Insum: Sparse GPU Kernels Simplified and Optimized with Indirect Einsums

Jaeyeon Won
Massachusetts Institute of Technology
CSAIL
Cambridge, MA, USA
jaeyeon@csail.mit.edu

Willow Ahrens
Georgia Institute of Technology
Atlanta, GA, USA
ahrens@gatech.edu

Saman Amarasinghe
Massachusetts Institute of Technology
CSAIL
Cambridge, MA, USA
saman@csail.mit.edu

Joel S. Emer
Massachusetts Institute of Technology
CSAIL
Cambridge, MA, USA
NVIDIA
Architecture Research Group
Westford, MA, USA
emer@csail.mit.edu

Abstract

Programming high-performance sparse GPU kernels is notoriously difficult, requiring both substantial effort and deep expertise. Sparse compilers aim to simplify this process, but existing systems fall short in two key ways. First, they are primarily designed for CPUs and rarely produce high-performance GPU code. Second, when computations involve both sparse and dense regions, these compilers often fail to optimize the dense portions effectively. In this paper, we propose a new approach for expressing sparse computations. We start from format-agnostic Einsums over sparse tensors and rewrite them into format-conscious indirect Einsums, which explicitly encode format information by mapping sparse data and metadata onto dense tensor operations through indirect indexing. To execute indirect Einsums, we introduce the **Insum** compiler, which generates efficient GPU code for these Einsums by lowering to the PyTorch compiler, extended to better support Tensor Core–enabled indirect Einsums. We also present two fixed-length sparse formats, **GroupCOO** and **BlockGroupCOO**, designed to fit naturally with indirect Einsums. Our approach achieves 1.14×–3.81× speedups across a range of sparse GPU applications while reducing lines of code by 202×–4491× compared to hand-written implementations. The source code for **Insum** is publicly available at https://github.com/nullplay/IndirectEinsum.

ASPLOS '26, Pittsburgh, PA, USA

ACM ISBN 979-8-4007-2359-9/2026/03
https://doi.org/10.1145/3779212.3790176

CCS Concepts: • **Software and its engineering** → **Compilers**; **Domain specific languages**; • **Computer systems organization** → *Parallel architectures*; *Neural networks.*

Keywords: Tensor compilers; GPU code generation; Sparse tensor compilers; Sparse GPU kernels; Einsums

ACM Reference Format:
Jaeyeon Won, Willow Ahrens, Saman Amarasinghe, and Joel S. Emer. 2026. Insum: Sparse GPU Kernels Simplified and Optimized with Indirect Einsums. In *Proceedings of the 31st ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2 (ASPLOS '26), March 22–26, 2026, Pittsburgh, PA, USA.* ACM, New York, NY, USA, 14 pages. https://doi.org/10.1145/3779212.3790176

1 Introduction

Sparse computations arise naturally in many real-world applications, from machine learning models with graphs [10, 37] to scientific simulations [29]. In these scenarios, storing all values including many zeros leads to inefficient implementation, both in memory and computation. To address this, developers rely on sparse formats that help avoid storing zeros and performing ineffectual computations ($a * 0 = 0$).

At the same time, GPUs have become the hardware backbone of high-performance computing due to their massive parallelism. As demand grows for fast processing, there's a strong incentive to accelerate sparse workloads on GPUs.

Unfortunately, implementing efficient sparse kernels on GPUs is notoriously difficult. Unlike dense computations, sparse operations require indirect memory accesses, irregular data dependencies, and careful handling of load balancing. Developers must select appropriate sparse formats for their use case and manage a wide range of GPU-specific optimizations, including shared memory utilization, load balancing, vectorization, avoiding bank conflicts, exploiting tensor

Name	Struct. SpMM	Unstruct. SpMM	Equivariant Tensor Prod.	Sparse Conv.
TorchBSR[16]	202 LoC	✗	✗	✗
Sputnik[11]	✗	1918 LoC	✗	✗
e3nn[12]	✗	✗	225 LoC	✗
TorchSparse[31]	✗	✗	✗	4491 LoC
Ours	1 LoC	1 LoC	1 LoC	1 LoC
LoC Saving	**202×**	**1918×**	**225×**	**4491×**
Speedup	**1.95×**	**1.20×**	**3.81×**	**1.14×**

Table 1. Comparison of different libraries across applications. While each library is tailored to a specific application, our compiler-based approach supports all of them, achieving better speedups with significantly fewer lines of code.

cores, and memory coalescing. These challenges significantly complicate the development of sparse GPU kernels.

These complexities often lead to specialized, hand-optimized GPU implementations, which are not only time-consuming to write but also difficult to maintain and generalize. The complexity of this problem is evident from the continuous influx of research papers introducing new implementations for sparse GPU workloads. For example, as shown in Table 1, TorchSparse [31] is a state-of-the-art GPU sparse convolution library with over 4,000 lines of CUDA (NVIDIA's low-level GPU programming language). Its implementation is so sophisticated that its underlying strategies have led to at least five separate research papers [15, 31–34], each proposing novel techniques to address specific performance challenges.

Sparse tensor compilers have emerged to simplify the development of sparse kernels [1, 4, 17, 40, 44]. These systems allow developers to express computations as if operating on dense tensors, while automatically generating optimized implementations for sparse formats. Typically, they achieve this separation through two key abstractions: (1) a format-agnostic Einsum [9] that specifies the computation (e.g., $C_i = A_i * B_i$ for elementwise multiplication), and (2) an explicit specification of the sparse storage format (e.g., A stored in a sparse format such as COO or CSR [8]).

Sparse compilers have largely focused on sparse-sparse kernels—operations where multiple sparse inputs interact. These kernels require identifying nonzero values at overlapping (intersection) or combined (union) coordinates, which introduces complex control flow more suitable for CPUs. However, many real-world sparse GPU workloads involve sparse-dense operations, where only one operand is sparse and the rest are dense. Existing sparse compilers often fail to fully optimize the dense computation components common in these workloads, resulting in poor performance.

In this work, we build on the idea of using indirect indexing in Einsums to express sparse computation. Our contribution lies in applying it to encode format information directly into the Einsum, enabling sparse operations to be expressed in a *format-conscious* manner. For example, an elementwise multiplication in COO format is expressed as the following indirect Einsum : $C_{AI_p} = AV_p \times B_{AI_p}$ [1] where AV contains the nonzero values and AI holds their coordinates. This operation gathers elements from B at positions specified by AI and scatters the results back to C.

On top of this, we develop **Insum**, a compiler that lowers these indirect Einsums into high-performance GPU kernels by reusing existing dense tensor compilers. Unlike traditional sparse compilers, which focus on CPU codegen and sparse-specific primitives, Insum leverages dense compilation infrastructure while preserving support for irregular memory access patterns. By fusing indirect accesses with computation, Insum generates sparse GPU kernels from a single high-level statement, bridging the gap between sparse abstraction and dense-level performance.

This paper's contributions are as follows:

- We show how to implement format-agnostic Einsums over sparse tensors by rewriting them into *Indirect Einsums*, which are format-conscious and map the data and metadata of sparse tensor formats onto dense tensor operations through indirect indexing.
- We propose *GroupCOO* and *BlockGroupCOO*, new sparse formats enabling efficient indirect Einsums.
- We present a compiler for indirect Einsums, called *Insum*, which lowers these expressions to dense tensor operations. In this work, we use the PyTorch [3, 25] as the backend, targeting its FX graph [27] intermediate representation.
- We extend PyTorch compiler for better generated code: [2]
  - We enable the compiler to natively generate Tensor Core–enabled matrix multiplication, eliminating the need for hand-written templates and allowing fusion with operations, including gather and scatter.
  - We add a codegen technique called *Lazy Broadcasting* to improve performance when using Tensor Cores.
- Our compiler achieves competitive or superior performance across diverse sparse applications, while requiring significantly fewer lines of code (LoC), as shown in Table 1.

2 Background and Related Works

2.1 Einsums

The Einstein summation [9] provides a compact and expressive notation for describing tensor computations, where operations are expressed as products over indexed expressions. Reductions are implicitly defined over index variables that do not appear in the output expression. In other words, if an index appears in the input tensors but not in the output, the computation reduces over that index. For example, matrix

[1] Usually expressed as `C[AI[p]] = AV[p] * B[AI[p]]` using square brackets in programming languages, since subscript notation is not easily supported in text editors.

[2] This native matmul feature has been incorporated as an official PyTorch feature starting in PyTorch 2.10 and can be enabled by setting `torch._inductor.config.triton.native_matmul = True`.

multiplication is expressed as $C_{m,n} = A_{m,k} \times B_{k,n}$, where the computation reduces over the index k.

In recent years, Einsum notation has gained widespread popularity [5, 7, 13, 14, 17, 19, 20, 23–26, 39], with generalizations that support arbitrary pointwise operations and reduction operators (e.g., `sum`, `min`, `max`) over tensor accesses with complex index expressions, including affine [40] or indirect indexing [26]. Einsums define computation as a traversal over all combinations of index variables, making it a powerful abstraction for dense tensor operations [19].

Importantly, Einsums are not tied to any particular data representation—it serves as an abstract language for expressing computation between tensors. Prior sparse frameworks [1, 4, 17, 19, 42, 43] leverage this property by decoupling the Einsum specification from the storage format [8, 30], allowing users to select appropriate sparse representations for each tensor. Some of these abstractions are grounded in the coordinate tree concept, which was initially introduced in the context of the format abstraction [8] within TACO and subsequently refined and formalized as the *fibertree* abstraction [30]. In this work, we focus on *indirect Einsums*, an extension that allows tensor access to appear within the index expressions of other tensors.

2.2 Implementation of Einsums

2.2.1 Dense Tensor Compilers. A number of dense tensor compilers support high-performance tensor programs using Einsum notation as input. Systems like Halide [26] and TVM [7] introduce scheduling languages that decouple the algorithm from its execution strategy. These frameworks support loop-level transformations such as `split`, `fuse`, and `reorder` to express a wide variety of implementations.

Deep learning frameworks such as PyTorch [25], NumPy [13], and JAX [5] also provide native support for Einsum primitives. PyTorch can compile Einsum expressions through the PyTorch compiler stack [3], which eventually compiles to Triton [36] for GPU execution. Unlike Halide or TVM, the PyTorch compiler does not expose a full scheduling language; instead, it applies a fixed set of heuristics and autotuning over a small search space (e.g., tile size). This design allows for fast compilation while still achieving competitive or better performance than other existing tensor compilers [3].

2.2.2 Sparse Tensor Compilers. Sparse tensor compilers such as TACO [17], Finch [1], and `mlir-sparse` [4] take two inputs: an Einsum and a sparse format specification for each operand. To skip ineffectual (zero) computations, these compilers must generate complex control flow for intersection and union patterns over sparse indices—capabilities not supported by most dense tensor compilers. As a result, these systems often build their own code generation frameworks from scratch, using techniques such as *Looplets* [2] or *merge lattices* [17]. Due to the inherent control-flow complexity, most of these compilers target CPU code generation and tend to under-optimize the dense portions of computation.

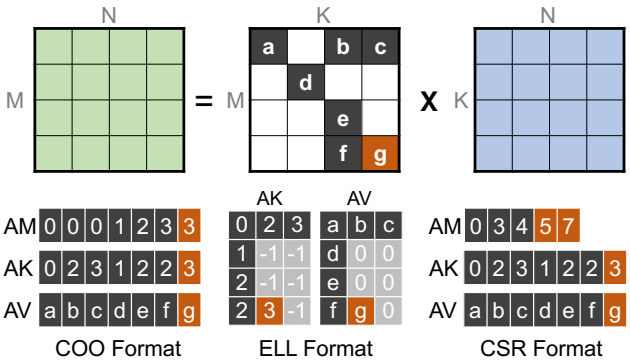


Figure 1. SpMM example ($C_{m,n} = A_{m,k} \times B_{k,n}$) and various sparse formats for the matrix A: COO, ELL, and CSR.

Some recent efforts, such as COMET [35] and SparseTIR [44], aim to build sparse compilers using existing dense compiler infrastructures. COMET, built on MLIR [18], primarily targets CPU code generation. SparseTIR, built on TVM [7], can generate high-performance sparse GPU kernels using TVM's code generation backend and is the most closely related to our work. However, SparseTIR still requires manual scheduling, which demands significant user effort and expertise. We thoroughly compare our approach against existing GPU sparse compilers, SparseTIR and TACO, in Section 6.7.

3 Overview

3.1 Extending Einsums with Indirect Indexing

In this work, we use an extended Einsum notation that supports indirect indexing—indexing via values from other tensors. For example, we allow expressions like $C_{X_i} = A_{Y_{i,j}} \times B_j$, where C_{X_i} and $A_{Y_{i,j}}$ are indirect accesses.

We can implement indirect indexing on the right-hand side using a gather, and on the left-hand side using a scatter. Similar to traditional Einsums, we assume that the output tensor is implicitly initialized to zero, and multiple writes to the same location during the scatter are resolved by summation. This behavior is consistent with the operational semantics of Einsum described in prior work, where overlapping contributions to the same output position are accumulated [19].

Sparse Computation in Indirect Einsums. To avoid ineffectual operations ($a * 0 = 0$), sparse tensors are often stored in compressed formats that record only the nonzero values along with metadata describing their coordinates as shown in Figure 1. These formats make it easier to identify and skip ineffectual operations. The key idea of this paper is to express sparse computations by mapping both the data and metadata of sparse representations onto dense tensor operations using indirect indexing. In doing so, we convert a

Figure 2. COO SpMM : $C_{AM_p,n} = AV_p \times B_{AK_p,n}$

Einsum originally operating over sparse data into an indirect Einsum that operates entirely over dense tensors.

Consider a matrix multiplication expressed in Einsum notation as $C_{m,n} = A_{m,k} \times B_{k,n}$, where A is sparse and B is dense, creating a sparse-dense matrix multiply (SpMM). To avoid storing and multiplying by zeros in A, one can leverage a sparse format such as COO, which stores only the nonzero values of A in a vector `AV`, along with their corresponding row and column coordinates in `AM` and `AK`, respectively. Using this representation, we access only the elements of B that correspond to the nonzero coordinates in A, avoiding ineffectual computation. This strategy corresponds to a leader-follower intersection as described in prior works [2, 43]. This operation can be expressed using an indirect Einsum by accumulating into C as Figure 2.

3.2 Compiler Overview

Figure 3 illustrates the workflow of our compiler. First, users express sparse computations using indirect Einsums (Section 4). To execute indirect Einsums on a GPU, we developed the `Insum` compiler, which leverages the existing compiler for dense tensor operations. `Insum` lowers the indirect Einsum expression into a semantically equivalent PyTorch program using PyTorch primitives such as `torch.index_select` and `torch.einsum` (Section 5.1). Finally, this program is passed to our extended PyTorch compiler (Section 5.2).

4 Fixed-Length Sparse Formats

While the previous section explains how to express sparse computations in COO format using indirect Einsums, not all sparse formats are directly compatible with this approach. A fundamental limitation of the Einsum-based expression model is that the loop bounds for each index variable must be fixed and cannot depend on data values. Many sparse formats, such as CSR format, require data-dependent loop bounds. For instance, SpMM CSR in Figure 1 iterates over rows, and within each row, it iterates over a variable number of nonzeros encoded in `AM`, as shown below:

```
for m in 0..M:
  for p in AM[m]..AM[m+1]: # variable-length loop
    for n in 0..N:
      C[m,n] += Av[p] * B[AK[p],n]
```

Since this kind of variable-length loop cannot be directly expressed in Einsums, we must either store the data in a format like COO or pad nonzeros per row to have uniform length like ELL [28], which is compatible with fixed loop bounds. We refer to such formats as *fixed-length formats*.

The ELL format has the advantage of avoiding explicit storage of coordinates along the row dimension, thereby eliminating the need for scatter operations. However, it can introduce significant padding overhead when the nonzero distribution is uneven. To address this, we propose a fixed-length sparse format called **GroupCOO**, which strikes a balance between COO and ELL by grouping nonzeros in the same dimension. This results in improved performance by reducing padding compared to ELL and requiring less scatter than COO. Additionally, we demonstrate that GroupCOO can be extended to support block-sparse formats as well.

4.1 Grouping

We introduce **grouping**, a technique that helps derive various GroupCOO formats. To illustrate this, we walk through an example of SpMM, $C_{m,n} = A_{m,k} \times B_{k,n}$, where `A` is sparse.

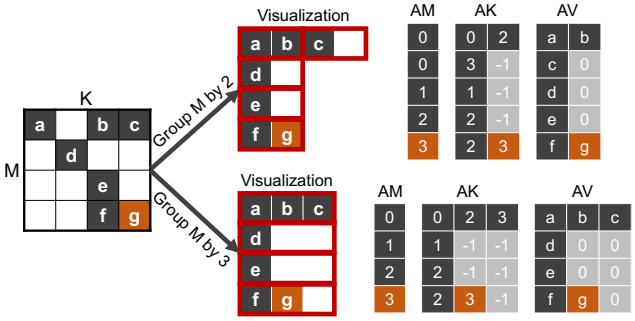


Figure 4. GroupCOO format with grouping on the `M` dimension by group sizes 2 and 3. Grouping with the maximum number of nonzero per row (size = 3) yields the ELL format.

Grouping: While the COO format is simple, it can be inefficient when the same row or column indices appear repeatedly. To reduce this redundancy, we introduce **GroupCOO**, a fixed-length format that partitions nonzeros into **groups** along a chosen dimension (e.g., rows) and stores the group index only once per group. For example, in Figure 4, we group the nonzeros along the `M` dimension (rows) using group sizes of 2 and 3. Within each group, we pad the data if necessary and this padding makes the format fixed-length while reducing redundancy of the same coordinates.

The GroupCOO SpMM can be written as follows, where `p` iterates over groups and `q` over elements within each group:

$$C_{AM_p,\,n} = AV_{p,q} \times B_{AK_{p,q},\,n}$$

GroupCOO sits between the COO format and the ELL format: setting group size to 1 degenerates to the original COO

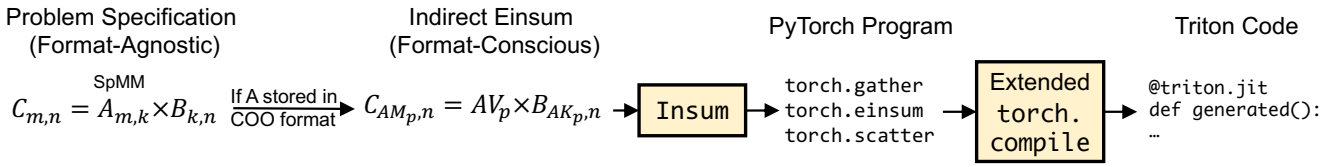


Figure 3. Overview of our system. Instead of expressing sparse computations using format-agnostic Einsums as in prior approaches, users write indirect Einsums in a format-conscious manner. This expression is translated into a PyTorch program by the `Insum` compiler and then lowered to a Triton kernel via our extended `torch.compile` pipeline.

format, while setting group size to the maximum number of nonzeros per row yields the ELL format (Figure 4).

Group+Blocking: Blocking is a common technique in sparse computation to exploit local dense patterns by dividing a sparse matrix into dense blocks [8, 16, 41]. The BlockCOO format stores the coordinates of each nonzero block in `AM` (row block) and `AK` (column block), and the block values in `AV` as a dense tensor of shape `[num_blocks, bM, bK]`. SpMM in BlockCOO is expressed as Figure 5.

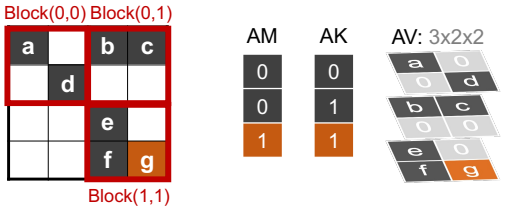


Figure 5. BlockCOO SpMM: $C_{AM_p,bm,n} = AV_{p,bm,bk} \times B_{AK_p,bk,n}$, where `bm` and `bk` index within each block.

Grouping can also be applied to block formats. In **BlockGroupCOO**, we group along one block-coordinate dimension, so that `AV` has shape `[num_groups, group_size, bM, bK]`, and `AM`/`AK` store group and block-level coordinates. SpMM in BlockGroupCOO is expressed as Figure 6.

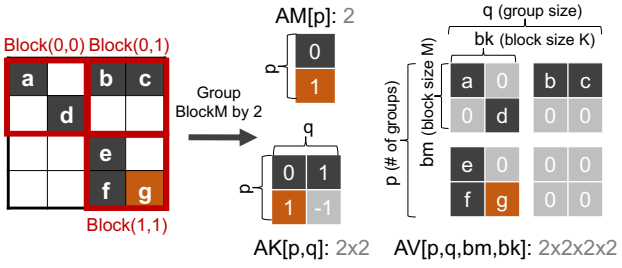
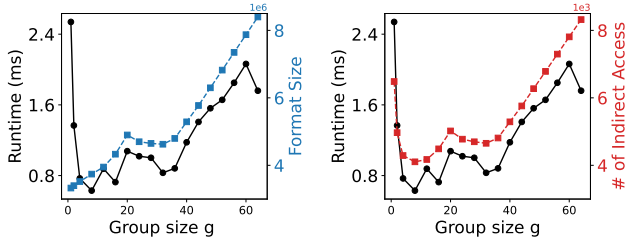


Figure 6. BlockGroupCOO SpMM : $C_{AM_p,bm,n} = AV_{p,q,bm,bk} \times B_{AK_{p,q},bk,n}$, where `p` indexes groups, `q` entries within each group, and `bm`, `bk` index each block.



(a) Runtime vs. Format Size **(b)** Runtime vs. # of Ind. Access

Figure 7. Runtime correlation with indirect access and format size under grouping. The number of indirect accesses (i.e., gathers and scatters) shows a stronger correlation with runtime performance than the total memory required.

4.2 Choosing Group Size

Choosing an appropriate group size g is crucial for both storage footprint and compute efficiency. If g is too large, each row is padded to the next multiple of g, leading to wasted memory. If g is too small, the format degenerates into standard COO, losing the benefits of grouping. Ideally, we want to choose the group size g that yields the best runtime.

Our experiments suggest that minimizing *indirect memory accesses*, i.e., gathers and scatters, is the key to achieving high performance. These indirect accesses often result in expensive DRAM transactions due to poor spatial locality. In Figure 7, we run BlockGroupCOO SpMM on a 4096×4096 block-sparse matrix with 32×32 dense blocks at 80% sparsity, sweeping various group sizes g.

Another proxy we tested for the optimal g is the one that minimizes total storage (i.e., the size of AM, AK, and AV in Figure 6), thereby reducing memory footprint. However, as shown in Figure 7a, the format memory increases almost monotonically with larger g, especially in BlockGroupCOO, where padding in AV is influenced by block size. So, we found runtime does not correlate well with the format size.

In contrast, Figure 7b demonstrates that runtime is closely aligned with the total number of *gathers and scatters* (i.e.,

accesses to AM and AK), which defined as $F(g)$:

$$F(g) = \underbrace{\sum_{i=0}^{n-1} \left\lceil \frac{occ_i}{g} \right\rceil}_{\text{AM: Scatter}} + \underbrace{g \sum_{i=0}^{n-1} \left\lceil \frac{occ_i}{g} \right\rceil}_{\text{AK: Gather}} = (g+1) \sum_{i=0}^{n-1} \left\lceil \frac{occ_i}{g} \right\rceil.$$

Here, n is the number of rows in the matrix and occ_i is the number of nonzeros in row i (Figure 4 gives $occ = [3, 1, 1, 2]$). With group size g, each row i is divided into $\lceil occ_i/g \rceil$ groups.

Finding the exact optimal g over the range $[1, \max_i occ_i]$ requires brute-force evaluation of the $F(g)$ above, leading to $O(n \cdot \max_i occ_i)$ complexity. To obtain a much faster, yet nearly optimal estimate, we replace the ceiling with a conservative approximation: $\lceil occ_i/g \rceil \approx occ_i/g + 1$.

Letting $S = \sum_i occ_i$, the relaxed cost becomes:

$$\tilde{F}(g) = (g+1)\left(\frac{S}{g} + n\right) = S + \frac{S}{g} + ng + n.$$

Treating g as a real variable and differentiating:

$$\frac{d\tilde{F}}{dg} = -\frac{S}{g^2} + n = 0 \quad \Rightarrow \quad g^\star = \sqrt{\frac{S}{n}}.$$

This estimate provides a good approximation to the group size that minimizes indirect accesses and likely achieves optimal runtime. In practice, we round $g^\star$ to the nearest power-of-two values and select the one with the best runtime. This choice is motivated by the fact that Triton, our backend, performs best with power-of-two block sizes, as evidenced by the downward spikes in Figure 7. This heuristic consistently yields efficient configurations across applications (Section 6).

5 Compiler Implementation

This section describes our approach for generating efficient GPU code from indirect Einsum expressions. We leverage a dense tensor compiler that produces high-performance code and supports gather/scatter operations. Among available options, we selected the PyTorch compiler [3] for four reasons: (1) it is Python-native and user-friendly; (2) it supports automatic differentiation, essential for sparse deep learning; (3) it targets Triton, enabling Tensor Core support without manual scheduling [36]; and (4) it delivers state-of-the-art performance among existing dense tensor compilers [3].

5.1 Insum: Rewriting Indirect Einsums in PyTorch

To use the PyTorch compiler effectively, we first need to translate indirect Einsum expressions into an equivalent PyTorch program. To facilitate this, we introduce the `Insum` compiler, which converts indirect Einsum strings into standard PyTorch operations. The interface for `Insum` is as follows:

```
Insum(expression: str, **tensors: dict)
```

Here, `expression` is a string representing the indirect Einsum computation, and `tensors` is a dictionary mapping tensor names to their corresponding PyTorch tensors. For example, an indirect Einsum $C_{D_y,x} = A_{y,E_r} \times B_{r,x}$ can be expressed as `Insum("C[D[y],x] += A[y,E[r]] * B[r,x]", C = C, A = A, B = B, D = D, E = E)`.

The `Insum` function works by parsing the provided string expression and constructing an intermediate representation known as an FX graph [27]. In the PyTorch compiler stack, an FX graph is a functional representation of a PyTorch program that clearly encodes tensor operations and their dependencies. Since PyTorch lacks a native primitive for indirect Einsums, we need to translate it into equivalent PyTorch operations. The basic workflow for this translation involves three main steps:

1. **Gather Inputs**: If the right-hand side involves indirect indexing, gather elements from required tensors using operations like `torch.index_select` or `torch.gather`.
2. **Execute Einsum**: Perform the Einsum (i.e., `torch.einsum`) computation using the gathered data.
3. **Scatter Outputs**: If the left-hand-side expression involves indirect indexing, scatter the computed results to their target locations using operations like `torch.index_add`.

For the above example, `Insum` will create an FX graph equivalent to the following PyTorch program:

```
# (1) Gather: Atmp[y,r] = A[y,E[r]]
Atmp = torch.index_select(A, dim=1, index=E)
# (2) Einsum: Ctmp[y,x] = Atmp[y,r] * B[r,x]
Ctmp = torch.einsum("yr,rx->yx", Atmp, B)
# (3) Scatter: C[D[y],x] += Ctmp[y,x]
C.index_add_(dim=0, index=D, source=Ctmp)
```

5.2 Native Matrix Multiply Support in TorchInductor

Once we construct an FX graph, we pass it to TorchInductor—the code generation backend of the PyTorch compiler. TorchInductor lowers the FX graph into optimized code by first translating it into an intermediate representation called *InductorIR*, a loop-level IR that describes how operations are executed through nested loops. At this level, TorchInductor performs optimizations such as loop tiling, reordering, and fusion before generating final Triton codes.

Limitation: A core optimization in TorchInductor is the *fusion* of multiple PyTorch operations into a single loop nest. TorchInductor can fuse many pointwise and reduction operations into a single Triton kernel. However, there is a significant exception: matrix multiplication.

Due to its critical role in deep learning workloads, matrix multiplication is not generated natively by TorchInductor. Instead, it relies on a hand-optimized Triton template. While this template allows limited fusion of pointwise operations, it prevents complex operations, such as gather and scatter, from being fused. This lack of fusion can result in increased memory traffic, reduced locality, and degraded performance for computations involving matrix multiplication.

When the FX graph for `"C[D[y],x] += A[y,E[r]] * B[r,x]"` is passed to TorchInductor, it generates three separate (i.e., not fused) Triton kernels: one for gathering `A`, one using a matrix multiplication template (`Ctmp[y,x] = Atmp[y,r] * B[r,x]`), and one for scattering to `C`.

To overcome the limitations of fixed templates, we extend TorchInductor's codegen pass to emit the Triton `tl.dot` intrinsic, which maps directly to Tensor Core operations. This enables the compiler to natively generate Tensor Core optimized kernels without relying on hand-written templates. Implemented directly on TorchInductor's loop-level IR, our approach integrates seamlessly with its fusion infrastructure, allowing fully fused kernels for indirect Einsums.

5.2.1 Forcing TorchInductor to Generate Matmul.

While TorchInductor uses a hand-written Triton template for matrix multiplication, it can also be configured to generate it natively. We build on this capability in our compiler extension.

Consider the PyTorch matrix multiplication example (`C = torch.matmul(A,B)`) with matrices of shape $(2048, 2048)$. To prompt TorchInductor to generate matrix multiplication kernels without a template, a useful trick is to rewrite the operation explicitly as replicated multiplication (i.e., batched Cartesian product) followed by summation :

```
# Prod[y,r,x] = A[y,r] * B[r,x]
Prod = A.unsqueeze(2) * B.unsqueeze(0)
# C[y,x] = Prod[y,r,x]
C = Prod.sum(dim=1)
```

When provided with this rewritten computation, TorchInductor generates the Triton kernel depicted in Figure 8a. However, the resulting kernel suffers from performance limitations due to two critical issues:

- **Lack of Tensor Core Utilization:** TorchInductor generates naive multiplication and summation (as seen on the green-highlighted code) instead of leveraging Triton's optimized `tl.dot` operation for tensor cores.
- **Absence of Proper Tiling:** TorchInductor currently flattens all pointwise indices—those not involved in the reduction—into a single loop dimension. This approach often benefits from the GPU's abundant parallelism by enlarging the parallelizable loop. However, it can also negatively impact memory locality. In particular, output dimensions y and x, which are critical for memory access patterns in matrix multiplication, are flattened together. This behavior is evident in the loop-level InductorIR:

```
for yx:
  y, x = yx // 2048, yx % 2048
  for r:
    Prod[y,r,x] = A[y,r] * B[r,x] # multiply
    C[y,x] = Prod[y,r,x] # summation
```

5.2.2 Natively Invoking Tensor Cores with `tl.dot`.

To achieve optimal performance on GPUs, the generated code must leverage Triton's intrinsic `tl.dot` and apply efficient two-dimensional tiling over the output matrix. To support this, we introduce a new IR node in InductorIR called `ops.dot`, which explicitly represents matrix multiplication.

Although `ops.dot` has the same semantics as broadcast multiplication followed by summation, it is lowered differently during code generation: it maps directly to `tl.dot`. We implemented a rewrite pass that detects patterns of broadcast multiplication followed by summation, and replaces them with an `ops.dot` node.

When this node is present, Inductor explicitly tiles the computation along the pointwise variables y and x, producing the following tiled IR:

```
for y:
  for x:
    for r:
      # Equivalent to C[y,x] += A[y,r] * B[r,x]
      ops.dot(C[y, x], A[y, r], B[r, x])
```

Figure 8b shows the generated code using the tiled IR with `ops.dot`. Compared to the naive version in Figure 8a, there are two notable differences:

1. **Tiled y and x (Lines 8–13):** The computation is now tiled over three explicit axes: y, x, and r, with y and x parallelized. This replaces the earlier flattened `yx` index.
2. **Tensor Core Invocation via `tl.dot` (Lines 23–28):** Instead of manually multiplying and summing over the reduction axis, Inductor now emits a `tl.dot` operation from `ops.dot`, enabling the use of Tensor Cores. The `tl.dot` intrinsic expects its operands to have shapes (Y, R) and (R, X), producing an output of shape (Y, X). To meet these shape requirements, the input tensors are first reshaped to (`YBLOCK`, `RBLOCK`) and (`RBLOCK`, `XBLOCK`) using `tl.view`, and the second operand is transposed with `tl.trans`. The accumulator, initialized in line 16, excludes the reduction dim `RBLOCK`, following an output-stationary dataflow to retain the partial sums directly in the output tile.

5.2.3 Lazy Broadcasting.

While adding `ops.dot` and tiling the pointwise dimensions already allows us to generate a fully fused gather-scatter kernels with Tensor Core, we found that an additional optimization—*Lazy Broadcasting*—is necessary to reach peak performance.

By default, when Inductor lowers IR to Triton code, it uses what we refer to as *Eager Broadcasting*: every loop variable is assigned a unique axis in the Triton block dimensions up front. For example, in the yellow-highlighted code in Figure 8b, `YBLOCK`, `XBLOCK`, and `RBLOCK` correspond to y, x, and r, shaped as `[:,None,None]`, `[None,:,None]`, and `[None,None,:]`, respectively. This eager broadcasting is convenient for code generation, as it simplifies indexing

(a) Generated Code with Default Inductor

```
1  @triton.jit
2  def triton_v1(A, B, C):
3    YXBLOCK : tl.constexpr = 32*32
4    RBLOCK : tl.constexpr = 32
5    yxoffset = tl.program_id(0) * YXBLOCK
6    yx = yxoffset +
7         tl.arange(0, YXBLOCK)[:,None]
8    r_base =
9         tl.arange(0, RBLOCK)[None,:]
10
11   y = yx // 2048
12   x = yx % 2048
13
14   # (YX,R)
15   acc = tl.full([YXBLOCK, RBLOCK], 0.0)
16   for r_offset in range(0,2048,RBLOCK):
17     r = r_offset + r_base
18     # (YX,R)
19     A_yr = tl.load(A + 2048*y + r)
20     # (YX,R)
21     B_rx = tl.load(B + 2048*r + x)
22     acc += A_yr * B_rx
23
24   # (YX,R) -> (YX,1)
25   acc = tl.sum(acc, 1)[:, None]
26   tl.store(C + yx, acc)
```

(b) Generated Code with Native Matmul

```
1  @triton.jit
2  def triton_v2(A, B, C):
3    YBLOCK : tl.constexpr = 32
4    XBLOCK : tl.constexpr = 32
5    RBLOCK : tl.constexpr = 32
6    yoffset = tl.program_id(1) * YBLOCK
7    xoffset = tl.program_id(0) * XBLOCK
8    y = yoffset +
9        tl.arange(0,YBLOCK)[:,None,None]
10   x = xoffset +
11       tl.arange(0,XBLOCK)[None,:,None]
12   r_base =
13       tl.arange(0,RBLOCK)[None,None,:]
14
15   # (Y,X)
16   acc = tl.full([YBLOCK,XBLOCK], 0.0)
17   for r_offset in range(0,2048,RBLOCK):
18     r = r_offset + r_base
19     # (Y,1,R)
20     A_yxr = tl.load(A + 2048*y + r)
21     # (1,X,R)
22     B_yxr = tl.load(B + 2048*r + x)
23     # (Y,1,R) -> (Y,R)
24     A_yr=tl.view(A_xyr,[YBLOCK,RBLOCK])
25     # (1,X,R) -> (X,R)
26     B_xr=tl.view(B_xyr,[XBLOCK,RBLOCK])
27     # (Y,R) x (R,X) -> (Y,X)
28     acc += tl.dot(A_yr,tl.trans(B_xr))
29   acc = acc[:, :, None] #(Y,X)->(Y,X,1)
30   tl.store(C + 2048*y+x, acc)
```

(c) Generated Code with Lazy Broadcasting

```
1  @triton.jit
2  def triton_v3(A, B, C):
3    YBLOCK : tl.constexpr = 32
4    XBLOCK : tl.constexpr = 32
5    RBLOCK : tl.constexpr = 32
6    yoffset = tl.program_id(1) * YBLOCK
7    xoffset = tl.program_id(0) * XBLOCK
8    y = yoffset +
9        tl.arange(0, YBLOCK)[:,None]
10   x = xoffset +
11       tl.arange(0, XBLOCK)[None,:]
12   r_base =
13       tl.arange(0, RBLOCK)
14
15   # (Y,X)
16   acc = tl.full([YBLOCK, XBLOCK], 0.0)
17   for r_offset in range(0,2048,RBLOCK):
18     r = r_offset + r_base
19     # (Y,R)
20     A_yr = tl.load(A+2048*y+r[None,:])
21     # (R,X)
22     B_rx = tl.load(B+2048*r[:,None]+x)
23     # (Y,R) x (R,X) -> (Y,X)
24     acc += tl.dot(A_yr, B_rx)
25
26   tl.store(C + 2048*y+x, acc)
```

Figure 8. Triton code generated by TorchInductor for the matrix multiply between two 2048 × 2048 matrices, `C[y,x] = A[y,r] * B[r,x]`. (a) Default code generation without our compiler optimization. (b) Code generation with Native Matmul: introduces a new `ops.dot` IR node that enables Tensor Core (`tl.dot`) and applies 2D tiling over output dimensions. (c) Code generation with Lazy Broadcasting: eliminates redundant reshaping and broadcasting by delaying layout expansion until needed.

logic—each tensor can be accessed easily without needing to reason about broadcasting behavior at load time.

However, a major downside is that `tl.dot` expects operands in a very specific layout: $(Y, R) \times (R, X)$. The eager broadcasting breaks this layout, requiring explicit reshaping and transposing before the `tl.dot` call. We found that this introduces runtime overhead on reshaping and transposing.

To eliminate this overhead, we introduce *Lazy Broadcasting*. Rather than broadcasting every loop variable with a unique axis from the start, we delay broadcasting until it is actually required. This allows us to match the expected layout of `tl.dot`, without any reshaping or transposing.

Figure 8c shows the generated code after applying Lazy Broadcasting. Two key differences from Figure 8b:

1. **Minimal Eager Broadcasting (Lines 8–13):** Loop variables whose broadcasting dimensions stay constant are expanded once at the beginning. Only y and x are shaped as `tl.arange(0, YBLOCK)[:, None]` and `tl.arange(0, XBLOCK)[None, :]`, while r remains 1D.
2. **Lazy Broadcasting (Lines 19–22):** We apply broadcasting to r on demand, depending on its usage. For example, when loading `A[y, r]`, we use `r[None, :]` to form shape `(YBLOCK,RBLOCK)`; when loading `B[r, x]`, we use `r[:, None]` to form `(RBLOCK,XBLOCK)`. This ensures that the operands passed to `tl.dot` are already in the correct shape, removing the need for views or transposes.

To implement lazy broadcasting, we maintain the block shape metadata for every Triton variable during codegen. At kernel entry, only the output indices (`y` and `x`) are eagerly materialized as a 2D block, while the reduction index `r_base` is kept as a 1D block. As new variables are generated, their block shape is inferred from the operands they depend on. Comments in Figure 9 show the block shape of each variable.

While tracking block shapes during code generation, whenever a variable with 1D shape `(RBLOCK,)` interacts with a 2D variable shaped with `YBLOCK` or `XBLOCK`, we apply lazy broadcasting to the 1D variable along the required axis. For instance, when compiling $C_{D_{y,x}} = A_{y,E_r} \times B_{r,x}$ in Figure 9,

```
@triton.jit
def generated_triton(A, B, C):
 yoffset = tl.program_id(1) * YBLOCK
 xoffset = tl.program_id(0) * XBLOCK

 y = yoffset + tl.arange(0, YBLOCK)[:, None] # (Y,1)
 x = xoffset + tl.arange(0, XBLOCK)[None, :] # (1,X)
 r_base = tl.arange(0, RBLOCK) # (R,)

 acc = tl.full([YBLOCK, XBLOCK], 0.0) # (Y,X)
 for r_offset in range(0, 2048, RBLOCK):
  r = r_offset + r_base # (R,)
  E_r = tl.load(E + r) # (R,)
  A_yr = tl.load(A + (E_r[None,:] + 2048*y)) # (Y,R)
  B_rx = tl.load(B + (x + 2048*r[:,None])) # (R,X)
  acc += tl.dot(A_yr, B_rx) # (Y,X)

 D_y = tl.load(D + y) # (Y,1)
 tl.atomic_add(C + (x + 2048 * D_y), acc) #(Y,X)
```

Figure 9. Generated Triton code for `C[D[y],x] += A[y,E[r]] * B[r,x]`, showing full fusion of gather, matmul, and scatter using Tensor Cores. Yellow-highlighted lines indicate where lazy broadcasting occurs.

we defer broadcasting of 1D variables `r` and `E_r` until they are needed in loading `A[y,E[r]]` or `B[r,x]`. As a result, the inputs to `tl.dot` naturally take the shape `(YBLOCK,RBLOCK)` and `(RBLOCK,XBLOCK)`, enabling direct use of Tensor Cores without requiring extra reshaping or transposition.

6 Evaluation

In this section, we evaluate our generated sparse GPU kernels using `Insum` using our PyTorch compiler extension. We focus on the following five research questions:

- **Q1:** How universal is our indirect Einsum model across different sparse applications?
- **Q2:** How does the performance of our generated code compare to hand-written, state-of-the-art kernels?
- **Q3:** How does grouping or blocking in the fixed-length sparse format affect performance?
- **Q4:** How does our new code generation, including native matrix multiplication and lazy broadcasting, affect performance?
- **Q5:** How does our compiler compare to existing sparse GPU compilers?

We evaluate **Q1** and **Q2** by comparing our generated code against hand-optimized kernels across four domains. We address **Q3** and **Q4** through an ablation study in Section 6.6. Finally, we answer **Q5** by comparing our compiler to existing compilers across various dimensions in Section 6.7.

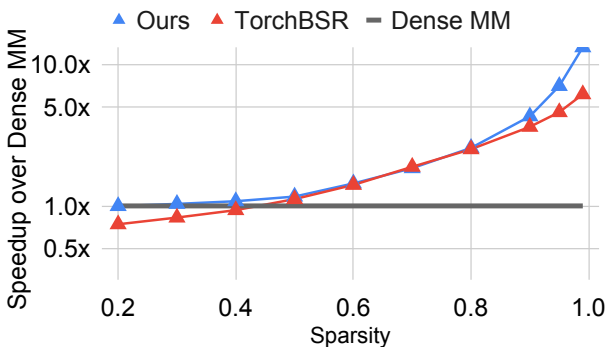


Figure 10. Speedup of our kernel vs. TorchBSR on FP16.

6.1 Experimental Setup

`Insum` comprises approximately 500 lines of code, and the corresponding modifications to TorchInductor add about 1,600 lines. Our implementation is based on a patched version of PyTorch (commit `e8304f0`).[3] All experiments are conducted on an RTX 3090 (24 GB, Ampere). All group sizes used by the format are selected using the heuristic described in Section 4.2.

Case Studies: We evaluate our compiler across four different sparse kernels commonly used in deep learning. For each application, we compare our generated code against state-of-the-art implementations, which often involve significant engineering effort and expert-written Triton or CUDA.

- **Structured SpMM:** A sparse matrix-dense matrix multiplication (SpMM), where the sparse matrix exhibits structured sparsity—specifically, block sparsity.
- **Unstructured SpMM:** Similar to Structured SpMM, but the sparse matrix features unstructured sparsity.
- **Point Cloud Sparse Convolution:** A sparse convolution operation over 3D point clouds or sparse voxels.
- **Equivariant Tensor Product:** A fully connected tensor product used in equivariant neural networks.

6.2 Case Study: Structured (Block Sparse) SpMM

In this case study, we compare our compiler-generated block sparse matrix multiplication (SpMM) kernels with the state-of-the-art TorchBSR kernel [16], a hand-written Triton implementation integrated into PyTorch. TorchBSR requires 202 lines of Triton code, while our approach expresses in a single-line Einsum. We use the BlockGroupCOO format (Section 4) with $(32, 32)$ block size and grouping along block rows. Figure 10 shows FP16 performance on 4096×4096 matrices, reported as speedup over dense matmul.

Our approach shifts the crossover point, where sparse outperforms dense, from about 40% to 25% sparsity. Our kernels consistently match or surpass TorchBSR, with a larger

[3] Our implementation has been upstreamed to PyTorch 2.10 and can be enabled by `torch._inductor.config.triton.native_matmul = True`.

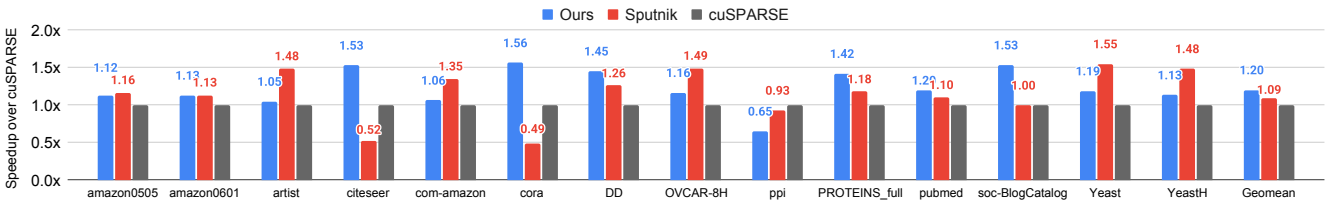


Figure 11. Comparison of our compiler-generated SpMM kernel, Sputnik and cuSPARSE on various sparse matrices.

advantage at high sparsity. This is mainly because TorchBSR relies on the BCSR format, a block variant of CSR. Like CSR, BCSR incurs an $O(N)$ row-pointer overhead, where N is the number of rows; even empty rows require storage and traversal. In hypersparse matrices, where most rows contain no nonzeros, this overhead becomes dominant [6]. In contrast, BlockGroupCOO is COO-based and stores only nonzero blocks, eliminating per-row overhead and scaling more efficiently in the hypersparse regime.

6.3 Case Study: Unstructured SpMM

In this study, we compare our compiler-generated unstructured SpMM kernel with two state-of-the-art hand-written CUDA kernels: Sputnik [11] and cuSPARSE [22]. While Sputnik implements SpMM in approximately 2,000 lines of code and cuSPARSE's implementation details are unavailable due to NVIDIA's proprietary nature, our approach requires only a single Einsum expression (using GroupCOO format).

The operation we evaluate is $C_{m,n} = A_{m,k} \times B_{k,n}$, where A is an unstructured sparse matrix. We selected real-world sparse matrices (Table 2) from TC-GNN datasets [38] with various sparsity patterns, including skewed distributions of nonzeros per row and matrices with large numbers of rows and nonzeros. For our experiments, we set the number of columns in C to 128.

Data	Rows	NNZ	Avg	Med	Max
soc-BlogCatalog	88784	2093195	73.1	6	9429
com-amazon	334863	1851744	5.5	4	549
Yeast	1710902	3636546	2.1	2	6
artist	50515	1638396	32.4	13	1469
PROTEINS_full	43466	162088	3.7	4	25
DD	334925	1686092	5.0	5	19
citeseer	3327	9228	2.8	2	99
OVCAR-8H	1889542	3946402	2.1	2	5
ppi	56944	818716	15.6	8	582
amazon0505	410236	4878874	11.9	10	2760
amazon0601	403394	3387388	8.4	5	2751
cora	2708	10556	3.9	3	168
YeastH	3138114	6487230	2.1	1	6
pubmed	19717	88651	4.5	2	171

Table 2. Sparse matrix statistics: number of rows (= columns), number of nonzeros (NNZ), and NNZ distribution per row.

Figure 11 compares the performance of different kernels on FP32 inputs, reported as speedup relative to cuSPARSE. Our kernel achieves the highest average performance, running 1.2× faster than cuSPARSE, compared to 1.09× for Sputnik. For unstructured sparsity, however, no single kernel dominates across all test cases, since performance depends heavily on the distribution of nonzeros. Sputnik benefits from its row–permutation strategy that groups rows with similar nonzero counts, which gives it an advantage on datasets with highly skewed distributions (e.g., artist).

We also evaluated FP16 inputs. These results are omitted from the figure because Sputnik provides only limited FP16 support. It can process matrices with fewer than 2^{16} rows, but cannot handle the large matrices in our test suite. In contrast, our kernel runs robustly across all benchmarks and, on average, performs 1.18× better than cuSPARSE in FP16.

Our kernel consistently delivers the best average performance. Importantly, it achieves this with a succinct, compiler-generated implementation, in contrast to Sputnik's complex, hand-optimized code. Furthermore, our approach supports a broader range of problem specifications, including FP16, highlighting the effectiveness of compiler-based approach.

6.4 Case Study: Point Cloud Sparse Convolution

In this case study, we evaluate sparse convolution on point clouds and compare our generated kernel against TorchSparse [31], a state-of-the-art library. TorchSparse implements sparse convolution using two different algorithms: (1) ImplicitGEMM [34] and (2) Fetch-on-Demand [15], each relying on distinct sparse formats and separate CUDA codebases.

The point cloud sparse convolution can be expressed as:

$$Out_{x,m} = Map_{x,y,z} \times In_{y,c} \times Weight_{z,c,m}$$

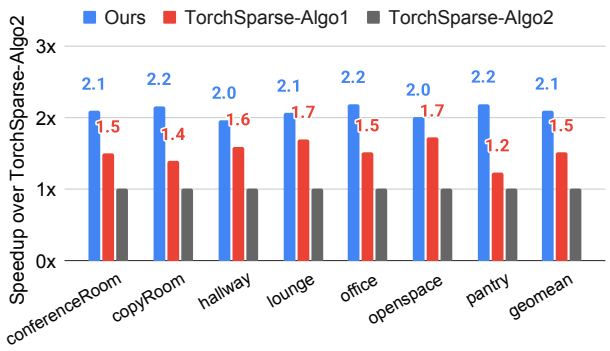
Here, c and m denote input and output channels, respectively. In and $Weight$ are dense tensors and Map is a 3D sparse tensor. If we store Map in COO format, with its nonzero indices stored in $MAPX, MAPY, MAPZ$, and corresponding values in MAPV, the sparse convolution becomes:

$$Out_{MAPX_p,m} = MAPV_p \times In_{MAPY_p,c} \times Weight_{MAPZ_p,c,m}$$

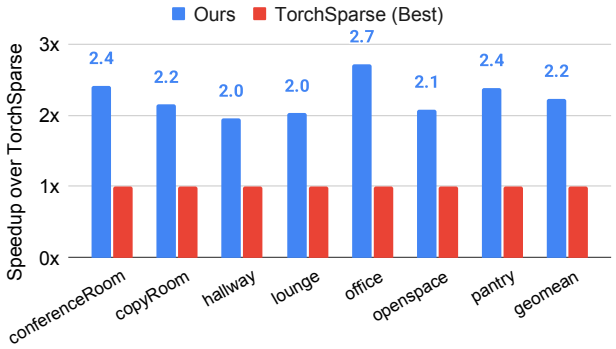
We can further optimize this by grouping entries by $MAPZ$, yielding the following einsum form:

$$Out_{MAPX_{p,q},m} = MAPV_{p,q} \times In_{MAPY_{p,q},c} \times Weight_{MAPZ_p,c,m}$$

This includes a batch matmul between Out, In, and $Weight$ (i.e., $pqm \leftarrow pqc \times pcm$), enabling the use of Tensor Cores.



(a) Speedup of our method vs. TorchSparse-Algo1 and TorchSparse-Algo2 for FP16 inputs on RTX3090 (Ampere).



(b) Speedup of our method vs. the best of TorchSparse-Algo1 and Algo2 for FP16 inputs on H100 (Hopper).

Figure 12. Comparison of our compiler-generated sparse convolution kernels with TorchSparse on two different GPUs. TorchSparse implements sparse convolution in two variants: TorchSparse-Algo1 (referred to as ImplicitGEMM) and TorchSparse-Algo2 (referred to as Fetch-on-Demand).

While TorchSparse requires over 4000 lines of CUDA code to implement and optimize these sparse convolution kernels across data types and channel sizes, our approach achieves the same computation using a single einsum expression.

Figure 12 shows performance results for channel size 128 on seven real-world indoor point clouds from the S3DIS dataset (Area 6). We used a voxel size of 5 cm for quantization, as described in prior work [40]. Our method consistently outperforms both TorchSparse algorithms on FP16. In Figure 12b, we evaluate our generated code on an HBM-equipped GPU (NVIDIA H100) and observe consistent performance improvements over hand-written kernels. Our kernel achieves higher speedups on Hopper than on Ampere, primarily because TorchSparse is optimized for the Ampere architecture but not for Hopper, whereas our compiler-generated Triton code can automatically adapt and autotune for Hopper. This highlights the power of compiler-based approach compared to highly specialized handwritten implementations.

$\ell_{\max}$	Channels	Ours	cuequivariance	e3nn
1	16	8.3×	2.6×	1.0×
	32	4.2×	1.5×	1.0×
	64	2.3×	0.9×	1.0×
2	16	5.2×	1.1×	1.0×
	32	5.4×	1.1×	1.0×
	64	3.3×	0.5×	1.0×
3	16	2.6×	0.5×	1.0×
	32	3.6×	0.6×	1.0×
	64	2.5×	0.3×	1.0×

Table 3. Speedup of our method vs. cuequivariance and e3nn. Speedups are reported relative to e3nn (normalized performance). All experiments are conducted in FP32.

6.5 Case Study: Equivariant Tensor Product

In this case study, we implement a fully connected tensor product in an equivariant neural network [12] using our compiler. The fully connected tensor product—also referred to as the `uvw` mode—can be written as:

$$Z_{b,i,w} = CG_{i,j,k,l} \times X_{b,j,u} \times Y_{b,k} \times W_{b,l,u,w}$$

Here, CG is a sparse 4D tensor, while all other tensors (X, Y, W, Z) are dense tensors. The indices i, j, k, l in CG determine the interaction pattern among the input tensors, while u and w denote input/output channels and b denotes the batch dimension. We store CG in COO format using index arrays CGI, CGJ, CGK, CGL and corresponding nonzero values CGV, which allows us to rewrite the computation as:

$$Z_{b,CGI_p,w} = CGV_p \times X_{b,CGJ_p,u} \times Y_{b,CGK_p} \times W_{b,CGL_p,u,w}$$

To further optimize, we group entries by `CGL`, leading to:

$$Z_{b,CGI_{p,q},w} = CGV_{p,q} \times X_{b,CGJ_{p,q},u} \times Y_{b,CGK_{p,q}} \times W_{b,CGL_p,u,w}$$

This exposes a batched matrix multiply pattern between `Z`, `X`, and `W` of the form: $(bp)qw \leftarrow (bp)qu \times (bp)uw$. Here, the batch dimension is `bp` and the reduction dimension is `u`, allowing our compiler to apply Tensor Core acceleration.

We compare our grouped-COO einsum formulation against two hand-written libraries: `cuequivariance`[21] and `e3nn`[12]. All experiments are conducted with a batch size of 10,000. To control the sparsity pattern of the `CG` tensor, we use real Clebsch-Gordan (CG) coefficients corresponding to different values of a hyperparameter $\ell_{\max}$.

As shown in Table 3, while `cuequivariance` occasionally outperforms `e3nn`, our method consistently achieves higher speedups across all tested channel sizes and $\ell_{\max}$ values—up to 8.3× faster and at least 2× in every setting.

6.6 Ablation Study

Figure 13 shows an ablation study on structured SpMM to evaluate the impact of format selection and compiler optimizations. All matrices are of size 4096×4096, with sparse matrix having 90% uniform sparsity using 32×32 dense blocks.

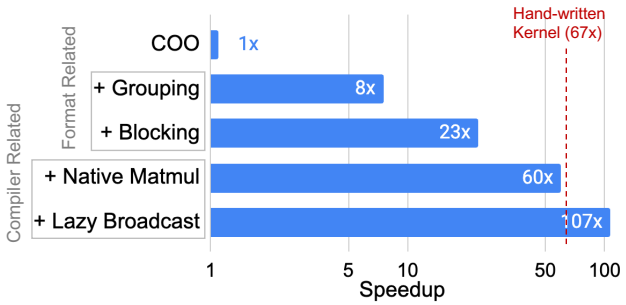


Figure 13. Ablation study on structured SpMM. It shows how each optimization contributes to speedup over the baseline (COO without fusion). When all format and compiler optimizations are enabled, the performance surpasses that of the hand-written kernel (TorchBSR).

Impact of Format: We store the sparse matrix A using three different formats. Each format is explained in Section 4:

- **COO:** A is stored in coordinate list (COO) format, where the row and column indices of nonzeros are stored as AM and AK. The indirect Einsum is: $C_{AM_p,n} = AV_p \times B_{AK_p,n}$.
- **COO + Group:** We group the row indices AM in groups of 16. The indirect Einsum is: $C_{AM_p,n} = AV_{p,q} \times B_{AK_{p,q},n}$.
- **COO + Group + Block:** Once we block the matrix with 32×32 dense block, we then group the block rows AM (by 4). The Einsum is: $C_{AM_p,bm,n} = AV_{p,q,bm,bk} \times B_{AK_{p,q},bk,n}$.

In the top three rows of Figure 13, we compare these formats without our compiler optimizations. In this setting, the PyTorch compiler separately launches gather, matrix multiplication, and scatter operations.

Grouping alone provides a substantial speedup, 8× over the baseline. This comes partly from reduced memory consumption: GroupCOO uses only 69% of the memory required by COO. More importantly, grouping improves data reuse and reduces the number of scatter operations.

```
# COO SpMM
parallel-for p in [0, NNZ):
  for n in [0, N):
    C[AM[p], n] += AV[p] * B[AK[p], n] # NNZ * N
        atomics

# GroupCOO SpMM (group size = g)
parallel-for p in [0, NNZ / g):
  for n in [0, N):
    acc = 0 # accumulate in register
    for q in [0, g): # nonzeros in a group
      acc += AV[p, q] * B[AK[p, q], n]
    C[AM[p], n] += acc # NNZ / g * N atomics
```

COO SpMM indirectly accesses B and C for every nonzero loop p, issuing an atomic update per element of C. In contrast, GroupCOO reuses the same row of C across all g nonzeros in a group, accumulating their contributions in registers before issuing an atomic update. This improves locality for both C and B, and reduces the number of atomic calls by a factor of g, which explains much of the observed speedup.

Finally, we observe that combining grouping and blocking delivers even greater performance—up to 20× over the baseline. The blocked formats achieve this by enabling Tensor Core, while grouping improves data reuse.

Impact of Compiler Optimizations: In the bottom two rows of Figure 13, we apply our compiler optimizations on top of the COO + Group + Block format. Enabling native matrix multiplication with Tensor Core delivers a 2.6× speedup over the default PyTorch compiler by fusing gather, matmul, and scatter into a single Triton kernel. This eliminates the need to materialize large intermediate tensors (which exceed 1.5GB when fusion is disabled) and avoids costly reloads from global memory. With fusion, the gathered data remains in shared memory and is fed directly into the Tensor Core without unnecessary trips to global memory. Finally, Lazy Broadcasting further improves memory efficiency by eliminating the transposing and reshaping overhead in the Triton kernel, increasing the instruction level parallelism feeding into Tensor Cores.

6.7 Comparison Against Other Sparse Compilers

Metric	**Insum**	**TACO**[17]	**SparseTIR**[44]
Compile (s)	9.9	**0.01**	0.32
Autotune (s)	**4.9**	N/A (10 LoC)	N/A (860 LoC)
FormatConvert (ms)	0.55	**0.47**	13.47
Runtime (ms)	**0.47**	253.53	1.05

Table 4. Performance, compilation cost, and conversion cost for point cloud convolution on the conferenceRoom input with FP16 and channel size 128. TACO and SparseTIR lack autotuners, requiring users to manually search for the best schedule, which often demands substantial time and code: 10 lines of schedule for TACO and 860 lines for SparseTIR, respectively.

While our compiler generates kernels that outperform hand-written kernels, two sources of overhead must be considered before kernel launch: (1) compilation and (2) format conversion. Focusing on these aspects, we compare Insum with other sparse GPU compilers, TACO [17] and SparseTIR [44], using the point cloud convolution example.

Compilation overhead typically consists of two parts: scheduling (tuning) to select tile sizes and format to store, followed by code generation and compilation. In deep learning workloads, this cost is easily amortized since compiled operators are cached and reused many times during inference. The difficulty arises because compilers lack an auto-scheduler, forcing developers to manually craft schedules. Using TACO, we spent hours identifying a schedule that still underutilized

the GPU (no shared memory or Tensor Core usage), and SparseTIR required adopting a schedule of nearly 800 lines provided by its authors. In contrast, our compiler requires only a single indirect Einsum. We integrate the PyTorch compiler's autotuning infrastructure to automatically discover efficient Triton configurations for our fused kernel, eliminating the need for manual schedule engineering. Although our compile+autotuning time is longer (14.8 seconds), the generated code is both the fastest and the only compiler-based approach to surpass the hand-written implementation (0.66 ms, TorchSparse). This process is fully automated, and its cost is easily amortized over repeated execution.

For the majority of applications, the format conversion overhead can also be amortized [41, 44]. Many sparse workloads are static in structure, such as fixed graphs in GNNs [38] and pruned weight matrices [40], allowing conversion to be done once offline and reused throughout inference. Even in dynamic scenarios such as point cloud convolution, the format conversion is done once and reused across multiple layers [31]. Our framework shows moderate conversion cost (0.55 ms). The conversion kernel of SparseTIR is implemented on the CPU, resulting in significantly higher overhead (13.47 ms). TACO achieves the fastest conversion (0.47 ms), but its advantage is offset by prohibitively slow runtime performance. Thus, our approach strikes a balance: reasonable conversion time combined with the best kernel runtime.

7 Conclusion

This paper introduces a new approach for expressing sparse computations using indirect Einsums over fixed-length sparse formats, and presents Insum, a compiler that generates efficient GPU kernels by leveraging the PyTorch compiler. We further extend the PyTorch compiler to natively support Tensor Cores with lazy broadcasting, enabling Insum to generate fully fused gather–Einsum–scatter operations. Our evaluation demonstrates that sparse workloads can be expressed in a single indirect Einsum, while achieving better performance compared to hand-written sparse kernels.

Acknowledgments

This work was supported in part by the National Science Foundation under grants CCF-2217064, CCF-2107244, and CCF-2217099; by the Defense Sciences Office of DARPA under the PROWESS program (HR0011-23-C-0101) and SBIR award HR001123C0139; a startup grant from the Georgia Institute of Technology; and by the U.S. Department of Energy under the PSAAP program (DE-NA0003965). We thank the anonymous reviewers for their thoughtful feedback.

References

[1] Willow Ahrens, Teodoro Fields Collin, Radha Patel, Kyle Deeds, Changwan Hong, and Saman Amarasinghe. 2025. Finch: Sparse and Structured Tensor Programming with Control Flow. *Proc. ACM Program. Lang.* 9, OOPSLA1, Article 117 (April 2025), 31 pages. doi:10.1145/3720473

[2] Willow Ahrens, Daniel Donenfeld, Fredrik Kjolstad, and Saman Amarasinghe. 2023. Looplets: A Language for Structured Coiteration. In *Proceedings of the 21st ACM/IEEE International Symposium on Code Generation and Optimization*. 41–54. doi:10.1145/3579990.3580020

[3] Jason Ansel, Edward Yang, Horace He, Natalia Gimelshein, Animesh Jain, Michael Voznesensky, Bin Bao, Peter Bell, David Berard, Evgeni Burovski, et al. 2024. Pytorch 2: Faster machine learning through dynamic python bytecode transformation and graph compilation. In *Proceedings of the 29th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2*. 929–947. doi:10.1145/3620665.3640366

[4] Aart Bik, Penporn Koanantakool, Tatiana Shpeisman, Nicolas Vasilache, Bixia Zheng, and Fredrik Kjolstad. 2022. Compiler Support for Sparse Tensor Computations in MLIR. *ACM Transactions on Architecture and Code Optimization* 19, 4 (2022), 1–25. doi:10.1145/3554733

[5] James Bradbury, Roy Frostig, Peter Hawkins, Matthew James Johnson, Chris Leary, Dougal Maclaurin, George Necula, Adam Paszke, Jake VanderPlas, Skye Wanderman-Milne, and Qiao Zhang. 2018. *JAX: composable transformations of Python+NumPy programs*. http://github.com/jax-ml/jax

[6] Aydin Buluc and John R. Gilbert. 2008. On the representation and multiplication of hypersparse matrices. In *2008 IEEE International Symposium on Parallel and Distributed Processing*. 1–11. doi:10.1109/IPDPS.2008.4536313

[7] Tianqi Chen, Thierry Moreau, Ziheng Jiang, Lianmin Zheng, Eddie Yan, Haichen Shen, Meghan Cowan, Leyuan Wang, Yuwei Hu, Luis Ceze, et al. 2018. TVM: An automated End-to-End optimizing compiler for deep learning. In *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)*. 578–594. https://www.usenix.org/conference/osdi18/presentation/chen

[8] Stephen Chou, Fredrik Kjolstad, and Saman Amarasinghe. 2018. Format abstraction for sparse tensor algebra compilers. *Proceedings of the ACM on Programming Languages* 2, OOPSLA (2018), 1–30. doi:10.1145/3276493

[9] Albert Einstein. 1916. The foundation of the general theory of relativity. *Annalen Phys.* 49, 7 (1916), 769–822. doi:10.1002/andp.19163540702

[10] Matthias Fey and Jan Eric Lenssen. 2019. Fast graph representation learning with PyTorch Geometric. *arXiv preprint arXiv:1903.02428* (2019).

[11] Trevor Gale, Matei Zaharia, Cliff Young, and Erich Elsen. 2020. Sparse GPU Kernels for Deep Learning. In *SC20: International Conference for High Performance Computing, Networking, Storage and Analysis*. 1–14. doi:10.1109/SC41405.2020.00021

[12] Mario Geiger and Tess Smidt. 2022. e3nn: Euclidean neural networks. *arXiv preprint arXiv:2207.09453* (2022).

[13] Charles R. Harris, K. Jarrod Millman, Stéfan J. van der Walt, Ralf Gommers, Pauli Virtanen, David Cournapeau, Eric Wieser, Julian Taylor, Sebastian Berg, Nathaniel J. Smith, Robert Kern, Matti Picus, Stephan Hoyer, Marten H. van Kerkwijk, Matthew Brett, Allan Haldane, Jaime Fernández del Río, Mark Wiebe, Pearu Peterson, Pierre Gérard-Marchant, Kevin Sheppard, Tyler Reddy, Warren Weckesser, Hameer Abbasi, Christoph Gohlke, and Travis E. Oliphant. 2020. Array programming with NumPy. *Nature* 585 (Sept. 2020), 357–362. doi:10.1038/s41586-020-2649-2

[14] Kartik Hegde, Hadi Asghari-Moghaddam, Michael Pellauer, Neal Crago, Aamer Jaleel, Edgar Solomonik, Joel Emer, and Christopher W Fletcher. 2019. Extensor: An accelerator for sparse tensor algebra. In *Proceedings of the 52nd annual IEEE/ACM international symposium on microarchitecture*. 319–333. doi:10.1145/3352460.3358275

[15] Ke Hong, Zhongming Yu, Guohao Dai, Xinhao Yang, Yaoxiu Lian, Ningyi Xu, and Yu Wang. 2023. Exploiting hardware utilization and adaptive dataflow for efficient sparse convolution in 3D point clouds. *Proceedings of Machine Learning and Systems* 5 (2023), 428–441.

[16] Andrew James. 2023. PyTorch 2.1: Quansight's Improvements to BSR Sparse Matrix Multiplication. https://quansight.com/post/pytorch-2-1-quansights-improvements-to-bsr-sparse-matrix-multiplication/.

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

[18] Chris Lattner, Mehdi Amini, Uday Bondhugula, Albert Cohen, Andy Davis, Jacques Pienaar, River Riddle, Tatiana Shpeisman, Nicolas Vasilache, and Oleksandr Zinenko. 2021. MLIR: Scaling compiler infrastructure for domain specific computation. In *2021 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*. IEEE, 2–14. doi:10.1109/CGO51591.2021.9370308

[19] Nandeeka Nayak, Toluwanimi O Odemuyiwa, Shubham Ugare, Christopher Fletcher, Michael Pellauer, and Joel Emer. 2023. TeAAL: A declarative framework for modeling sparse tensor accelerators. In *Proceedings of the 56th annual IEEE/ACM international symposium on microarchitecture*. 1255–1270. doi:10.1145/3613424.3623791

[20] Nandeeka Nayak, Xinrui Wu, Toluwanimi O. Odemuyiwa, Michael Pellauer, Joel S. Emer, and Christopher W. Fletcher. 2024. FuseMax: Leveraging Extended Einsums to Optimize Attention Accelerator Design. In *2024 57th IEEE/ACM International Symposium on Microarchitecture (MICRO)*. 1458–1473. doi:10.1109/MICRO61859.2024.00107

[21] NVIDIA. 2025. cuEquivariance: High-Performance Equivariant Neural Network Library. https://github.com/NVIDIA/cuEquivariance.

[22] NVIDIA. 2025. cuSPARSE: A CUDA Library for Sparse Matrix Computations. https://docs.nvidia.com/cuda/cusparse/.

[23] Toluwanimi O. Odemuyiwa, Joel S. Emer, and John D. Owens. 2024. The EDGE Language: Extended General Einsums for Graph Algorithms. http://arxiv.org/abs/2404.11591

[24] Angshuman Parashar, Priyanka Raina, Yakun Sophia Shao, Yu-Hsin Chen, Victor A. Ying, Anurag Mukkara, Rangharajan Venkatesan, Brucek Khailany, Stephen W. Keckler, and Joel Emer. 2019. Timeloop: A Systematic Approach to DNN Accelerator Evaluation. In *2019 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*. 304–315. doi:10.1109/ISPASS.2019.00042

[25] 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 Yang, Zach 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*. Curran Associates Inc., Red Hook, NY, USA.

[26] 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 (PLDI '13)*. Association for Computing Machinery, New York, NY, USA, 519–530. doi:10.1145/2491956.2462176

[27] James Reed, Zachary DeVito, Horace He, Ansley Ussery, and Jason Ansel. 2022. torch. fx: Practical program capture and transformation for deep learning in python. *Proceedings of Machine Learning and Systems* 4 (2022), 638–651.

[28] John R Rice and Ronald F Boisvert. 2012. *Solving elliptic problems using ELLPACK*. Vol. 2. Springer Science & Business Media.

[29] Olaf Schenk, Klaus Gärtner, Wolfgang Fichtner, and Andreas Stricker. 2001. PARDISO: a high-performance serial and parallel sparse linear solver in semiconductor device simulation. *Future Generation Computer Systems* 18, 1 (2001), 69–78. doi:10.1016/S0167-739X(00)00076-5

[30] Vivienne Sze, Yu-Hsin Chen, Tien-Ju Yang, and Joel S Emer. 2020. Efficient processing of deep neural networks. *Synthesis Lectures on Computer Architecture* 15, 2 (2020), 1–341. doi:10.1007/978-3-031-01766-7

[31] Haotian Tang, Zhijian Liu, Xiuyu Li, Yujun Lin, and Song Han. 2022. Torchsparse: Efficient point cloud inference engine. *Proceedings of Machine Learning and Systems* 4 (2022), 302–315.

[32] Haotian Tang, Zhijian Liu, Shengyu Zhao, Yujun Lin, Ji Lin, Hanrui Wang, and Song Han. 2020. Searching Efficient 3D Architectures with Sparse Point-Voxel Convolution. In *European Conference on Computer Vision (ECCV)*. doi:10.1007/978-3-030-58604-1_41

[33] Haotian Tang, Shang Yang, Zhijian Liu, Ke Hong, Zhongming Yu, Xiuyu Li, Guohao Dai, Yu Wang, and Song Han. 2023. Torchsparse++: Efficient point cloud engine. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*. 202–209.

[34] Haotian Tang, Shang Yang, Zhijian Liu, Ke Hong, Zhongming Yu, Xiuyu Li, Guohao Dai, Yu Wang, and Song Han. 2023. Torchsparse++: Efficient training and inference framework for sparse convolution on gpus. In *Proceedings of the 56th Annual IEEE/ACM International Symposium on Microarchitecture*. 225–239. doi:10.1145/3613424.3614303

[35] 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. doi:10.1109/LLVMHPC54804.2021.00009

[36] Philippe Tillet, H. T. Kung, and David Cox. 2019. Triton: an intermediate language and compiler for tiled neural network computations. In *Proceedings of the 3rd ACM SIGPLAN International Workshop on Machine Learning and Programming Languages* (Phoenix, AZ, USA) *(MAPL 2019)*. Association for Computing Machinery, New York, NY, USA, 10–19. doi:10.1145/3315508.3329973

[37] Minjie Yu Wang. 2019. Deep graph library: Towards efficient and scalable deep learning on graphs. In *ICLR workshop on representation learning on graphs and manifolds*.

[38] Yuke Wang, Boyuan Feng, Zheng Wang, Guyue Huang, and Yufei Ding. 2023. TC-GNN: Bridging sparse GNN computation and dense tensor cores on GPUs. In *2023 USENIX Annual Technical Conference (USENIX ATC 23)*. 149–164.

[39] Jaeyeon Won, Willow Ahrens, Teodoro Fields Collin, Joel S. Emer, and Saman Amarasinghe. 2025. The Continuous Tensor Abstraction: Where Indices Are Real. *Proc. ACM Program. Lang.* 9, OOPSLA2, Article 368 (Oct. 2025), 29 pages. doi:10.1145/3763146

[40] Jaeyeon Won, Changwan Hong, Charith Mendis, Joel Emer, and Saman Amarasinghe. 2023. Unified Convolution Framework: A compiler-based approach to support sparse convolutions. *Proceedings of Machine Learning and Systems* 5 (2023), 666–679.

[41] Jaeyeon Won, Charith Mendis, Joel S Emer, and Saman Amarasinghe. 2023. WACO: learning workload-aware co-optimization of the format and schedule of a sparse tensor program. In *Proceedings of the 28th ACM international conference on architectural support for programming languages and operating systems, volume 2*. 920–934. doi:10.1145/3575693.3575742

[42] Yannan N. Wu, Po-An Tsai, Saurav Muralidharan, Angshuman Parashar, Vivienne Sze, and Joel S. Emer. 2023. HighLight: Efficient and Flexible DNN Acceleration with Hierarchical Structured Sparsity. In *ACM/IEEE International Symposium on Microarchitecture (MICRO)*. doi:10.1145/3613424.3623786

[43] Yannan Nellie Wu, Po-An Tsai, Angshuman Parashar, Vivienne Sze, and Joel S Emer. 2022. Sparseloop: An analytical approach to sparse tensor accelerator modeling. In *2022 55th IEEE/ACM International Symposium on Microarchitecture (MICRO)*. IEEE, 1377–1395. doi:10.1109/MICRO56248.2022.00096

[44] 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*. 660–678. doi:10.1145/3582016.3582047