator varibles to indices (for buffer access $B[x+y, z]$ within the sparse iteration where $\mathbf{v}^{(c)} = \{x, y, z\}$, its indices functions $\mathbf{I}^{(\text{coord})}$ should be $\{(x, y, z) \mapsto x + y, (x, y, z) \mapsto z\}$). The coordinate translation can be formulated as an iterative algorithm:

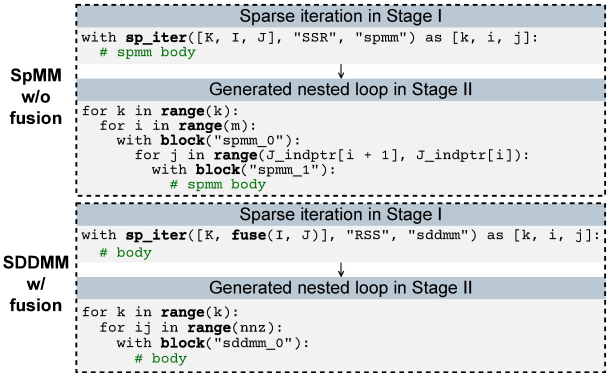


Figure 8: Nested loop generation in sparse iteration lowering. Without fusion, we emit one loop per axis in the sparse iteration; With fusion of i and j, we only emit one loop ij over the fused iteration space.

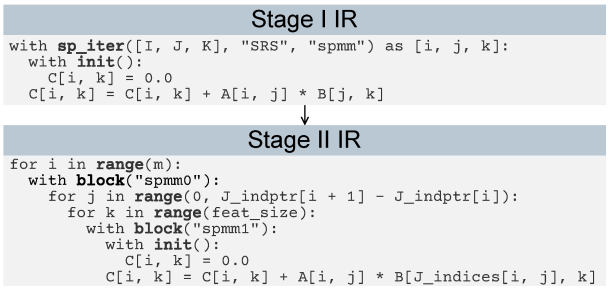


Figure 9: Translation from coordinate space to position space for SpMM operator.

$$\mathbf{p}_j \triangleq f^{-1}(\mathbf{A}^{(\text{buffer})}, j, \{\mathbf{p}\}_1^{j-1}, \mathbf{I}^{(\text{coord})}(\mathbf{c}_1, \ldots, \mathbf{c}_M)), \tag{1}$$

where $\mathbf{c}$ refers the coordinate array corresponding to $\mathbf{v}^{(p)}$ after translation from position space, f and $f^{(-1)}$ are decompress (position to coordinate) and compress (coordinate to position) functions:

$$\mathbf{c}_i \triangleq f(\mathbf{A}^{(\text{iter})}, \{\mathbf{c}\}_1^{i-1}, \mathbf{v}_i^{(p)}) \tag{2}$$

$$f(\mathbf{A}, i, \mathbf{c}, x) \triangleq \begin{cases} x & \mathbf{A}_i\text{: D(ense)} \\ \mathbf{A}_i\_\text{indices}[\mathbf{c}[\text{anc}(\mathbf{A}, i)], x] & \mathbf{A}_i\text{: S(parse)} \end{cases} \tag{3}$$

$$f^{(-1)}(\mathbf{A}, j, \mathbf{p}, x) \triangleq \begin{cases} x & \mathbf{A}_j\text{: D} \\ \text{find}(\mathbf{A}_j\_\text{indices}[\mathbf{p}[\text{anc}(\mathbf{A}, j)], :], x) & \mathbf{A}_j\text{: S,} \end{cases} \tag{4}$$

where the "find" function in the later case of equation 4 refers to searching a given value in sorted array, SparseTIR emits a binary search `block` to search for the index of x in sorted indices array. The "anc" function collects the indices of ancestor(including self) axes of $\mathbf{A}_i$ from its root in axes dependency tree, and $\mathbf{p}[\text{anc}(\mathbf{A}, j)]$ gathers values from $\mathbf{p}$ by ancestors' indices:

$$\text{anc}(\mathbf{A}, i) \triangleq \begin{cases} [i] & \mathbf{A}_i \text{ is root} \\ [\text{anc}(\mathbf{A}, j) : i] & \mathbf{A}_j = \text{parent}(\mathbf{A}_i). \end{cases} \tag{5}$$

Step 5: Read/Write Region Analysis. The buffer read/write region information is necessary for TensorIR's `block` construct. We perform a buffer region analysis pass to collect buffer access information and takes the union of all read/write regions accessed inside each block and annotate them as block attributes.

3.3.2 Stage II Schedules. The stage II schedules are responsible for manipulating loops (`fuse/reorder/split`), moving data across the memory hierarchy (`cache_read/cache_write`), binding loops to physical/logical threads to parallelize them, and using vector/tensor instructions in hardware (`vectorize/tensorize`). As a dialect of TensorIR, we fully support TVM schedule primitives [3] at stage II.

3.4 Stage III: Loop-Level IR

Stage III removes all SparseTIR constructs. It keeps only the nested loop structures whose body includes statements that operate on flattened buffers. This stage should be compatible with loop-level IR in existing tensor compilers. We select TensorIR [34] in Apache TVM [16] as stage III IR to make efficient use of NVIDIA's Tensor Cores, as it fully supports tensorization.

3.4.1 Sparse Buffer Lowering. Sparse buffer lowering removes all axes, flattens all multi-dimensional sparse buffers to 1-dimension, and rewrites memory access to these buffers. Suppose the original sparse buffer A is composed of axes $\{\mathbf{A}_i\}_{i=1}^n$. For memory access $A[x_1, ..., x_n]$, the overall offset after flattening is computed by:

$$\sum_{i=1}^{n} \text{is\_leaf}(\mathbf{A}_i) \times \text{offset}(i) \times \text{stride}(i+1), \tag{6}$$

where $\text{is\_leaf}(\mathbf{A}_i)$ means that if axis $\mathbf{A}_i$ has no dependence in $\{\mathbf{A}_j\}_{j=i+1}^n$, offset and stride are defined as:

$$\text{offset}(i) \triangleq \begin{cases} x_i & \text{is\_root}(\mathbf{A}_i) \\ \mathbf{A}_i\_\text{indptr}[\text{offset}(j)] + x_i & \mathbf{A}_j = \text{parent}(\mathbf{A}_i) \end{cases} \tag{7}$$

$$\text{stride}(i) \triangleq \begin{cases} 1 & i > n \\ \text{nnz}(\text{Tree}(\mathbf{A}_i)) \times \text{stride}(i+1) & \text{is\_root}(\mathbf{A}_i) \\ \text{stride}(i+1) & \text{otherwise,} \end{cases} \tag{8}$$

where $\text{nnz}(\text{Tree}(\mathbf{A}_i))$ refers to the number of non-zero elements of the sparse iteration space composed by the tree with $\mathbf{A}_i$ as its root. Figure 10 shows an example of sparse buffer lowering: sparse buffers A, B, C are flattened. The buffer access $A[i, j]$ is translated to $A[\text{J\_indptr}[i] + j]$ by equation 6.

3.5 Target-Specific Code Generation

SparseTIR re-uses the backend provided by existing tensor compilers for target-specific code generation. SparseTIR emits multiple CUDA kernels for composable formats, which incur extra kernel-launching overhead on the GPU. We insert a horizontal fusion [33, 56] pass to the TVM backend to reduce this overhead.

4 EVALUATION

We now study how composable formats and composable transformations help optimize sparse deep learning workloads in both single-operator and end-to-end settings. In summary, compared

[3] https://tvm.apache.org/docs/reference/api/python/tir.html#tvm.tir.Schedule

Stage II IR

```
# loop structures
with block("spmm1"):
  with init():
    C[i, k] = 0.0
  C[i, k] = C[i, k] + A[i, j] * B[J_indices[i, j], k]
```

Stage III IR

```
# loop structures
with block("spmm1"):
  with init():
    C[i * feat_size + k] = 0.0
  C[i * feat_size + k] = C[i * feat_size + k]
    A[J_indptr[i] + j] * B[J_indices[J_indptr[i] + j] * feat_size + k]
```

Figure 10: Sparse buffer lowering: sparse constructs are totally removed, and memory accesses are flattened to 1-dimension.

to vendor libraries, SparseTIR obtains a 1.20-2.34x speedup on GNN operators and a 1.05-2.98x speedup on sparse attention operators. When used in an end-to-end setting, SparseTIR obtains a 1.08-1.52x speedup on end-to-end GraphSAGE training and a 4.20-40.18x speedup on end-to-end RGCN inference, 0.56-7.44x on Sparse Convolution operators.

4.1 Experiment Setup

Environment. We evaluate all experiments under two different GPU environments: NVIDIA RTX 3070 and NVIDIA Tesla V100.

Baselines. cuSPARSE [24] is NVIDIA's official library for sparse tensor algebra, which includes high-performance implementation of common sparse operators. dgSPARSE [28] is a collection of state-of-the-art sparse kernel implementations for GNNs, which includes GE-SpMM [49], DA-SpMM [25] and PRedS [106]. PyG [36] and DGL [96] are two open-source frameworks that support GNN training and inference. Sputnik [37] is a library for sparsity in Deep Learning. Neither dgSPARSE nor Sputnik uses Tensor Cores. TACO [54] is an open-source sparse tensor compiler. Triton [89] is a tiling-based IR for programming neural networks, and we use its block sparse operator implementation. TorchSparse [86] is a library for point cloud processing, with state-of-the-art sparse convolution implementation.

For SpMM, we select the TACO-generated operator, cuSPARSE 11.7, and dgSPARSE 0.1 as baselines. For SDDMM, we select the TACO-generated operator, cuSPARSE, dgSPARSE and DGL 0.9.1's implementation as baselines. The DGL's SDDMM implementation uses the optimizations proposed in FeatGraph [47]. For end-to-end GNN training, we compare a GraphSAGE model written in PyTorch 1.12 [66] that integrates a SparseTIR-tuned kernel with DGL. For RGCN, we select the Graphiler [103], DGL 0.9.1 and PyG 2.2.0 implementations as our baseline.[4] For sparse transformers, we select Triton[5]'s block-sparse kernel as our baseline. For sparse convolution, we select TorchSparse [6] for comparison. The computation results of all SparseTIR generated kernels have been compared with existing frameworks/libraries to confirm numerical accuracy.

[4]Both DGL and PyG provide several different official implementations of RGCN; we select the best performing among them.
[5]Main branch until commit 0e8590
[6]Main branch until commit 2caf084

Table 1: Statistics of Graphs used in GNN experiments.

Graph	#nodes	#edges	%padding
cora [78]	2,708	10,556	15.9
citeseer [78]	3,327	9,228	13.0
pubmed [78]	19,717	88,651	23.1
ppi [39]	44,906	1,271,274	22.9
ogbn-arxiv [45]	169,343	1,166,243	17.5
ogbn-proteins [45]	132,534	39,561,252	21.6
reddit [39]	232,965	114,615,892	28.6

4.2 Graph Neural Networks

In this section, we evaluate the performance of SparseTIR on GNN workloads. SpMM and SDDMM [63] are two of the most generic operators in GNNs. Table 1 describes the characteristics of graphs used in our evaluation; on the table, %padding refers to the ratio of padded zero elements after we transform the original sparse matrix to composable formats.

4.2.1 SpMM. SpMM is the most generic sparse operator in deep learning, which can be formulated as:

$$Y_{i,k} = \sum_{j=1}^{n} A_{i,j} X_{j,k},$$

where A is a sparse matrix and X, Y are dense matrices. A high-performing SpMM kernel on a GPU requires efficient memory access patterns and load balancing [104]. Runtime load balancing, well studied in SpMM acceleration literature, always incurs runtime overhead. The composable format and composable transformation can help generate kernels that achieve compile-time load balancing and better cache utilization.

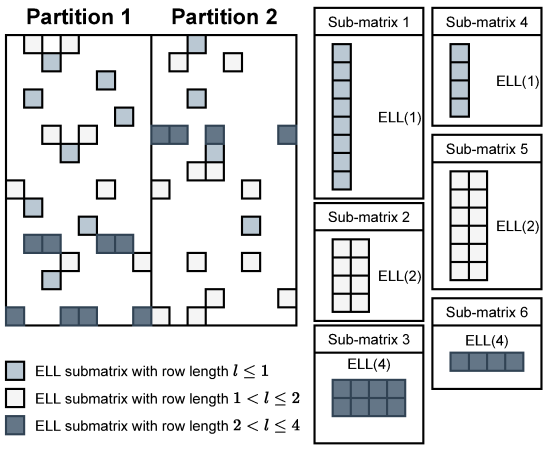


Figure 11: Example of $hyb(2,2)$: the original matrix is decomposed to 6 ELLPACK sub-matrices; elements in partition 1 are stored in sub-matrix 1-3, and elements in partition 2 are stored in sub-matrices 4-6.

We design a parameterized composable format $hyb(c,k)$ for sparse matrix A with two parameters c and k. We partition columns

of the sparse matrix by the given factor c, so that each column partition has width w. For each column partition, we collect the rows with length l that satisfy $2^{i-1} < l \leq 2^i$ to bucket i, and we pad the length of these rows to 2^i; each bucket then forms a sub-matrix with the ELL format. Figure 11 shows a special case, hyb(2, 2).

For bucket i of each column partition, we group each 2^{k-i} rows and map them to a unique thread block in GPUs. The number of non-zero elements in A that are processed by each thread block is 2^k, which is implemented with TVM's `split` and `bind` primitives. We use the schedule proposed in GE-SpMM [49] for each sub-matrix for the remaining dimensions. The column partition in our design is intended to improve cache locality; when processing column partition j, only $B[jw : (j+1)w]$ would be accessed for B. Featgraph [47] proposes to apply column partitions for SpMM on CPUs; however, it does not extend the idea to GPUs. Our bucketing technique was designed to achieve compile-time load balancing. In practice, we searches for the best c over $\{1, 2, 4, 8, 16\}$ and let $k = \lceil \log_2 \frac{nnz}{n} \rceil$, which generally works well.

We evaluate the SpMM written in SparseTIR with and without the proposed *hyb* format on real-world GNN datasets for both V100 and RTX3070. We measure the geometric mean speedup of different SpMM implementations against cuSPARSE for feature size $d \in \{32, 64, 128, 256, 512\}$. Figure 13 shows our results. The SparseTIR kernel on *hyb* format obtains a 1.22-2.34x speedup on V100 and a 1.20-1.91x speedup on RTX 3070 compared to cuSPARSE. We also achieve consistently better performance than state-of-the-art open source sparse libraries dgSPARSE and Sputnik, and TACO scheduled kernels [79]. Though TACO also explores compile-time load balancing, it does not support caching the partially aggregated result in registers, which is critical to GPU performance, and the irregularity of the CSR format limits the application of loop unrolling. SparseTIR perform these optimizations in stage II schedules.

Importance of composable formats. We evaluate the SparseTIR kernel without format decomposition (see SparseTIR(no-hyb) in the figure). Results suggest that the SparseTIR kernel without format decomposition and per-format scheduling performs generally worse: ogbn-arxiv is a citation network graph whose degrees obey power-law distribution, and our designed format can perform significantly better because of more efficient load balancing. Notably, though padded zeros in our proposed composable format slightly increase FLOPs as shown in Table 1, the runtime of SparseTIR generated kernels on composable format is still faster because of better scheduling. The degree distribution of the ogbn-proteins graph is centralized, and the benefit of using a hybrid format is compensated for the extra overhead introduced by padding. To evaluate the effect of column partitioning, we fix the feature size to 128 and measure several kernel metrics generated by SparseTIR on a Reddit dataset under a different column partition setting. Figure 12 shows the results; L1 and L2's cache hit rates improve as we increase the number of column partitions. However, more partitions will increase the required memory transactions of the kernel because we will need to update the results matrix c times if the number of partitions is c. As a result, the benefit of column partitioning saturates as we increase the number of partitions.

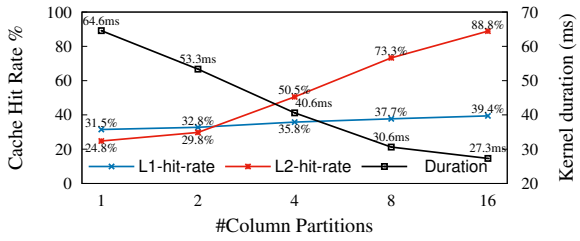


Figure 12: The kernel duration and L1/L2 hit-rate of SparseTIR SpMM kernels under different column partitions.

4.2.2 SDDMM. SDDMM can be formulated as the following:

$$B_{i,j} = \sum_{k=1}^{d} A_{i,j} X_{i,k} Y_{k,j},$$

where A and B are two sparse matrices that share a sparse structure, X, Y are dense matrices, and d is the feature size. In SDDMM, the computation per (i, j) is independent, and the workload per position is the same, so we need not worry about load balancing issues if we parallelize the computation by each non-zero (i, j). The `sparse_fuse` schedule primitive in stage I introduced in Section 3.2.2 helps us iterate over non-zero (i, j) directly instead of first iterating over i and then iterating over non-zero j for each i.

PRedS [106] is the state-of-the-art open-source SDDMM implementation, which optimizes SDDMM in two ways. First, it uses vectorized load/store intrinsics in CUDA, such as `float4`/`float2`, which improves memory throughput. Second, it performs the reduction in two stages: (1) *intra-group reduction*, which computes the reduction inside each group independently, and (2) *inter-group reduction*, which summarizes the reduction result per group. We formulate the optimization in PRedS as composable transformations in SparseTIR with `vectorize` and `rfactor` [84] schedule primitives at stage II, and we generalize the parameters, such as group size, vector length and number of workloads per thread block, as tunable parameters.

Figure 14 shows the geometric mean speedup of different SDDMM implementations vs our baseline for feature size $d \in \{32, 64, 128, 256, 512\}$. We do not use composable formats in SDDMM. The baseline we select is DGL's SDDMM implementation, which uses the optimization proposed in Featgraph [47]. cuSPARSE and Sputnik's SDDMM implementations are not optimized for highly sparse matrices such as graphs and thus achieve very low performance. We obtain generally better performance than dgSPARSE [28], which implements the PRedS [106] algorithm, because of the parameterized scheduling space. SparseTIR significantly outperforms the DGL baseline and the TACO scheduled kernel because these implementations do not include two-stage reduction and vectorized load/store.

Importance of composable transformations. The provenance graph data structure in TACO does not support multiple branches, thus we cannot perform schedules such as `rfactor` at this level. The composable transformation design of SparseTIR enables us to apply such schedules at lower stages.

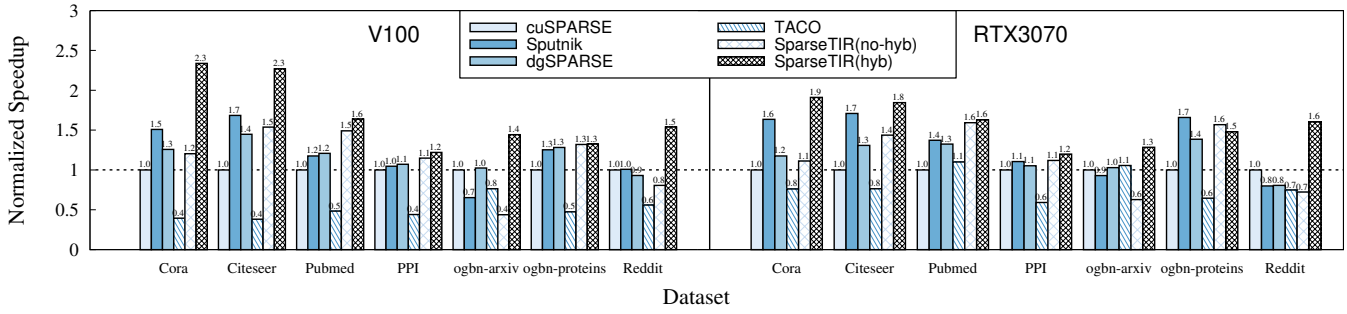


Figure 13: Normalized speedup against cuSPARSE for SpMM. SparseTIR consistently outperforms vendor libraries and TACO. Comparing SparseTIR(no-hyb) and SparseTIR(hyb) demonstrates the importance of format composability.

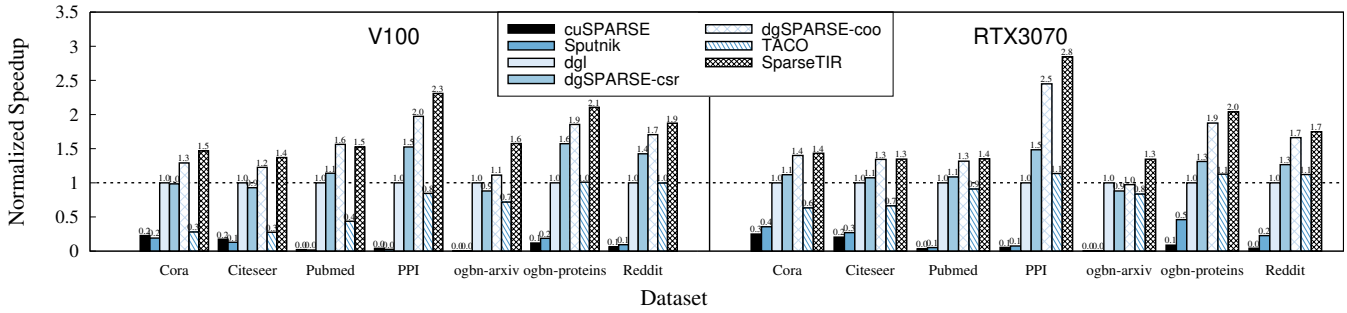


Figure 14: Normalized speedup against Featgraph for SDDMM. SparseTIR beats the state-of-the-art vendor library dgSPARSE on average by parametrizing scheduling space.

4.2.3 End-to-end GraphSAGE Training. We also integrate SparseTIR-generated SpMM operators in the GraphSAGE [39] model written in PyTorch and compare the end-to-end speedup to DGL. Figure 15 shows that we obtain a 1.18-1.52x speedup on V100 and a 1.08-1.47x speedup on RTX 3070 [7].

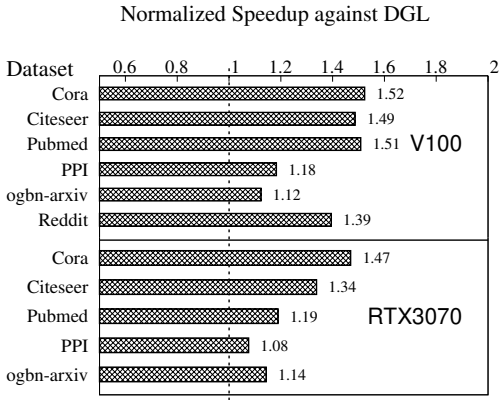


Figure 15: Normalized speedup of PyTorch+SparseTIR against DGL on end-to-end GraphSAGE training.

4.3 Sparsity in Transformers

Sparsity in Transformers comes from (1) sparse attentions [6, 15, 20], and (2) sparsity in network weights after pruning [55, 76]. We evaluate SparseTIR generated kernel in both cases[8].

4.3.1 Sparse Attention. Sparse transformers reduce the complexity of Transformers by making the attention matrix sparse. The key operator in Sparse Transformers is still SpMM and SDDMM, but unlike GNNs whose sparse matrices are provided by graph structures, the sparse matrices used in sparse attentions are mostly manually designed and have a block-sparse pattern to better utilize tensor cores in modern GPUs. We select two examples: Longformer [6] and Pixelated Butterfly Transformer [15], whose sparse structures are band matrix and butterfly matrix [65], respectively. We implement the batched-SpMM and batched-SDDMM operators for both CSR and BSR formats. For BSR operators, we use the `tensorize` primitive during stage II IR schedules to use tensorized instructions in CUDA. Figure 16 shows different implementations' speedup against Triton's [89] block-sparse operator. We fix the matrix size to 4096×4096, batch(head) size to 12, band size to 256, and feature size per head to 64. Results show that SparseTIR-BSR obtains a 1.05-1.59x speedup on multi-head SpMM and a 1.50-2.98x speedup on multi-head SDDMM.

[7] Reddit result is not reported on RTX 3070 because of Out-Of-Memory issue.

[8] In this section, we use half-precision data type for all operators to use Tensor Cores.

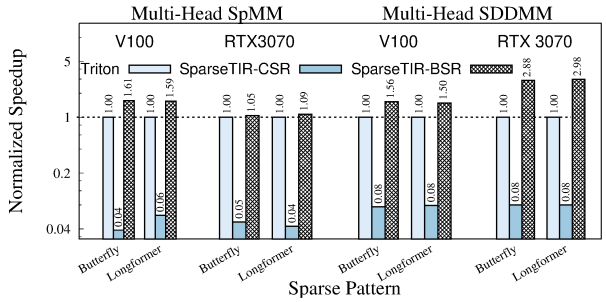


Figure 16: Normalized speedup against Triton on sparse transformer operators.

4.3.2 Sparse Weight (Network Pruning). Network pruning [40] is another source of sparsity in Transformers. Pruning can significantly reduce the number of model parameters at the cost of negligible performance (accuracy) loss by making the weights sparse. PruneBERT [55, 76] applies pruning to Transformers, and we evaluate SparseTIR's performance on PruneBERT in both structured pruning and unstructured pruning settings.

Structured Pruning. Structured Pruning prunes groups of weights together at the channel or block level to speed up execution. Block pruning [55] is an example of structured pruning on Transformers where network weights are pruned to block-sparse format, the operator used in block-pruned Transformer is SpMM. We extract all SpMM operators in a block-pruned model[9] with block size 32 and average weight sparsity 93% for the benchmark. We fix the batch size to 1 and the sequence length to 512. Figure 17 shows the performance of SparseTIR kernels, Triton's BSRMM, and cuBLAS on these operators. The block sparse weights in the block-pruned model have many all-zero rows, and we propose to use doubly-compressed BSR (DBSR, inspired by doubly compressed sparse row (DCSR) format [13]) format to skip zero rows. The results show that SparseTIR kernel on DBSR format can consistently outperform SparseTIR kernel on BSR format, and achieve better with Triton's BSRMM implementation.

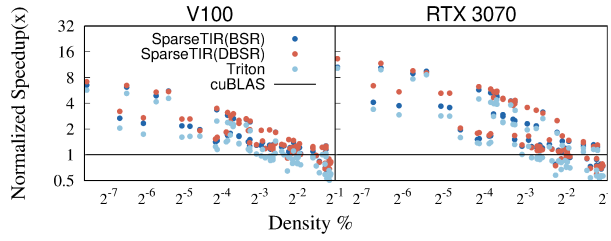


Figure 17: Normalized speedup against cuBLAS for operators extracted from block-pruned transformers. The X-axis refers to the weight density in the SpMM operator, and Y-axis refers to the normalized Speedup against cuBLAS implementation which uses a dense matrix for sparse weight.

[9] https://huggingface.co/madlag/bert-base-uncased-squad1.1-block-sparse-0.07-v1

Unstructured Pruning. Unstructured pruning does not pose any constraints on the format of pruned weights, and the pruned weight matrices are typically stored in CSR format. Unstructured pruned model is known to be hard to optimize because of irregular computation, and directly converting them to BSR format would introduce too much fragmentation inside blocks. We use the SR-BCRS format proposed in Magicube [58] to alleviate the issue. Figure 18 explains how to represent SR-BCRS(t, g) and corresponding SpMM schedules in SparseTIR: the matrix is firstly divided into many $t \times 1$ tiles, and we omit tiles whose elements are all zero. The non-zero tiles inside the same rows are grouped by a factor of g, and we pad the tailing groups with zero tiles. Sparse matrices in SR-BCRS format can be composed by 4 axes in SparseTIR. When performing SpMM on SR-BCRS, we can load a group of tiles in A and corresponding rows in X to local registers and use Tensor Cores in GPU (or Matrix-Multiply Units(MXU) in TPU [51], equivalently) to compute their multiplication results, these schedules can be described as `cache_read/write` and `tensorize` primitives at stage-II in SparseTIR. Compared to BSR, the SR-BCRS format greatly reduces intra-block fragmentation: the non-zero ratio lower bound in SR-BCRS(t, g) is $1/t$, while BSR with block size b has a lower bound of $1/b^2$.

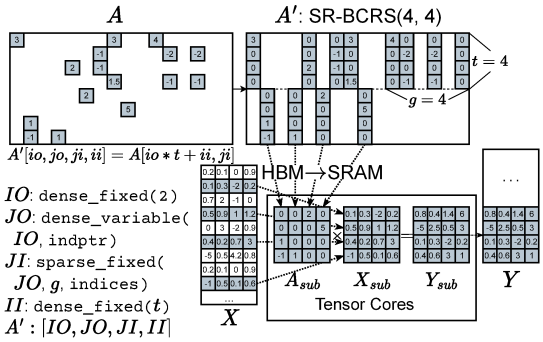


Figure 18: Conversion from unstructured sparse matrix to SR-BCRS(t, g), and SpMM schedule on the it.

We extract all SpMM operators in a movement-pruned model[10] with average weight sparsity of 94% for benchmark. Figure 19 shows the performance of SparseTIR on SR-BCRS$(8, 32)$[11], BSR format with block size 32, and vendor libraries cuBLAS and cuSPARSE's CSRMM. We set the batch size to 1 and the sequence length to 512. We do not compare with Triton because it has no native implementation of SpMM on SR-BCRS. SparseTIR on SR-BCRS beats SparseTIR on BSR in most of the settings except for density $\geq 2^{-3}$, in which case both transformed sparse matrices have a density close to 1. cuSPARSE's CSRMM can only beat cuBLAS' GeMM when weight density is extremely low (e.g., $\leq 2^{-6}$).

4.4 Relational Gather-Matmul-Scatter

Relational Gather-Matmul-Scatter (RGMS for short) is an emerging sparse operator which can be expressed as follows:

[10] https://huggingface.co/huggingface/prunebert-base-uncased-6-finepruned-w-distil-squad

[11] To use m8n32k16 MMA instructions in GPU.

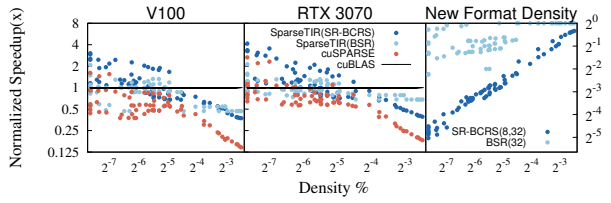


Figure 19: Normalized speedup aginst cuBLAS for operators extracted from unstructured pruned transformers, and the weight density in new format vs original weight.

Table 2: Statistics of Heterogeneous Graphs used in RGCN.

Graph	#nodes	#edges	#etypes	%padding
AIFB [72]	7,262	48,810	45	17.9
MUTAG [72]	27,163	148,100	46	8.0
BGS [72]	94,806	672,884	96	4.3
ogbl-biokg [45]	93,773	4,762,678	51	4.2
AM [72]	1,885,136	5,668,682	96	10.8

$$Y_{i,l} = \sum_{r=1}^{R} \sum_{j=1}^{n} \sum_{k=1}^{d_{in}} A_{r,i,j} X_{j,k} W_{r,k,l},$$

where A is a 3D sparse matrix, whose leading dimension size is R, denoting number of relations. For each relation, the last two dimensions of A form a unique 2D sparse matrix. X is a 2D feature matrix and W is a 3D weight matrix whose leading dimension size is also R. For each relation, the last two dimensions of W form a unique 2D dense weight matrix. The scheduling for the RGMS operator is complicated because we need to consider (1) load balancing and (2) the utilization of Tensor Cores. Until now, no sparse library implements this kernel.

4.4.1 Relational Graph Convolution Network. RGCN [77] is a generalization of GCN model to heterogeneous graphs (graphs with multiple relations/edge types). The operator used in RGCN is RGMS, where A_r refers to the adjacency matrix corresponding to subgraph whose edge type is r, and W_i refers to the weight matrix corresponding to edge type r. Table 2 introduces the characteristics of heterogeneous graphs used in RGCN evaluation; in the table, #etypes refers to the number of edge types (also known as "relations") in the heterogeneous graph, %padding refers to the ratio of padded zero elements after we transform the original sparse matrix with composable formats. Existing GNN libraries implement RGMS operator in a two-stage approach:

$$T_{r,j,l} = \sum_{k=1}^{d_{in}} X_{j,k} W_{r,k,l}, \tag{9}$$

$$Y_{i,l} = \sum_{r=1}^{R} \sum_{j=1}^{n} A_{r,i,j} T_{r,j,l}, \tag{10}$$

where the first stage fuses gathering and matrix multiplication, and the second stage performs scattering. Such implementation materializes the intermediate result T on HBM, which incurs a lot of GPU memory consumption. In SparseTIR we fuses the two stage into a single operator: we generalize the *hyb* format proposed in Figure 11 to 3-dimensional so that 2D sparse matrix corresponding to each relation is decomposed to $hyb(1, 5)$ formats. Figure 21 explain the scheduling of RGMS operator on 3D *hyb* in SparseTIR: for each ELL matrix A^{rk} (r refers to edge type and k refers to bucket index), we pin its corresponding weight matrix W^r in SRAM and gather related rows of X from HBM to SRAM, then perform partial matrix multiplication with Tensor Cores and scatter results to Y. Note that the matrix multiplication and intra-group scatter are all performed inside SRAM. Such design reduces the overhead of data copy between SRAM and HBM for intermediate matrix T. We evaluate end-to-end RGCN inference (feature size: 32) and Figure 20 shows results: SparseTIR(*hyb*+TC) can significantly improve previous state-of-the-art GNN compiler Graphiler [103] by 4.2-40.2x in different settings. By comparing SparseTIR(naive), SparseTIR(*hyb*) and SparseTIR(*hyb*+TC) we show that both composable formats and composable transformations (which enables Tensorization) matter: even though *hyb* increases FLOPs by padding zeros (as shown in Table 2), it still makes the kernel faster by 2-4.4x because of better load-balancing. SparseTIR's generated fused kernel can also greatly reduce GPU memory footprint because we do not explicitly stores T in HBM, with fragments of T consumed immediately after produced in SRAM. SparseTIR(*hyb*+TC) consumes more GPU memory than SparseTIR(naive) and SparseTIR(*hyb*) because of the half-precision/single-precision data type conversion.

4.4.2 Sparse Convolution. Sparse Convolution [23] is widely used in 3D cloud point data. We found that the Sparse Convolution operator is a special form of RGMS, and Figure 22 illustrates the equivalence: each relative offset inside the convolution kernel can be viewed as a relation in RGMS. For each relation, the mapping between non-zero elements in feature map of previous layer to non-zero elements in feature of next layer forms a bipartite graph which can be viewed as a 2D sparse matrix whose number of non-zero elements per row is no greater than 1.

We extract all of the Sparse Convolution operators in MinkowskiNet [23] on SemanticKitti dataset [5] for benchmark, and evaluate SparseTIR's RGMS kernel[12]. Figure 23 shows our normalized speedup against state-of-the-art TorchSparse [86] library. Unlike the SparseTIR's schedule in Figure 21, TorchSparse does not fuse Gather-Matmul-Scatter on chip. Instead, it explicit materializes T and uses coarse-grained cuBLAS operators rather than Tensor-Core level instructions for matrix multiplication[13]. SparseTIR's RGMS can outperform TorchSparse for most of the operators because of less HBM/SRAM data exchange as mentioned before. However, for large channel size (> 128), SparseTIR's RGMS cannot beat TorchSparse because matrix multiplication overhead become dominant (The FLOPs of Matmul is quadratic to channel size while the FLOPs of Gather and Scatter is linear to channel size) and cuBLAS is better optimized than SparseTIR's RGMS for large channel.

[12] We don't need to use composable formats for Sparse Convolution because the sparse matrix for each relation is already an $ELL(1)$.
[13] It's not necessary to use adaptive matrix multiplication grouping when using fine-grained Tensor-Core instructions.

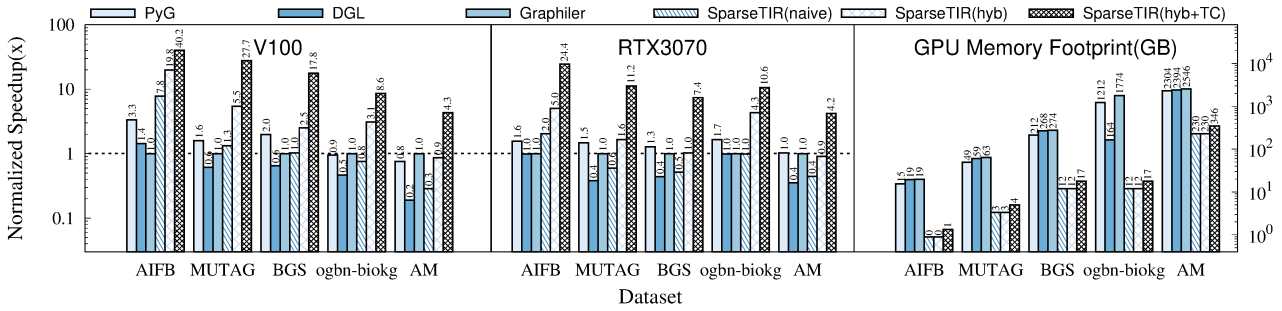


Figure 20: Normalized RGCN inference speedup against Graphiler and GPU Memory Footprint. SparseTIR(*hyb*+TC) uses schedule proposed in Figure 21, SparseTIR(*hyb*) uses composable format but use CUDA Cores instead of Tensor Cores for on-chip Matrix Multiplication, SparseTIR(naive) uses neither composable formats nor Tensor Cores.

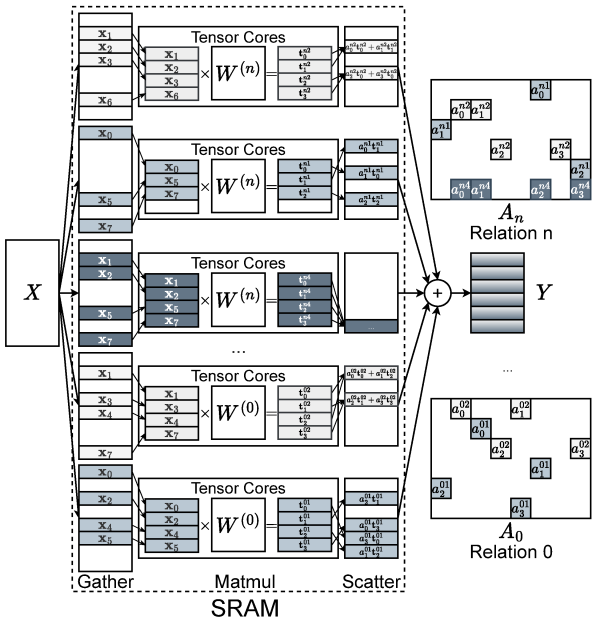


Figure 21: Schedule of RGMS operator in SparseTIR. Composable formats *hyb* are used for load balancing.

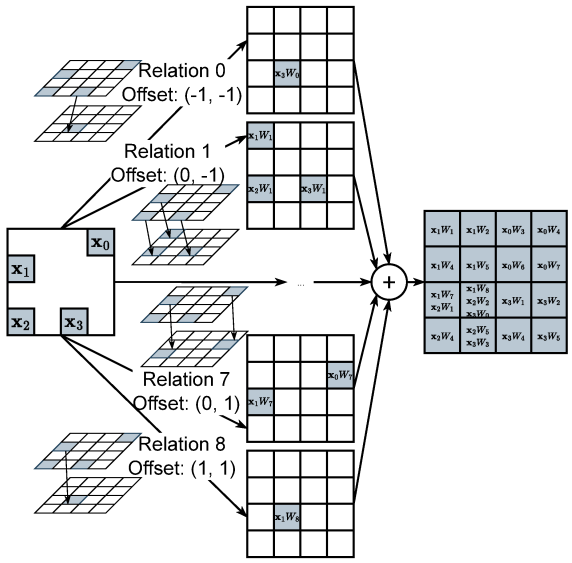


Figure 22: Equivalence of RGMS and Sparse Convolution, each relative offset inside the convolution kernel forms a relation in RGMS. The equivalence also holds in 3D setting.

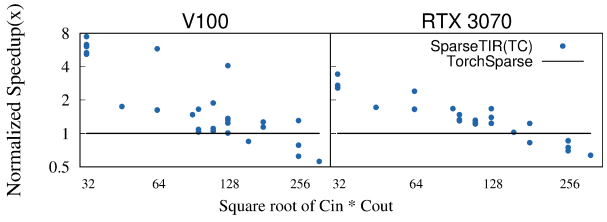


Figure 23: Normalized speedup against TorchSparse for Sparse Convolution. The X-axis refers to square root of input channel and output channel: $\sqrt{C_{in}C_{out}}$, and the Y-axis refers to speedup against TorchSparse.

5 RELATED WORK

Tensor and deep learning compilers. Halide [70] and TVM [16, 17] are tensor compilers that decouple kernel description and schedules for dense computation. XLA [27] and Relay [73] proposed computational-graph-level abstractions for deep learning, where we can apply optimizations such as *kernel fusion* and *graph substitution* [50]. However, these compilers have limited support for representing and optimizing sparse operators, impeding the wider deployment of sparse deep learning workloads such as GNNs. TensorIR [34] is TVM's new tensor-level programming abstraction for automatic tensorization. Triton [89] is an intermediate language that offers tile-level operations and optimizations, FreeTensor [87] is a compiler for irregular tensor programs with loop-based programming model. These IRs could serve as stage-III IR for SparseTIR.

Sparse compilers. MT1 [8–12], SIPR [69], Ironman [60] and Ahmed et al. [2] introduces the idea of compiling kernels for a given sparse

data structure and a kernel description. TACO [21, 53, 54] proposes sparse format abstractions and a merge lattices-based code generation routine. Senanayake et al. [79] propose a sparse-iteration space transformation framework for scheduling sparse operators. Chou et al. [22] introduce an approach for generating efficient kernels for sparse format conversion. Henry et al. [41] generalize TACO to sparse array programming. The format abstraction and IR design of SparseTIR are insipred by TACO and earlier work, with a focus on Deep Learning operators. Sympiler [18] builds a symbolic inspector to analyze sparse structure at compile-time and generates efficient code. Parsy [19] generalize the idea to support parallelization. SPF [83] proposes a inspector-executor framework compatible with polyhedral transformations. Mohammadi et al. [62] proposes data dependence simplication algorithm for compiler generated inspectors. Like composable formats in SparseTIR, these compilers can utilize sparse structures for acceleration. Taichi [46] decouple data structure and kernel description for physics simulation programming; its compiler optimizations focus on spatial sparse data, unsuitable for DL. Tiramisu [4] supports automatic selection of dense/sparse kernels at computational graph-level. However, it lacks tensor-level sparse code generation. COMET [88] and MLIR Sparse Dialect [7] are two MLIR dialects that explore composable IR design for sparse tensor algebra. Both treat sparse tensors with format annotation as first-class members in the IR; however, neither considers decomposable formats. CoRA [33] proposes a compiler infrastructure for ragged tensors [26]: a special form of sparse tensors. The operation splitting in CoRA is a special case of format decomposition in SparseTIR. SparTA [110] proposes abstractions for model sparsity; its annotation is still dense and thus not applicable to highly sparse matrices used in GNNs. SparseLNR [29] proposes *branched iteration graph* to support factoring reductions and loop-fusion for sparse tensor algebra, these schedules can be formulated as stage-I schedules in SparseTIR as we support branches in the IR.

GNN systems and compilers. PyG [36] and DGL [96] propose programming interfaces for the programming message-passing [38] modules in GNN models. Both frameworks use vendor libraries and handwritten operators to accelerate specific message-passing patterns. Featgraph [47] optimizes generic GNN operators with TVM. However, it fails to support more operators because TVM lacks sparsity support. FusedMM [71] fuses SDDMM and SpMM operators, thus accelerating GNN training and saving GPU memory footprint. FusedMM can be described and optimized in SparseTIR. Seastar [102] and Graphiler [103] compile user-defined message-passing functions to their intermediate representations (IR) and then optimize the IR and emit template-based, target-specific code: these templates still have limited expressiveness and cannot consider a wide range of the optimization space. SparseTIR could serve as a backend for these GNN compilers. GNNAdvisor [100] proposes a CUDA template for GNN computations and uses graph characteristics to guide the performance tuning of GNN training. QGTC [99] and TC-GNN [98] explore accelerating GNNs with TensorCores. Notably, the "condensing" technique proposed in TC-GNN is equivalent to SpMM on SR-BCRS format as shown in Section 4.3.2. The contribution of these papers is orthogonal to SparseTIR.

Sparse kernel optimizations. Merge-SpMM [104], ASpT [43], GE-SpMM [49], Sputnik [37] and DA-SpMM [25] explore different schedule spaces for SpMM optimization on GPUs. We carefully examined the optimizations suggested in theses papers and propose a composable abstraction to unify them. OSKI [94] is a library for auto-tuning sparse operators, with a focus on optimizing operators on cache-based, super-scalar architectures such as CPUs. However, OSKI do not support customizing sparse operators.

Sparse format optimizations. Pichon et al. [67] propose to reorder rows and columns in 2D sparse matrices to increase the block granularity of sparse matrices. Li et al. [57] study the problem of reordering sparse matrices to improve cache locality of operators on them. Mehrabi et al. [61] and Wang et al. [100] propose to reorder rows and columns of sparse matrices to accelerate SpMM on GPUs. These algorithms can act as pre-processing steps in SparseTIR to discover efficient composable formats.

Hardware-efficient algorithms. There have been a growing trend of sparsity in Deep Learning [42]. To make better use of underlying hardware, researchers propose pruning algorithms with block-sparsity [55] and bank-sparsity [14, 112] to utilize acceleration units in GPUs, and ES-SpMM [59] for load balancing. SparseTIR's composable abstractions can help researchers explore more complex sparse patterns with ideal performance on modern hardware.

6 FUTURE WORK

Automatic scheduling. SparseTIR still requires users to specify schedule templates like they do for the first-generation of Halide and TVM. The Halide auto-scheduler [1], FlexTensor [111], Ansor [109] and Meta-scheduler [80] have been proposed to automatically generate schedule templates for dense tensor compilers. We expect these techniques would also prove helpful for sparse compilation. Searching for the optimal schedule is time consuming, Ahrens et al. [3] propose an asymptotic cost model for sparse tensor algebra to narrow the schedule space of sparse kernels, which could also benefit our work.

Automatic format decomposition. In this paper we explore only manually designed format decomposition rules. We leave automatic format selection [11, 12] and decomposition for future work.

Dynamic sparsity. Some models [32, 68, 81] exhibit dynamic sparsity, where the position of non-zero elements changes overtime thus searching for best schedule for each matrix become impractical. DietCode [108] proposes *shape-generic* search space, micro-kernel based cost model and a lightweight dispatcher to dispatch kernel at runtime, the idea is also applicable to sparse tensor programs.

Integration with graph-level IR. SparseTIR models only tensor-level sparsity, we plan to extend the sparse attributes in SparseTIR to graph-level IRs like XLA [27] and Relay [73].

7 CONCLUSION

We introduce SparseTIR, a composable abstraction for sparse operators in deep learning. Its key innovation is the use of composable formats and composable transformations, and together they form the parameter search space for performance tuning. Evaluations on generic sparse deep learning show that SparseTIR achieves significant performance improvements over existing vendor libraries and frameworks.

ACKNOWLEDGMENTS

We thank all anonymous ASPLOS reviewers for their constructive comments. We thank Siyuan Feng, Bohan Hou and Wuwei Lin for discussions on tensorization and IR design, Sandy Kaplan for help on paper writing, Zhijian Liu for providing Sparse Convolution benchmarks, Joel Emer, Yuwei Hu, Jie Liu, Steven S. Lyubomirsky, Fredrik Kjolstad, Ye Tian, Zhiqiang Xie, and Zhongyuan Zhao for feedbacks on the paper. This work was supported in part by the Center for Intelligent Storage and Processing in Memory (CRISP), a Semiconductor Research Corporation (SRC) program co-sponsored by DARPA. It was also supported by the Real Time Machine Learning (RTML) NSF and DARPA program, and the NSF award CCF-1518703, CNS-2211882. The opinions and conclusions in this paper do not reflect the views of these funding agencies.

A PROGRAMMING INTERFACE FOR COMPOSABLE FORMATS

This section further explains the programming interface for composable formats and the format decomposition pass introduced in §3.2.1, SparseTIR provide two APIs for composable formats:

FormatRewriteRule is a class for a sparse format rewriting rule description, its input include: the name of format rewrite rule, the sparse buffer to rewrite, a SparseTIR description of new format, the mapping from original axes to new axes, and the index mapping f and inverse index mapping f^{-1} between original sparse buffer A and the transformed sparse buffer A': $A[\mathbf{I}] = A'[f(\mathbf{I})], A[f^{-1}(\mathbf{I'})] = A'[\mathbf{I'}]$, both f and f^{-1} need to be affine maps written in Python's lambda functions.

decompose_format is a function that accepts a list of format rewrite rules and an SparseTIR program as input and performs the format decomposition pass on the given SparseTIR program.

Below is an example illustrating how to use the two APIs to compose `ELL(2)` and `BSR(2)` rewrite rules and perform format decomposition in Figure 5:

```
@T.prim_func
def spmm(
    a: T.handle, b: T.handle, c: T.handle,
    indptr: T.handle, indices: T.handle,
    m: T.int32, n: T.int32, nnz: T.int32, feat_size: T.int32
) -> None:
    I = T.dense_fixed(m, idtype="int32")
    J = T.sparse_variable(
        I, (n, nnz), (indptr, indices), idtype="int32")
    J_ = T.dense_fixed(n, idtype="int32")
    K = T.dense_fixed(feat_size, idtype="int32")
    A = T.match_sparse_buffer(a, (I, J), "float32")
    B = T.match_sparse_buffer(b, (J_, K), "float32")
    C = T.match_sparse_buffer(c, (I, K), "float32")
    with T.sp_iter([I, J, K], "SRS", "csrmm") as [i, j, k]:
        with T.init():
            C[i, k] = 0.0
        C[i, k] = C[i, k] + A[i, j] * B[j, k]


def BSR(block_size: int):
    # block_size: the block size in BSR format.
    @T.prim_func
    def bsr_desc(
        a: T.handle,
        indptr: T.handle, indices: T.handle,
        m: T.int32, n: T.int32, nnz: T.int32
    ) -> None:
        IO = T.dense_fixed(m, idtype="int32")
        JO = T.sparse_variable(
            IO, (n, nnz), (indptr, indices), idtype="int32")
        II = T.dense_fixed(block_size, idtype="int32")
        JI = T.dense_fixed(block_size, idtype="int32")
        A = T.match_sparse_buffer(a, (IO, JO, II, JI), "float32")
        pass

    return FormatRewriteRule(
        "bsr_{}".format(str(block_size)),
        bsr_desc,
        ["A"], ["I", "J"], ["IO", "JO", "II", "JI"],
        {"I": ["IO", "II"], "J": ["JO", "JI"]},
        lambda i, j:
            return (i // block_size, j // block_size,
                    i % block_size, j % block_size)
        lambda io, jo, ii, ji:
            return io * block_size + ii, jo * block_size + ji
    )

def ELL(nnz_cols: int):
    # nnz_cols: number of non-zero columns per row in ELL format.
    @T.prim_func
    def ell(
        a: T.handle,
        indices: T.handle,
        m: T.int32, n: T.int32,
    ) -> None:
        I2 = T.dense_fixed(m, idtype="int32")
        J2 = T.sparse_fixed(
            I2, (n, nnz_cols), indices, idtype="int32")
        A = T.match_sparse_buffer(a, (I2, J2), "float32")
        pass

    return FormatRewriteRule(
        "ell_".format(str(nnz_cols)),
        ell_desc,
        ["A"], ["I", "J"], ["I2", "J2"],
        {"I": ["I2"], "J": ["J2"]},
        lambda i, j: return i, j
        lambda i2, j2: return i2, j2
    )

composable_format = [BSR(2), ELL(2)]
spmm_hybrid = decompose_format(spmm, composable_format)
```

where the prefix `T` is used to prevent name conflicts with keywords in Python. Note that format conversion is a special case of format decomposition where we only put one `FormatRewriteRule` in the list of composable formats.

B ARTIFACT APPENDIX

B.1 Abstract

This artifact includes scripts and dependencies for reproducing all experiments in the paper. We require a host with x86_64 CPU and NVIDIA GPUs with Turing or later architectures to run the artifact. The SparseTIR compiler is a submodule in the artifact, which is implemented in C++ and Python. The benchmarking is mainly written in Python. We modify the source code of some old dependencies to make sure they are compatible with the software version specified in the Dockerfile. We provide a docker image for users to run benchmarks inside the container, and scripts to generate tables and figures for comparison.

B.2 Artifact Checklist

- **Data set:** OGB, SemanticKITTI, DGL built-in datasets.
- **Run-time environment:** NVIDIA Container Toolkit.
- **Hardware:** NVIDIA GPUs with Turing/Ampere/Hopper architecture.
- **Execution:** All kernels being profiled are executed in GPUs, some data pre-processing are performed on CPUs.
- **Metrics:** Execution time, GPU memory footprint.
- **Output:** Execution time/GPU memory usage tables, and figures.

- **Experiments:** SpMM, SDDMM, GraphSAGE end-to-end training, Sparse Transformer operators, 3D Sparse Convolution, Relational Graph Convolutional Networks inference.
- **How much disk space required (approximately)?:** 55GB.
- **How much time is needed to prepare workflow (approximately)?:** 2 hour for building docker container.
- **How much time is needed to complete experiments (approximately)?:** 10 hours.
- **Publicly available?:** Yes.
- **Code licenses (if publicly available)?:** The SparseTIR-artifact is distributed under The MIT license and the SparseTIR compiler is released under the Apache License, v2.0.
- **Archived (provide DOI)?:** https://doi.org/10.5281/zenodo.7643745

B.3 Description

B.3.1 How to Access. The artifact [105] is available on Github: https://github.com/uwsampl/sparsetir-artifact and Zenodo: https://doi.org/10.5281/zenodo.7643745. Which includes the installation scripts for all dependencies and benchmark scripts to reproduce results. The SparseTIR compiler, which is available at https://github.com/uwsampl/sparsetir, has been incorporated as a submodule of the artifact.

B.3.2 Hardware Dependencies. We conduct experiments on two machines, one with NVIDIA RTX 3070 GPU and another with NVIDIA Tesla V100 GPU, both of them are equipped with x86_64 CPUs. Other NVIDIA GPUs with Turing, Ampere, or Hopper architecture should also work. A GPU with memory greater than or equal to 16GB is enough to reproduce all results, otherwise, users might encounter an Out-Of-Memory issue for relatively large datasets like *Reddit* on end-to-end GraphSAGE training.

B.3.3 Software Dependencies. We create a Docker image for this artifact, enabling users to run all experiments on a platform that meets the installation requirements of the NVIDIA Container Toolkit.

B.3.4 Datasets. For GNN-related experiments, we use Open Graph Benchmark [45] and built-in datasets provided by DGL [96], for Sparse Convolution, we use SemanticKITTI dataset [5], for PrunedBERT, we use models publicly available in HuggingFace [101].

B.4 Installation

To install the artifact, users can either clone the repository and build the artifact by themselves:

```
git clone https://github.com/uwsampl/
    sparsetir-artifact.git --recursive
cd sparsetir-artifact
docker build -t sparsetir .
```

or pull the pre-built image we provided from the docker hub (only compatible with Ampere NVIDIA GPU architecture):

```
docker image pull expye/sparestir-ae:
    latest
docker tag expye/sparsetir-ae:latest
    sparsetir
```

B.5 Experiment Workflow

We provide a `run.sh` script under each folder, and user can run these scripts in docker container for corresponding benchmarks:

spmm contains scripts to reproduce SpMM experiments in §4.2.1.
sddmm contains scripts to reproduce SDDMM experiments in §4.2.2.
e2e contains scripts to reproduce GraphSAGE end-to-end training experiments in §4.2.3.
sparse-attention contains scripts to reproduce Sparse Transformer operator experiments in §4.3.1.
pruned-bert contains scripts to reproduce PrunedBERT experiments in §4.3.2 and §4.3.2.
rgcn contains scripts to reproduce RGCN inference end-to-end experiments in figure §4.4.1.
sparse-conv contains scripts to reproduce Sparse Convolution operator experiments in §4.4.2.

The scripts will produce logging files containing the profiling results including average execution time and GPU memory usage, and figures plotted in the same style as the paper. We also provide a `run-all.sh` script under the root directory for running all experiments in a single command, which would take around 10 hours to finish on a GPU like RTX 3080. We use `cudaEvent` APIs to profile CUDA kernels. During profiling, we discard the samples for the first 10 runs as warm-up steps and repeat for 100 cycles.

B.6 Evaluation and Expected Results

The specific running time and speedup differ on different platforms but we expect the results users reproduced should roughly match the numbers reported in the paper. (see Figures 13, 14, 15, 16, 17, 19, 20 and 23).

B.7 Experiment Customization

Artifact users can customize the benchmark scripts to use other datasets, for GNN operator or end-to-end training/inference benchmarks, users can create their own datasets as `DGLGraph` class (the graph data structure used in DGL). For the sparse convolution benchmark, users need to convert the customized point cloud dataset to `SparseTensor` class introduced in TorchSparse. For the network pruning benchmark, user can convert their own pruned weights to scipy sparse matrix.

B.8 Notes

Many previous work do not flush L2 cache when profiling CUDA kernels, which results in incorrect measurement especially for "small" operators, because the data accessed in the previous run would reside in L2 cache thus reducing the memory latency in the next run if they are accessed before being replaced. In this artifact we provide an option for the user to determine whether to enable L2 or not: if the environment variable `FLUSH_L2` is set to `ON`, we enable L2 flush for all benchmarks, and if `FLUSH_L2` is set to `OFF` we will disable L2 flush. All experiment results reported in this paper are obtained with `FLUSH_L2=ON`.

REFERENCES

[1] Andrew Adams, Karima Ma, Luke Anderson, Riyadh Baghdadi, Tzu-Mao Li, Michaël Gharbi, Benoit Steiner, Steven Johnson, Kayvon Fatahalian, Frédo Durand, and Jonathan Ragan-Kelley. 2019. Learning to optimize halide with tree search and random programs. *ACM Trans. Graph.* 38, 4 (2019), 121:1–121:12. https://doi.org/10.1145/3306346.3322967

[2] Nawaaz Ahmed, Nikolay Mateev, Keshav Pingali, and Paul Stodghill. 2000. A Framework for Sparse Matrix Code Synthesis from High-level Specifications. In *Proceedings Supercomputing 2000, November 4-10, 2000, Dallas, Texas, USA. IEEE Computer Society, CD-ROM*, Jed Donnelley (Ed.). IEEE Computer Society, 58. https://doi.org/10.1109/SC.2000.10033

[3] Peter Ahrens, Fredrik Kjolstad, and Saman Amarasinghe. 2022. Autoscheduling for Sparse Tensor Algebra with an Asymptotic Cost Model. In *Proceedings of the 43rd ACM SIGPLAN International Conference on Programming Language Design and Implementation* (San Diego, CA, USA) *(PLDI 2022)*. Association for Computing Machinery, New York, NY, USA, 269–285. https://doi.org/10.1145/3519939.3523442

[4] Riyadh Baghdadi, Abdelkader Nadir Debbagh, Kamel Abdous, Fatima-Zohra Benhamida, Alex Renda, Jonathan Elliott Frankle, Michael Carbin, and Saman P. Amarasinghe. 2020. TIRAMISU: A Polyhedral Compiler for Dense and Sparse Deep Learning. *CoRR* abs/2005.04091 (2020). arXiv:2005.04091 https://arxiv.org/abs/2005.04091

[5] J. Behley, M. Garbade, A. Milioto, J. Quenzel, S. Behnke, C. Stachniss, and J. Gall. 2019. SemanticKITTI: A Dataset for Semantic Scene Understanding of LiDAR Sequences. In *Proc. of the IEEE/CVF International Conf. on Computer Vision (ICCV)*.

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

[7] Aart Bik, Penporn Koanantakool, Tatiana Shpeisman, Nicolas Vasilache, Bixia Zheng, and Fredrik Kjolstad. 2022. Compiler Support for Sparse Tensor Computations in MLIR. *ACM Trans. Archit. Code Optim.* 19, 4, Article 50 (sep 2022), 25 pages. https://doi.org/10.1145/3544559

[8] A.J.C. Bik and H.A.G. Wijshoff. 1995. Advanced Compiler Optimizations for Sparse Computations. *J. Parallel and Distrib. Comput.* 31, 1 (1995), 14–24. https://doi.org/10.1006/jpdc.1995.1141

[9] Aart J. C. Bik. 1996. *Compiler Support for Sparse Matrix Computations*. Ph. D. Dissertation.

[10] Aart J. C. Bik and Harry A. G. Wijshoff. 1993. Compilation Techniques for Sparse Matrix Computations. In *Proceedings of the 7th international conference on Supercomputing, ICS 1993, Tokyo, Japan, July 20-22, 1993*, Yoichi Muraoka (Ed.). ACM, 416–424. https://doi.org/10.1145/165939.166023

[11] Aart J. C. Bik and Harry A. G. Wijshoff. 1994. Nonzero Structure Analysis. In *Proceedings of the 8th International Conference on Supercomputing* (Manchester, England) *(ICS '94)*. Association for Computing Machinery, New York, NY, USA, 226–235. https://doi.org/10.1145/181181.181538

[12] Aart J. C. Bik and Harry A. G. Wijshoff. 1996. Automatic Data Structure Selection and Transformation for Sparse Matrix Computations. *IEEE Trans. Parallel Distributed Syst.* 7, 2 (1996), 109–126. https://doi.org/10.1109/71.485501

[13] Aydin Buluç and John R. Gilbert. 2008. On the representation and multiplication of hypersparse matrices. In *22nd IEEE International Symposium on Parallel and Distributed Processing, IPDPS 2008, Miami, Florida USA, April 14-18, 2008*. IEEE, 1–11. https://doi.org/10.1109/IPDPS.2008.4536313

[14] 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* (Seaside, CA, USA) *(FPGA '19)*. Association for Computing Machinery, New York, NY, USA, 63–72. https://doi.org/10.1145/3289602.3293898

[15] Beidi Chen, Tri Dao, Kaizhao Liang, Jiaming Yang, Zhao Song, Atri Rudra, and Christopher Ré. 2022. Pixelated Butterfly: Simple and Efficient Sparse training for Neural Network Models. In *The Tenth International Conference on Learning Representations, ICLR 2022, Virtual Event, April 25-29, 2022*. OpenReview.net. https://openreview.net/forum?id=Nfl-iXa-y7R

[16] Tianqi Chen, Thierry Moreau, Ziheng Jiang, Lianmin Zheng, Eddie Yan, Haichen Shen, Meghan Cowan, Leyuan Wang, Yuwei Hu, Luis Ceze, Carlos Guestrin, and Arvind Krishnamurthy. 2018. TVM: An Automated End-to-End Optimizing Compiler for Deep Learning. In *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)*. USENIX Association, Carlsbad, CA, 578–594. https://www.usenix.org/conference/osdi18/presentation/chen

[17] Tianqi Chen, Lianmin Zheng, Eddie Yan, Ziheng Jiang, Thierry Moreau, Luis Ceze, Carlos Guestrin, and Arvind Krishnamurthy. 2018. Learning to Optimize Tensor Programs. In *Proceedings of the 32nd International Conference on Neural Information Processing Systems* (Montréal, Canada) *(NIPS'18)*. Curran Associates Inc., Red Hook, NY, USA, 3393–3404.

[18] Kazem Cheshmi, Shoaib Kamil, Michelle Mills Strout, and Maryam Mehri Dehnavi. 2017. Sympiler: transforming sparse matrix codes by decoupling symbolic analysis. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis, SC 2017, Denver, CO, USA, November 12 - 17, 2017*, Bernd Mohr and Padma Raghavan (Eds.). ACM, 13. https://doi.org/10.1145/3126908.3126936

[19] Kazem Cheshmi, Shoaib Kamil, Michelle Mills Strout, and Maryam Mehri Dehnavi. 2018. ParSy: inspection and transformation of sparse matrix computations for parallelism. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage, and Analysis, SC 2018, Dallas, TX, USA, November 11-16, 2018*. IEEE / ACM, 62:1–62:15. http://dl.acm.org/citation.cfm?id=3291739

[20] Rewon Child, Scott Gray, Alec Radford, and Ilya Sutskever. 2019. Generating Long Sequences with Sparse Transformers. *CoRR* abs/1904.10509 (2019). arXiv:1904.10509 http://arxiv.org/abs/1904.10509

[21] Stephen Chou, Fredrik Kjolstad, and Saman Amarasinghe. 2018. Format Abstraction for Sparse Tensor Algebra Compilers. *Proc. ACM Program. Lang.* 2, OOPSLA, Article 123 (oct 2018), 30 pages. https://doi.org/10.1145/3276493

[22] 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* (London, UK) *(PLDI 2020)*. Association for Computing Machinery, New York, NY, USA, 823–838. https://doi.org/10.1145/3385412.3385963

[23] Christopher Choy, JunYoung Gwak, and Silvio Savarese. 2019. 4D Spatio-Temporal ConvNets: Minkowski Convolutional Neural Networks. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*. 3075–3084.

[24] NVIDIA Corporation. 2022. cuSPARSE :: CUDA Toolkit Documentation v11.7.1. https://docs.nvidia.com/cuda/cusparse/index.html.

[25] Guohao Dai, Guyue Huang, Shang Yang, Zhongming Yu, Hengrui Zhang, Yufei Ding, Yuan Xie, Huazhong Yang, and Yu Wang. 2022. Heuristic Adaptability to Input Dynamics for SpMM on GPUs. In *Proceedings of the 59th ACM/IEEE Design Automation Conference* (San Francisco, California) *(DAC '22)*. Association for Computing Machinery, New York, NY, USA, 595–600. https://doi.org/10.1145/3489517.3530508

[26] Tensorflow Developers. 2018. Ragged tensors | TensorFlow Core. https://www.tensorflow.org/guide/ragged_tensor.

[27] Tensorflow Developers. 2018. XLA: Optimizing Compiler for Machine Learning | TensorFlow. https://www.tensorflow.org/xla.

[28] dgSPARSE team. 2021. dgSPARSE Library. https://github.com/dgSPARSE/dgSPARSE-Library.

[29] Adhitha Dias, Kirshanthan Sundararajah, Charitha Saumya, and Milind Kulkarni. 2022. SparseLNR: Accelerating Sparse Tensor Computations Using Loop Nest Restructuring. In *Proceedings of the 36th ACM International Conference on Supercomputing* (Virtual Event) *(ICS '22)*. Association for Computing Machinery, New York, NY, USA, Article 15, 14 pages. https://doi.org/10.1145/3524059.3532386

[30] Iain S. Duff. 1987. *The Use of Vector and Parallel Computers in the Solution of Large Sparse Linear Equations*. Birkhäuser Boston, Boston, MA, 331–348. https://doi.org/10.1007/978-1-4684-6754-3_20

[31] Iain S Duff, Albert M Erisman, and John K Reid. 1986. *Direct Methods for Sparse Matrices*. Oxford University Press, Inc., USA.

[32] William Fedus, Barret Zoph, and Noam Shazeer. 2022. Switch Transformers: Scaling to Trillion Parameter Models with Simple and Efficient Sparsity. *Journal of Machine Learning Research* 23, 120 (2022), 1–39. http://jmlr.org/papers/v23/21-0998.html

[33] Pratik Fegade, Tianqi Chen, Phillip B. Gibbons, and Todd C. Mowry. 2022. The CoRa Tensor Compiler: Compilation for Ragged Tensors with Minimal Padding. In *Proceedings of Machine Learning and Systems*, A. Smola, A. Dimakis, and I. Stoica (Eds.).

[34] Siyuan Feng, Bohan Hou, Hongyi Jin, Wuwei Lin, Junru Shao, Ruihang Lai, Zihao Ye, Lianmin Zheng, Cody Hao Yu, Yong Yu, and Tianqi Chen. 2023. TensorIR: An Abstraction for Automatic Tensorized Program Optimization. In *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2* (Vancouver, BC, Canada) *(ASPLOS 2023)*. Association for Computing Machinery, New York, NY, USA, 804–817. https://doi.org/10.1145/3575693.3576933

[35] Yifan Feng, Haoxuan You, Zizhao Zhang, Rongrong Ji, and Yue Gao. 2019. Hypergraph Neural Networks. In *The Thirty-Third AAAI Conference on Artificial Intelligence, AAAI 2019, The Thirty-First Innovative Applications of Artificial Intelligence Conference, IAAI 2019, The Ninth AAAI Symposium on Educational Advances in Artificial Intelligence, EAAI 2019, Honolulu, Hawaii, USA, January 27 - February 1, 2019*. AAAI Press, 3558–3565. https://doi.org/10.1609/aaai.v33i01.33013558

[36] Matthias Fey and Jan E. Lenssen. 2019. Fast Graph Representation Learning with PyTorch Geometric. In *ICLR Workshop on Representation Learning on Graphs and Manifolds*.

[37] 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* (Atlanta, Georgia) *(SC '20)*. IEEE Press, Article 17, 14 pages.

[38] Justin Gilmer, Samuel S. Schoenholz, Patrick F. Riley, Oriol Vinyals, and George E. Dahl. 2017. Neural Message Passing for Quantum Chemistry. In *Proceedings of the 34th International Conference on Machine Learning - Volume 70* (Sydney, NSW, Australia) *(ICML'17)*. JMLR.org, 1263–1272.

[39] William L. Hamilton, Rex Ying, and Jure Leskovec. 2017. Inductive Representation Learning on Large Graphs. In *Proceedings of the 31st International Conference on Neural Information Processing Systems* (Long Beach, California, USA) *(NIPS'17)*. Curran Associates Inc., Red Hook, NY, USA, 1025–1035.

[40] Song Han, Huizi Mao, and William J. Dally. 2016. Deep Compression: Compressing Deep Neural Network with Pruning, Trained Quantization and Huffman Coding. In *4th International Conference on Learning Representations, ICLR 2016, San Juan, Puerto Rico, May 2-4, 2016, Conference Track Proceedings*, Yoshua Bengio and Yann LeCun (Eds.). http://arxiv.org/abs/1510.00149

[41] Rawn Henry, Olivia Hsu, Rohan Yadav, Stephen Chou, Kunle Olukotun, Saman Amarasinghe, and Fredrik Kjolstad. 2021. Compilation of Sparse Array Programming Models. *Proc. ACM Program. Lang.* 5, OOPSLA, Article 128 (oct 2021), 29 pages. https://doi.org/10.1145/3485505

[42] Torsten Hoefler, Dan Alistarh, Tal Ben-Nun, Nikoli Dryden, and Alexandra Peste. 2021. Sparsity in Deep Learning: Pruning and growth for efficient inference and training in neural networks. *J. Mach. Learn. Res.* 22 (2021), 241:1–241:124. http://jmlr.org/papers/v22/21-0366.html

[43] 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* (Washington, District of Columbia) *(PPoPP '19)*. Association for Computing Machinery, New York, NY, USA, 300–314. https://doi.org/10.1145/3293883.3295712

[44] E. N. Houstis, J. R. Rice, N. P. Chrisochoides, H. C. Karathanasis, P. N. Papachiou, M. K. Samartzis, E. A. Vavalis, Ko Yang Wang, and S. Weerawarana. 1990. //ELLPACK: A Numerical Simulation Programming Environment for Parallel MIMD Machines. *SIGARCH Comput. Archit. News* 18, 3b (jun 1990), 96–107. https://doi.org/10.1145/255129.255144

[45] Weihua Hu, Matthias Fey, Marinka Zitnik, Yuxiao Dong, Hongyu Ren, Bowen Liu, Michele Catasta, and Jure Leskovec. 2020. Open Graph Benchmark: Datasets for Machine Learning on Graphs. In *Advances in Neural Information Processing Systems 33: Annual Conference on Neural Information Processing Systems 2020, NeurIPS 2020, December 6-12, 2020, virtual*, Hugo Larochelle, Marc'Aurelio Ranzato, Raia Hadsell, Maria-Florina Balcan, and Hsuan-Tien Lin (Eds.). https://proceedings.neurips.cc/paper/2020/hash/fb60d411a5c5b72b2e7d3527cfc84fd0-Abstract.html

[46] Yuanming Hu, Tzu-Mao Li, Luke Anderson, Jonathan Ragan-Kelley, and Frédo Durand. 2019. Taichi: A Language for High-Performance Computation on Spatially Sparse Data Structures. *ACM Trans. Graph.* 38, 6, Article 201 (nov 2019), 16 pages. https://doi.org/10.1145/3355089.3356506

[47] Yuwei Hu, Zihao Ye, Minjie Wang, Jiali Yu, Da Zheng, Mu Li, Zheng Zhang, Zhiru Zhang, and Yida Wang. 2020. FeatGraph: A Flexible and Efficient Backend for Graph Neural Network Systems. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis* (Atlanta, Georgia) *(SC '20)*. IEEE Press, Article 71, 13 pages.

[48] Ziniu Hu, Yuxiao Dong, Kuansan Wang, and Yizhou Sun. 2020. Heterogeneous Graph Transformer. In *WWW '20: The Web Conference 2020, Taipei, Taiwan, April 20-24, 2020*, Yennun Huang, Irwin King, Tie-Yan Liu, and Maarten van Steen (Eds.). ACM / IW3C2, 2704–2710. https://doi.org/10.1145/3366423.3380027

[49] Guyue Huang, Guohao Dai, Yu Wang, and Huazhong Yang. 2020. GE-SpMM: General-Purpose Sparse Matrix-Matrix Multiplication on GPUs for Graph Neural Networks. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis* (Atlanta, Georgia) *(SC '20)*. IEEE Press, Article 72, 12 pages.

[50] Zhihao Jia, Oded Padon, James Thomas, Todd Warszawski, Matei Zaharia, and Alex Aiken. 2019. TASO: Optimizing Deep Learning Computation with Automatic Generation of Graph Substitutions. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles* (Huntsville, Ontario, Canada) *(SOSP '19)*. Association for Computing Machinery, New York, NY, USA, 47–62. https://doi.org/10.1145/3341301.3359630

[51] Norman P. Jouppi, Cliff Young, Nishant Patil, David A. Patterson, Gaurav Agrawal, Raminder Bajwa, Sarah Bates, Suresh Bhatia, Nan Boden, Al Borchers, Rick Boyle, Pierre-luc Cantin, Clifford Chao, Chris Clark, Jeremy Coriell, Mike Daley, Matt Dau, Jeffrey Dean, Ben Gelb, Tara Vazir Ghaemmaghami, Rajendra Gottipati, William Gulland, Robert Hagmann, C. Richard Ho, Doug Hogberg, John Hu, Robert Hundt, Dan Hurt, Julian Ibarz, Aaron Jaffey, Alek Jaworski, Alexander Kaplan, Harshit Khaitan, Daniel Killebrew, Andy Koch, Naveen Kumar, Steve Lacy, James Laudon, James Law, Diemthu Le, Chris Leary, Zhuyuan Liu, Kyle Lucke, Alan Lundin, Gordon MacKean, Adriana Maggiore, Maire Mahony, Kieran Miller, Rahul Nagarajan, Ravi Narayanaswami, Ray Ni, Kathy Nix, Thomas Norrie, Mark Omernick, Narayana Penukonda, Andy Phelps, Jonathan Ross, Matt Ross, Amir Salek, Emad Samadiani, Chris Severn, Gregory Sizikov, Matthew Snelham, Jed Souter, Dan Steinberg, Andy Swing, Mercedes Tan, Gregory Thorson, Bo Tian, Horia Toma, Erick Tuttle, Vijay Vasudevan, Richard Walter, Walter Wang, Eric Wilcox, and Doe Hyun Yoon. 2017. In-Datacenter Performance Analysis of a Tensor Processing Unit. In *Proceedings of the 44th Annual International Symposium on Computer Architecture, ISCA 2017, Toronto, ON, Canada, June 24-28, 2017*. ACM, 1–12. https://doi.org/10.1145/3079856.3080246

[52] Thomas N. Kipf and Max Welling. 2017. Semi-Supervised Classification with Graph Convolutional Networks. In *5th International Conference on Learning Representations, ICLR 2017, Toulon, France, April 24-26, 2017, Conference Track Proceedings*. OpenReview.net. https://openreview.net/forum?id=SJU4ayYgl

[53] Fredrik Kjolstad, Peter Ahrens, Shoaib Kamil, and Saman Amarasinghe. 2019. Tensor Algebra Compilation with Workspaces. (2019), 180–192. http://dl.acm.org/citation.cfm?id=3314872.3314894

[54] Fredrik Kjolstad, Shoaib Kamil, Stephen Chou, David Lugato, and Saman Amarasinghe. 2017. The Tensor Algebra Compiler. *Proc. ACM Program. Lang.* 1, OOPSLA, Article 77 (oct 2017), 29 pages. https://doi.org/10.1145/3133901

[55] François Lagunas, Ella Charlaix, Victor Sanh, and Alexander Rush. 2021. Block Pruning For Faster Transformers. In *Proceedings of the 2021 Conference on Empirical Methods in Natural Language Processing*. Association for Computational Linguistics, Online and Punta Cana, Dominican Republic, 10619–10629. https://doi.org/10.18653/v1/2021.emnlp-main.829

[56] Ao Li, Bojian Zheng, Gennady Pekhimenko, and Fan Long. 2022. Automatic Horizontal Fusion for GPU Kernels. In *Proceedings of the 20th IEEE/ACM International Symposium on Code Generation and Optimization* (Virtual Event, Republic of Korea) *(CGO '22)*. IEEE Press, 14–27. https://doi.org/10.1109/CGO53902.2022.9741270

[57] Jiajia Li, Bora Uçar, Ümit V. Çatalyürek, Jimeng Sun, Kevin Barker, and Richard Vuduc. 2019. Efficient and Effective Sparse Tensor Reordering. In *Proceedings of the ACM International Conference on Supercomputing* (Phoenix, Arizona) *(ICS '19)*. Association for Computing Machinery, New York, NY, USA, 227–237. https://doi.org/10.1145/3330345.3330366

[58] Shigang Li, Kazuki Osawa, and Torsten Hoefler. 2022. Efficient Quantized Sparse Matrix Operations on Tensor Cores. In *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis* (Dallas, Texas) *(SC '22)*. IEEE Press, Article 37, 15 pages.

[59] Chien-Yu Lin, Liang Luo, and Luis Ceze. 2021. Accelerating SpMM Kernel with Cache-First Edge Sampling for Graph Neural Networks. *CoRR* abs/2104.10716 (2021). arXiv:2104.10716 https://arxiv.org/abs/2104.10716

[60] Nikolay Mateev, Keshav Pingali, Paul Stodghill, and Vladimir Kotlyar. 2000. Next-generation generic programming and its application to sparse matrix computations. In *Proceedings of the 14th international conference on Supercomputing, ICS 2000, Santa Fe, NM, USA, May 8-11, 2000*, John Reynders and Alexander V. Veidenbaum (Eds.). ACM, 88–99. https://doi.org/10.1145/335231.335240

[61] Atefeh Mehrabi, Donghyuk Lee, Niladrish Chatterjee, Daniel J. Sorin, Benjamin C. Lee, and Mike O'Connor. 2021. Learning Sparse Matrix Row Permutations for Efficient SpMM on GPU Architectures. In *IEEE International Symposium on Performance Analysis of Systems and Software, ISPASS 2021, Stony Brook, NY, USA, March 28-30, 2021*. IEEE, 48–58. https://doi.org/10.1109/ISPASS51385.2021.00016

[62] Mahdi Soltan Mohammadi, Tomofumi Yuki, Kazem Cheshmi, Eddie C. Davis, Mary W. Hall, Maryam Mehri Dehnavi, Payal Nandy, Catherine Olschanowsky, Anand Venkat, and Michelle Mills Strout. 2019. Sparse computation data dependence simplification for efficient compiler-generated inspectors. In *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2019, Phoenix, AZ, USA, June 22-26, 2019*, Kathryn S. McKinley and Kathleen Fisher (Eds.). ACM, 594–609. https://doi.org/10.1145/3314221.3314646

[63] Israt Nisa, Aravind Sukumaran-Rajam, Sureyya Emre Kurt, Changwan Hong, and P. Sadayappan. 2018. Sampled Dense Matrix Multiplication for High-Performance Machine Learning. In *2018 IEEE 25th International Conference on High Performance Computing (HiPC)*. 32–41. https://doi.org/10.1109/HiPC.2018.00013

[64] Thomas C. Oppe and David R. Kincaid. 1987. The performance of ITPACK on vector computers for solving large sparse linear systems arising in sample oil reseervoir simulation problems. *Communications in Applied Numerical Methods* 3, 1 (1987), 23–29. https://doi.org/10.1002/cnm.1630030106 arXiv:https://onlinelibrary.wiley.com/doi/pdf/10.1002/cnm.1630030106

[65] Douglass Stott Parker. 1995. *Random butterfly transformations with applications in computational linear algebra.* UCLA Computer Science Department.

[66] Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, Alban Desmaison, Andreas Köpf, Edward Z. 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: Annual Conference on Neural Information Processing Systems 2019, NeurIPS 2019, December 8-14, 2019, Vancouver, BC, Canada*, Hanna M. Wallach, Hugo Larochelle, Alina Beygelzimer, Florence d'Alché-Buc, Emily B. Fox, and Roman Garnett (Eds.). 8024–8035. https://proceedings.neurips.cc/paper/2019/hash/bdbca288fee7f92f2bfa9f7012727740-Abstract.html

[67] Gregoire Pichon, Mathieu Faverge, Pierre Ramet, and Jean Roman. 2017. Reordering Strategy for Blocking Optimization in Sparse Linear Solvers. *SIAM J. Matrix Anal. Appl.* 38, 1 (2017), 226–248. https://doi.org/10.1137/16M1062454 arXiv:https://doi.org/10.1137/16M1062454

[68] Jeff Pool. 2020. Accelerating Sparsity in the NVIDIA Ampere Architecture. https://developer.download.nvidia.com/video/gputechconf/gtc/2020/presentations/s22085-accelerating-sparsity-in-the-nvidia-ampere-architecture%E2%80%8B.pdf.

[69] William W. Pugh and Tatiana Shpeisman. 1998. SIPR: A New Framework for Generating Efficient Code for Sparse Matrix Computations. In *Languages and Compilers for Parallel Computing, 11th International Workshop, LCPC'98, Chapel Hill, NC, USA, August 7-9, 1998, Proceedings (Lecture Notes in Computer Science, Vol. 1656)*, Siddhartha Chatterjee, Jan F. Prins, Larry Carter, Jeanne Ferrante, Zhiyuan Li, David C. Sehr, and Pen-Chung Yew (Eds.). Springer, 213–229. https://doi.org/10.1007/3-540-48319-5_14

[70] Jonathan Ragan-Kelley, Connelly Barnes, Andrew Adams, Sylvain Paris, Frédo Durand, and Saman Amarasinghe. 2013. Halide: A Language and Compiler for Optimizing Parallelism, Locality, and Recomputation in Image Processing Pipelines. In *Proceedings of the 34th ACM SIGPLAN Conference on Programming Language Design and Implementation* (Seattle, Washington, USA) *(PLDI '13)*. Association for Computing Machinery, New York, NY, USA, 519–530. https://doi.org/10.1145/2491956.2462176

[71] Md. Khaledur Rahman, Majedul Haque Sujon, and Ariful Azad. 2021. FusedMM: A Unified SDDMM-SpMM Kernel for Graph Embedding and Graph Neural Networks. In *2021 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*. 256–266. https://doi.org/10.1109/IPDPS49936.2021.00034

[72] Petar Ristoski, Gerben Klaas Dirk de Vries, and Heiko Paulheim. 2016. A Collection of Benchmark Datasets for Systematic Evaluations of Machine Learning on the Semantic Web. In *The Semantic Web – ISWC 2016*, Paul Groth, Elena Simperl, Alasdair Gray, Marta Sabou, Markus Krötzsch, Freddy Lecue, Fabian Flöck, and Yolanda Gil (Eds.). Springer International Publishing, Cham, 186–194.

[73] Jared Roesch, Steven Lyubomirsky, Logan Weber, Josh Pollock, Marisa Kirisame, Tianqi Chen, and Zachary Tatlock. 2018. Relay: A New IR for Machine Learning Frameworks. In *Proceedings of the 2nd ACM SIGPLAN International Workshop on Machine Learning and Programming Languages* (Philadelphia, PA, USA) *(MAPL 2018)*. Association for Computing Machinery, New York, NY, USA, 58–68. https://doi.org/10.1145/3211346.3211348

[74] Youcef Saad. 1989. Krylov Subspace Methods on Supercomputers. *SIAM J. Sci. Statist. Comput.* 10, 6 (1989), 1200–1232. https://doi.org/10.1137/0910073 arXiv:https://doi.org/10.1137/0910073

[75] Youcef Saad. 1990. *SPARSKIT: A basic tool kit for sparse matrix computations*. Technical Report.

[76] Victor Sanh, Thomas Wolf, and Alexander Rush. 2020. Movement Pruning: Adaptive Sparsity by Fine-Tuning. In *Advances in Neural Information Processing Systems*, H. Larochelle, M. Ranzato, R. Hadsell, M.F. Balcan, and H. Lin (Eds.), Vol. 33. Curran Associates, Inc., 20378–20389. https://proceedings.neurips.cc/paper/2020/file/eae15aabaa768ae4a5993a8a4f4fa6e4-Paper.pdf

[77] Michael Sejr Schlichtkrull, Thomas N. Kipf, Peter Bloem, Rianne van den Berg, Ivan Titov, and Max Welling. 2018. Modeling Relational Data with Graph Convolutional Networks. In *The Semantic Web - 15th International Conference, ESWC 2018, Heraklion, Crete, Greece, June 3-7, 2018, Proceedings (Lecture Notes in Computer Science, Vol. 10843)*, Aldo Gangemi, Roberto Navigli, Maria-Esther Vidal, Pascal Hitzler, Raphaël Troncy, Laura Hollink, Anna Tordai, and Mehwish Alam (Eds.). Springer, 593–607. https://doi.org/10.1007/978-3-319-93417-4_38

[78] Prithviraj Sen, Galileo Namata, Mustafa Bilgic, Lise Getoor, Brian Gallagher, and Tina Eliassi-Rad. 2008. Collective Classification in Network Data. *AI Mag.* 29, 3 (2008), 93–106. https://doi.org/10.1609/aimag.v29i3.2157

[79] 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. *Proc. ACM Program. Lang.* 4, OOPSLA, Article 158 (Nov. 2020), 30 pages. https://doi.org/10.1145/3428226

[80] Junru Shao, Xiyou Zhou, Siyuan Feng, Bohan Hou, Ruihang Lai, Hongyi Jin, Wuwei Lin, Masahiro Masuda, Cody Hao Yu, and Tianqi Chen. 2022. Tensor Program Optimization with Probabilistic Programs. https://doi.org/10.48550/ARXIV.2205.13603

[81] Noam Shazeer, Azalia Mirhoseini, Krzysztof Maziarz, Andy Davis, Quoc V. Le, Geoffrey E. Hinton, and Jeff Dean. 2017. Outrageously Large Neural Networks: The Sparsely-Gated Mixture-of-Experts Layer. In *5th International Conference on Learning Representations, ICLR 2017, Toulon, France, April 24-26, 2017, Conference Track Proceedings*. OpenReview.net. https://openreview.net/forum?id=B1ckMDqlg

[82] Shaden Smith and George Karypis. 2015. Tensor-Matrix Products with a Compressed Sparse Tensor. In *Proceedings of the 5th Workshop on Irregular Applications: Architectures and Algorithms* (Austin, Texas) *(IA³ '15)*. Association for Computing Machinery, New York, NY, USA, Article 5, 7 pages. https://doi.org/10.1145/2833179.2833183

[83] Michelle Mills Strout, Mary W. Hall, and Catherine Olschanowsky. 2018. The Sparse Polyhedral Framework: Composing Compiler-Generated Inspector-Executor Code. *Proc. IEEE* 106, 11 (2018), 1921–1934. https://doi.org/10.1109/JPROC.2018.2857721

[84] Patricia Suriana, Andrew Adams, and Shoaib Kamil. 2017. Parallel Associative Reductions in Halide. In *Proceedings of the 2017 International Symposium on Code Generation and Optimization* (Austin, USA) *(CGO '17)*. IEEE Press, 281–291.

[85] Vivienne Sze, Yu-Hsin Chen, Tien-Ju Yang, and Joel S. Emer. 2020. *Efficient Processing of Deep Neural Networks*. Morgan & Claypool Publishers. https://doi.org/10.2200/S01004ED1V01Y202004CAC050

[86] Haotian Tang, Zhijian Liu, Xiuyu Li, Yujun Lin, and Song Han. 2022. TorchSparse: Efficient Point Cloud Inference Engine. In *Proceedings of Machine Learning and Systems*, D. Marculescu, Y. Chi, and C. Wu (Eds.), Vol. 4. 302–315. https://proceedings.mlsys.org/paper/2022/file/6512bd43d9caa6e02c990b0a82652dca-Paper.pdf

[87] Shizhi Tang, Jidong Zhai, Haojie Wang, Lin Jiang, Liyan Zheng, Zhenhao Yuan, and Chen Zhang. 2022. FreeTensor: A Free-Form DSL with Holistic Optimizations for Irregular Tensor Programs. In *Proceedings of the 43rd ACM SIGPLAN International Conference on Programming Language Design and Implementation* (San Diego, CA, USA) *(PLDI 2022)*. Association for Computing Machinery, New York, NY, USA, 872–887. https://doi.org/10.1145/3519939.3523448

[88] Ruiqin Tian, Luanzheng Guo, Jiajia Li, Bin Ren, and Gokcen Kestor. 2021. A High Performance Sparse Tensor Algebra Compiler in MLIR. In *2021 IEEE/ACM 7th Workshop on the LLVM Compiler Infrastructure in HPC (LLVM-HPC)*. 27–38. https://doi.org/10.1109/LLVMHPC54804.2021.00009

[89] Philippe Tillet, H. T. Kung, and David Cox. 2019. *Triton: An Intermediate Language and Compiler for Tiled Neural Network Computations*. Association for Computing Machinery, New York, NY, USA, 10–19. https://doi.org/10.1145/3315508.3329973

[90] Nicolas Vasilache, Oleksandr Zinenko, Aart J. C. Bik, Mahesh Ravishankar, Thomas Raoux, Alexander Belyaev, Matthias Springer, Tobias Gysi, Diego Caballero, Stephan Herhut, Stella Laurenzo, and Albert Cohen. 2022. Composable and Modular Code Generation in MLIR: A Structured and Retargetable Approach to Tensor Compiler Construction. *CoRR* abs/2202.03293 (2022). arXiv:2202.03293 https://arxiv.org/abs/2202.03293

[91] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N. Gomez, Łukasz Kaiser, and Illia Polosukhin. 2017. Attention is All You Need. In *Proceedings of the 31st International Conference on Neural Information Processing Systems* (Long Beach, California, USA) *(NIPS'17)*. Curran Associates Inc., Red Hook, NY, USA, 6000–6010.

[92] Petar Veličković, Guillem Cucurull, Arantxa Casanova, Adriana Romero, Pietro Liò, and Yoshua Bengio. 2018. Graph Attention Networks. In *International Conference on Learning Representations*. https://openreview.net/forum?id=rJXMpikCZ

[93] Pauli Virtanen, Ralf Gommers, Travis E. Oliphant, Matt Haberland, Tyler Reddy, David Cournapeau, Evgeni Burovski, Pearu Peterson, Warren Weckesser, Jonathan Bright, Stéfan J. van der Walt, Matthew Brett, Joshua Wilson, K. Jarrod Millman, Nikolay Mayorov, Andrew R. J. Nelson, Eric Jones, Robert Kern, Eric Larson, C J Carey, İlhan Polat, Yu Feng, Eric W. Moore, Jake VanderPlas, Denis Laxalde, Josef Perktold, Robert Cimrman, Ian Henriksen, E. A. Quintero, Charles R. Harris, Anne M. Archibald, Antônio H. Ribeiro, Fabian Pedregosa, Paul van Mulbregt, and SciPy 1.0 Contributors. 2020. SciPy 1.0: Fundamental Algorithms for Scientific Computing in Python. *Nature Methods* 17 (2020), 261–272. https://doi.org/10.1038/s41592-019-0686-2

[94] Richard Vuduc, James W Demmel, and Katherine A Yelick. 2005. OSKI: A library of automatically tuned sparse matrix kernels. *Journal of Physics: Conference Series* 16 (jan 2005), 521–530. https://doi.org/10.1088/1742-6596/16/1/071

[95] Endong Wang, Qing Zhang, Bo Shen, Guangyong Zhang, Xiaowei Lu, Qing Wu, and Yajuan Wang. 2014. Intel math kernel library. In *High-Performance Computing on the Intel® Xeon Phi™*. Springer, 167–188.

[96] Minjie Wang, Da Zheng, Zihao Ye, Quan Gan, Mufei Li, Xiang Song, Jinjing Zhou, Chao Ma, Lingfan Yu, Yu Gai, Tianjun Xiao, Tong He, George Karypis, Jinyang Li, and Zheng Zhang. 2019. Deep Graph Library: A Graph-Centric, Highly-Performant Package for Graph Neural Networks. *arXiv preprint arXiv:1909.01315* (2019).

[97] Xiao Wang, Houye Ji, Chuan Shi, Bai Wang, Yanfang Ye, Peng Cui, and Philip S. Yu. 2019. Heterogeneous Graph Attention Network. In *The World Wide Web Conference, WWW 2019, San Francisco, CA, USA, May 13-17, 2019*, Ling Liu, Ryen W. White, Amin Mantrach, Fabrizio Silvestri, Julian J. McAuley, Ricardo Baeza-Yates, and Leila Zia (Eds.). ACM, 2022–2032. https://doi.org/10.1145/3308558.3313562

[98] Yuke Wang, Boyuan Feng, and Yufei Ding. 2021. TC-GNN: Accelerating Sparse Graph Neural Network Computation Via Dense Tensor Core on GPUs. *CoRR* abs/2112.02052 (2021). arXiv:2112.02052 https://arxiv.org/abs/2112.02052

[99] Yuke Wang, Boyuan Feng, and Yufei Ding. 2022. QGTC: Accelerating Quantized Graph Neural Networks via GPU Tensor Core. In *Proceedings of the 27th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming* (Seoul,

Republic of Korea) *(PPoPP '22)*. Association for Computing Machinery, New York, NY, USA, 107–119. https://doi.org/10.1145/3503221.3508408

[100] Yuke Wang, Boyuan Feng, Gushu Li, Shuangchen Li, Lei Deng, Yuan Xie, and Yufei Ding. 2021. GNNAdvisor: An Adaptive and Efficient Runtime System for GNN Acceleration on GPUs. In *15th USENIX Symposium on Operating Systems Design and Implementation (OSDI 21)*. USENIX Association, 515–531. https://www.usenix.org/conference/osdi21/presentation/wang-yuke

[101] Thomas Wolf, Lysandre Debut, Victor Sanh, Julien Chaumond, Clement Delangue, Anthony Moi, Pierric Cistac, Tim Rault, Rémi Louf, Morgan Funtowicz, and Jamie Brew. 2019. HuggingFace's Transformers: State-of-the-art Natural Language Processing. *CoRR* abs/1910.03771 (2019). arXiv:1910.03771 http://arxiv.org/abs/1910.03771

[102] Yidi Wu, Kaihao Ma, Zhenkun Cai, Tatiana Jin, Boyang Li, Chengguang Zheng, James Cheng, and Fan Yu. 2021. Seastar: vertex-centric programming for graph neural networks. In *EuroSys '21: Sixteenth European Conference on Computer Systems, Online Event, United Kingdom, April 26-28, 2021*, Antonio Barbalace, Pramod Bhatotia, Lorenzo Alvisi, and Cristian Cadar (Eds.). ACM, 359–375. https://doi.org/10.1145/3447786.3456247

[103] Zhiqiang Xie, Minjie Wang, Zihao Ye, Zheng Zhang, and Rui Fan. 2022. Graphiler: Optimizing Graph Neural Networks with Message Passing Data Flow Graph. In *Proceedings of Machine Learning and Systems*, D. Marculescu, Y. Chi, and C. Wu (Eds.), Vol. 4. 515–528. https://proceedings.mlsys.org/paper/2022/file/a87ff679a2f3e71d9181a67b7542122c-Paper.pdf

[104] Carl Yang, Aydin Buluç, and John D. Owens. 2018. Design Principles for Sparse Matrix Multiplication on the GPU. In *Euro-Par 2018: Parallel Processing - 24th International Conference on Parallel and Distributed Computing, Turin, Italy, August 27-31, 2018, Proceedings (Lecture Notes in Computer Science, Vol. 11014)*, Marco Aldinucci, Luca Padovani, and Massimo Torquati (Eds.). Springer, 672–687. https://doi.org/10.1007/978-3-319-96983-1_48

[105] Zihao Ye and Ruihang Lai. 2023. uwsampl/sparsetir-artifact: v1.3. (Feb 2023). https://doi.org/10.5281/zenodo.7643745

[106] Zhongming Yu, Guohao Dai, Guyue Huang, Yu Wang, and Huazhong Yang. 2021. Exploiting Online Locality and Reduction Parallelism for Sampled Dense Matrix Multiplication on GPUs. In *39th IEEE International Conference on Computer Design, ICCD 2021, Storrs, CT, USA, October 24-27, 2021*. IEEE, 567–574. https://doi.org/10.1109/ICCD53106.2021.00092

[107] Jie Zhao, Bojie Li, Wang Nie, Zhen Geng, Renwei Zhang, Xiong Gao, Bin Cheng, Chen Wu, Yun Cheng, Zheng Li, Peng Di, Kun Zhang, and Xuefeng Jin. 2021. AKG: Automatic Kernel Generation for Neural Processing Units Using Polyhedral Transformations. In *Proceedings of the 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation* (Virtual, Canada) *(PLDI 2021)*. Association for Computing Machinery, New York, NY, USA, 1233–1248. https://doi.org/10.1145/3453483.3454106

[108] Bojian Zheng, Ziheng Jiang, Cody Hao Yu, Haichen Shen, Joshua Fromm, Yizhi Liu, Yida Wang, Luis Ceze, Tianqi Chen, and Gennady Pekhimenko. 2022. DietCode: Automatic Optimization for Dynamic Tensor Programs. In *Proceedings of Machine Learning and Systems*, D. Marculescu, Y. Chi, and C. Wu (Eds.), Vol. 4. 848–863. https://proceedings.mlsys.org/paper/2022/file/fa7cdfad1a5aaf8370ebeda47a1ff1c3-Paper.pdf

[109] Lianmin Zheng, Chengfan Jia, Minmin Sun, Zhao Wu, Cody Hao Yu, Ameer Haj-Ali, Yida Wang, Jun Yang, Danyang Zhuo, Koushik Sen, Joseph E. Gonzalez, and Ion Stoica. 2020. Ansor: Generating High-Performance Tensor Programs for Deep Learning. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*. USENIX Association, 863–879. https://www.usenix.org/conference/osdi20/presentation/zheng

[110] Ningxin Zheng, Bin Lin, Quanlu Zhang, Lingxiao Ma, Yuqing Yang, Fan Yang, Yang Wang, Mao Yang, and Lidong Zhou. 2022. SparTA: Deep-Learning Model Sparsity via Tensor-with-Sparsity-Attribute. In *16th USENIX Symposium on Operating Systems Design and Implementation (OSDI 22)*. USENIX Association, Carlsbad, CA, 213–232. https://www.usenix.org/conference/osdi22/presentation/zheng-ningxin

[111] Size Zheng, Yun Liang, Shuo Wang, Renze Chen, and Kaiwen Sheng. 2020. FlexTensor: An Automatic Schedule Exploration and Optimization Framework for Tensor Computation on Heterogeneous System. In *ASPLOS '20: Architectural Support for Programming Languages and Operating Systems, Lausanne, Switzerland, March 16-20, 2020*, James R. Larus, Luis Ceze, and Karin Strauss (Eds.). ACM, 859–873. https://doi.org/10.1145/3373376.3378508

[112] Aojun Zhou, Yukun Ma, Junnan Zhu, Jianbo Liu, Zhijie Zhang, Kun Yuan, Wenxiu Sun, and Hongsheng Li. 2021. Learning N: M Fine-grained Structured Sparse Neural Networks From Scratch. In *9th International Conference on Learning Representations, ICLR 2021, Virtual Event, Austria, May 3-7, 2021*. OpenReview.net. https://openreview.net/forum?id=K9bw7vqp_s

Received 2022-07-07; revised 2022-11-03; accepted 2023-01-19