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

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, hardware backends are evolving and becoming heterogeneous, 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 compositability:
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 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 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

---

1We use this term to describe rewriting the program to use Matrix-Multiply Units such as Tensor Cores in GPU and MXU in TPU.
SparseTIR: Composable Abstractions for Sparse Compilation in Deep Learning

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.

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

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.

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(feature_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]
```
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: \( A_{ijk} = A[i,j] \). 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. \( A[i \times m + j, k] \)) and non-affine indices such as integer values loaded from another buffer (e.g. \( B[\text{eid}[i], j \times n + k] \)). 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 composite 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 \( F : (x, i) \rightarrow (x', i') \), 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

²The SparseTIR has round-trip compatibility with Python, and this paper presents only its Python form.
The document discusses SparseTIR, a system for sparse compilation in deep learning. It explains the process of decomposing a matrix multiplication operation and generating new axes and sparse buffers. The text describes the steps involved in transforming stage I IR to stage II IR, focusing on auxiliary buffer materialization, nested loop generation, and coordinate translation. The figure 7 shows how the materialization works in stage II, while figure 8 shows how the nested loops are generated in stage II. The document provides code snippets and explanations for each step.
with sp_iter([K, I, J], “SRR”, “spmm”) as k, i, j:
  # body
  for k in range(k):
    for i in range(m):
      with block(“spmm0”):
        A[i, j] = 0.0
        with block(“spmm1”):
          C[i, k] = C[i, k] + A[i, j] * B[j, k]  # spmm body
          for k in range(k):
            for i in range(m):
              ... in Stage II
            Sparse iteration in Stage I
            Generated nested loop in Stage II
            SpMM w/ fusion
            SDDMM w/ fusion

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.

Stage I IR
with sp_iter([I, J, K], “SRR”, “spmm”) as [i, j, k]:
  with init():
    C[i, j, k] = 0.0
    C[i, k] = C[i, k] + A[i, j] * B[j, k]

for i in range(n):
  with block(“spmm0”):
    for j in range(n, J_indptr[i + 1] - J_indptr[i]):
      with block(“spmm1”):
        C[i, k] = C[i, k] + A[i, j] * B[j, indices[i, j], k]

Stage II IR
with sp_iter([K, I, J], “SRR”, “spmm”) as [k, i, j]:
  # body

for k in range(k):
  for i in range(m):
    with block(“spmm0”):
      A[i, j] = 0.0
      with block(“spmm1”):
        C[i, k] = C[i, k] + A[i, j] * B[j, indices[i, j], k]

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

\[ p_j = f^{-1}(A(buffer), j, \{p\}_{i=1}^{i-1}, I(coord)(c_1, \ldots, c_M)) \] (1)

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

\[ c_i = f(A_{iter}, \{c_i\}_{i=1}^{i-1}, v_i^{(p)}) \] (2)

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

\[ f(-1)(A, j, p, x) \triangleq \begin{cases} x & \text{A}_j: D(ense) \\ \text{find}(A_{j, indices}[p[\text{anc}(A, j), i], x]) & \text{A}_j: S(parse) \end{cases} \] (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 \( A_j \) from its root in axes dependency tree, and \( p[\text{anc}(A, j)] \) gathers values from \( p \) by ancestors’ indices:

\[ \text{anc}(A, i) \triangleq \begin{cases} [i] & A_i \text{ is root} \\ \text{anc}(A, j) : i & A_j = \text{parent}(A_i) \end{cases} \] (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 \(^7\) 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 \( \{A_i\}_{i=1}^n \). For memory access \( A[x_1, \ldots, x_n] \), the overall offset after flattening is computed by:

\[ \sum_{i=1}^n \text{is_leaf}(A_i) \times \text{offset}(i) \times \text{stride}(i + 1), \] (6)

where \( \text{is_leaf}(A_i) \) means that if axis \( A_i \) has no dependence in \( \{A_j\}_{j=i+1}^n \), offset and stride are defined as:

\[ \text{offset}(i) \triangleq \begin{cases} x_i & \text{is_root}(A_i) \\ A_i \text{indptr}[\text{offset}(j)] + x_i & A_j = \text{parent}(A_i) \end{cases} \] (7)

\[ \text{stride}(i) \triangleq \begin{cases} 1 & i > n \\ \text{nnz}(\text{Tree}(A_i)) \times \text{stride}(i + 1) & \text{is_root}(A_i) \\ \text{stride}(i + 1) & \text{otherwise} \end{cases} \] (8)

where \( \text{nnz}(\text{Tree}(A_i)) \) refers to the number of non-zero elements of the sparse iteration space composed by the tree with \( A_i \) 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 \( A[J \text{indptr}[i] + j] \) 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 composorable 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

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

Stage III IR

```python
# loop structures
with block("spmm1"):  
    with init():  
        C[i, k] = 0.0  
        C[i, k] = C[i, 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.

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. For sparse transformers, we select TorchSparse’s block-sparse kernel as our baseline. For SpMM, we select the best performing among them.

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 [76] | 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 $B$.

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

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 search for the best $c$ over $\{1, 2, 4, 8, 16\}$ and let $k = \lceil \log_2 \frac{nzw}{w} \rceil$, 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 1.20-1.9x 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.

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 sparseFuse 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 float2/float2t, 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.
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.

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.

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.

---

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

---

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

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 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/\(b^2\).

4.4 Relational Gather-Matmul-Scatter

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

\[
\text{SR-BCRS}((\lfloor \frac{m}{h} \rfloor, 4)) = A_{\lfloor \frac{m}{h} \rfloor} \times B_{4}
\]

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’ GetMM when weight density is extremely low (e.g., \(\leq 2^{-7}\)).
SparseTIR: Composable Abstractions for Sparse Compilation in Deep Learning

Figure 19: Normalized speedup against 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     | 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     |

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

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 \( W_r \) 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:

\[
T_{r,i,j} = \sum_{k=1}^{d_{in}} X_{j,k} W_{r,k,i},
\]

\[
Y_{i,j} = \sum_{r=1}^{R} \sum_{j=1}^{n} A_{r,i,j} T_{r,i,j},
\]

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 explains the scheduling of RGMS operator on 3D \( hyb \) in SparseTIR: for each ELL matrix \( A_{r,k} \) (\( r \) refers to edge type and \( k \) refers to bucket index), we pin its corresponding weight matrix \( W_r^k \) 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 (\( \geq 128 \)), 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.

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.

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: $\sqrt{C_{in} \times C_{out}}$, and the Y-axis refers to speedup against TorchSparse.

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 inspired by TACO and earlier work, with a focus on Deep Learning operators. Sympler [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] propose data dependence simulation 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 [35] 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. SparT [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. Feastgraph [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 these 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], An- sor [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.
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, Zhijuan Liu for providing Sparse Convolution benchmarks, Joel Emer, Yuwei Hu, Jie Liu, Steven S. Lyubomirsky, Fredrik Kjolstad, Ye Tian, Zhiquiang Xie, and Zhongyuan Zhao for feedback 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.

### ACKNOWLEDGMENTS

This section further explains the programming interface for composable formats: §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 \( A' \): \( A[f] = A'[f'] \). 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 format decomposition in Figure 5:

```python
def spmm():
    a: T.handle, b: T.handle, c: T.handle,
    m: T.int32, n: T.int32, nnz: T.int32
    return FormatRewriteRule(  
        "spmm",  
        a, (m, n, nnz),  
        b, (io * block_size + ii, jo * block_size + ji),  
        " float32")

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
    ) -> None:
        io = T.dense_fixed(m, idtype="int32")
        ii = T.dense_fixed(n, idtype="int32")
        A = T.match_sparse_buffer(a, (io, ii), "float32")
        B = T.match_sparse_buffer(b, (io, ii), "float32")
        C = T.match_sparse_buffer(c, (io, ii), "float32")
        with Tлеп.iter(i, j, k):
            C[i, k] = 0.0
            C[i, k] = C[i, k] + A[i, j] * B[j, k]

        return FormatRewriteRule(  
            "bsr”,  
            a, (io, m, indptr, indices),  
            b, (io * block_size + ii, jo * block_size + ji),  
            " float32")
```

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 hours 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 these 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], and 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:

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

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

```bash
docker image pull expye/sparsetir-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.
- **c2e** 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-a11.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.
SparseTIR: Composable Abstractions for Sparse Compilation in Deep Learning

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