FuseFlow: A Fusion-Centric Compilation Framework for Sparse Deep Learning on Streaming Dataflow

Rubens Lacouture
Stanford University
Stanford, USA
rubensl@stanford.edu

Nathan Zhang
Stanford University
Stanford, USA
stanfurd@stanford.edu

Ritvik Sharma
Stanford University
Stanford, USA
rsharma3@stanford.edu

Marco Siracusa
Barcelona Supercomputing Center
Barcelona, Spain
marco.siracusa@bsc.es

Fredrik Kjolstad
Stanford University
Stanford, USA
kjolstad@stanford.edu

Kunle Olukotun
Stanford University
Stanford, USA
kunle@stanford.edu

Olivia Hsu
Stanford University
Stanford, USA
Carnegie Mellon University
Pittsburgh, USA
owhsu@stanford.edu

Abstract

As deep learning models scale, sparse computation and specialized dataflow hardware have emerged as powerful solutions to address efficiency. We propose FuseFlow, a compiler that converts sparse machine learning models written in PyTorch to fused sparse dataflow graphs for reconfigurable dataflow architectures (RDAs). FuseFlow is the first compiler to support general cross-expression fusion of sparse operations. In addition to fusion across kernels (expressions), FuseFlow also supports optimizations like parallelization, dataflow ordering, and sparsity blocking. It targets a cycle-accurate dataflow simulator for microarchitectural analysis of fusion strategies. We use FuseFlow for design-space exploration across four real-world machine learning applications with sparsity, showing that full fusion (entire cross-expression fusion across all computation in an end-to-end model) is not always optimal for sparse models—fusion granularity depends on the model itself. FuseFlow also provides a heuristic to identify and prune suboptimal configurations. Using Fuseflow, we achieve performance improvements, including a ~2.7x speedup over an unfused baseline for GPT-3 with BigBird block-sparse attention.

1 Introduction

Sparse computation is a promising avenue for greater model efficiency in machine learning (ML) systems [17, 18, 23, 26, 58], often at the cost of lower hardware utilization. To increase hardware efficiency, researchers and companies are building specialized hardware to accelerate sparse computations [7, 11, 16, 24, 35, 42, 44, 46, 47, 50, 54]. In order to increase efficiency in these hardware architectures, they are also making increasing use of dataflow, or direct connections between coarse-grained functional units, to rely less on expensive caches, local memories, and memory operations. Dataflow architectures are particularly well-suited for sparse computation because they explicitly coordinate data movement through streaming connections rather than relying on caches, naturally handling the irregular memory access patterns inherent in sparse data [28, 35, 50]. Empirically, GPUs are underutilized: a 3-layer GCN inference in *PyG* on an RTX 5090 across five real-world graphs shows consistently low compute (SM) utilization (avg 16.7%) and ~1% memory utilization, with *mag* as the only case approaching moderate SM utilization (Fig. 1). These observations motivate specialized sparse dataflow accelerators and the compiler support to program them.

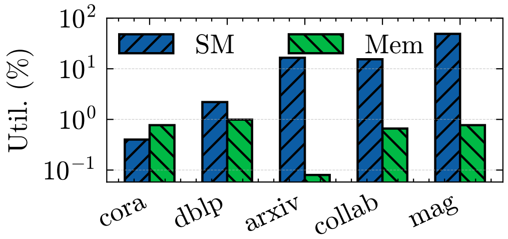


Figure 1. Log plot of SM and DRAM utilization (%) for *PyG* GCN inference on an RTX 5090 across five datasets.

Hsu et al. [28] introduced the Sparse Abstract Machine (SAM) to increase the programmability of these emerging sparse dataflow hardware architectures. The Sparse Abstract Machine is a dataflow abstract machine for sparse tensor algebra computations. It adopts a streaming dataflow model, where data flows between compute nodes. It can express any tensor algebra expression by composing a handful of simple and intuitive dataflow blocks. It also naturally supports expressing fused computation across multiple expressions, different ways to order the dataflow (the iteration order) within an expression (e.g., Gustavson's algorithm [21] versus inner product for sparse matrix multiplication), and lends itself to compile to fabricated hardware [35]. SAM is therefore a natural starting point as a compiler intermediate representation (IR) for targeting sparse ML models to dataflow hardware.

Hsu et al. [28] also describe a compilation flow from high-level sparse tensor algebra expressions to SAM dataflow graphs. Their Custard compiler generates SAM graphs that fuse operations across a sparse tensor algebra expression and lets users control the dataflow order. The Custard compilation algorithm is a significant step forward, as the first to demonstrate compilation of sparse tensor algebra expressions to dataflow. It is not, however, suitable for ML model compilation due to its limited capabilities for fusion.

Because Custard compiles individual expressions, it is unable to fuse operations across expressions—a key feature in ML compilers. Moreover, the intra-expression fusion that falls out of Custard's compilation algorithm fully fuses each expression without any support for partial fusion. Partial fusion is often desirable to provide some of the benefits of fusion while controlling the amount of reuse within a computation (as demonstrated by FlashAttention [12]). The tradeoff between fusion and reuse is even more critical, and looks different from that of dense computation, since fused sparse computation may have better asymptotic complexity [2, 4, 28, 33]. Therefore, an ML compiler should expose the fusion granularity as a user schedule so the fusion and reuse tradeoff can be explored across models.

In this paper, we describe a new approach for lowering general sparse ML workloads (beyond pure sparse tensor algebra) to SAM dataflow graphs. Sparse ML refers to deep learning models with one or more sparse tensors. Our approach supports both cross-expression fusion and partial fusion, allowing users to explore the trade-off between fusion and reuse. Our work consists of two new IRs that enable this fusion exploration. The first IR, a fused Einstein Summation (Einsum) representation, tracks the flow of indexing (coordinate) data and values across fused expressions. Then, we introduce a new fusion table representation, a lowering IR that names and memoizes intermediate streams allowing the compiler to reference subgraphs before their materialization to efficiently emit the fused dataflow graph.

We also develop an end-to-end ML compilation system, called FuseFlow, that implements our fusion approach. FuseFlow compiles PyTorch [3] with sparse annotations [20, 63] to SAM graphs. In FuseFlow, users leverage a scheduling language that lets them control fusion granularity and dataflow ordering of expressions. To support modern ML models, we also add support for dense blocks to support block-sparse tensors, non-linear functions, and masking operations to SAM. Finally, our FuseFlow system can generate dataflow graphs that execute in a data-parallel fashion in addition to the pipeline parallelism native to dataflow graphs. Our technical contributions are thus:

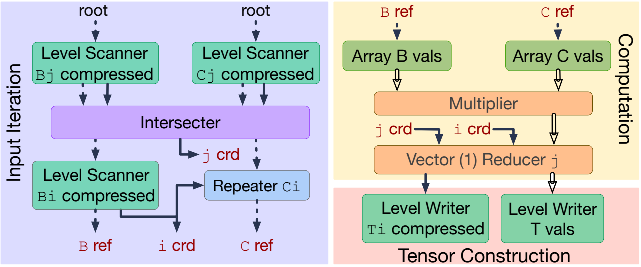


Figure 2. SAM graph for sparse-matrix vector multiplication with $j \rightarrow i$ dataflow. Streams: solid grey = coordinate (crd), dashed grey = reference (ref), double black = value (val).

- An algorithm for fusion across multiple independent Einsum expressions (Section 5),
- A lowering algorithm that converts the fused Einsum expressions to a dataflow representation (Section 6),
- An end-to-end compiler framework for sparse dataflow machines (Section 7). The implementation includes optimizations necessary for performant application code like parallelization, block sparsity, dataflow ordering, and a fusion heuristic.

We demonstrate the effectiveness of FuseFlow across four model classes by generating 56 equivalent dataflow configurations that yield speedups from ~1.01x to ~3.9x. Our evaluation underscores the importance of selecting the appropriate fusion granularity and shows that FuseFlow's heuristic successfully prunes inefficient configurations, offering critical insights for the deployment of large-scale sparse ML applications on dataflow architectures.

2 Sparse Abstract Machine Background

We provide necessary background on the Sparse Abstract Machine (SAM) [28] to understand our FuseFlow system and the code that it generates. SAM expresses tensor algebra kernels as dataflow graphs by providing a streaming-tensor abstraction and primitives that compose to perform tensor algebra operations. Tensor algebra kernels can be expressed in Einsum notation where tensors are indexed by variables, with addition and multiplication as the core operations. The index variables specify how levels across tensors are broadcast, reduced, and contracted. SAM also introduces the Custard compiler, which compiles high-level Einsum into SAM dataflow graphs. These dataflow graphs are suitable for VLSI implementations and simulation but remain abstract in order to cleanly decouple programs from accelerator implementations.

Figure 2 shows the SAM graph for sparse matrix-vector multiplication (SpMV) $T_i = B_{ij}C_j$ with the $j \rightarrow i$ dataflow ($\forall_{ji}\ T_i \mathrel{+}= B_{ij}C_j$) where B is stored in a compressed format (e.g., compressed sparse row (CSR)) [10, 20] SAM expresses tensors as streams of data with control tokens, where these tensor streams flow on the arrows between primitives (boxes) in a SAM dataflow graph. SAM's primitives include:

Level scanners (LS) traverse tensor levels. Nested LS produce streams that are logically equivalent to multidimensional tensors (e.g., B_j with B_i fetch matrix B's coordinates).

Stream joiners (Intersect/Union) combine or skip coordinates across tensors (e.g., the j intersect joins B_j and C_j.

Repeaters (Rep) broadcast operands (e.g., C across each i)

ALU and reducers (Red) elementwise ops and reductions over named indices (e.g., reduce over j in $j \rightarrow i$).

Level writers (LW) and coordinate droppers (CD) write results and elide empty coordinates.

As in Figure 2, SAM primitives compose together with streams to form SAM graphs that represent any tensor algebra expression with varying dataflows. Arrows in Figure 2 connect dataflow primitives together and transmit streams, where each stream is a sequence of tokens that transmits one level of a tensor in fibertree form (a nested representation of per-level coordinates and values) [61]. Streams are of three types: coordinates (`crd`), references to inner levels (`ref`), and values (`val`). An n-order tensor is represented by n coordinate streams plus one values stream (n+1 total).

SAM graphs comprise three regions (see shading in Figure 2): input iteration, computation, and tensor construction. The input iteration region (shown in blue) iterates through the tensor coordinates of all input operands, joining the sparse coordinates together (e.g. through intersecter$_j$). The computation region (shown in yellow) fetches data values using coordinates and computes the result values. And, the tensor construction region (shown in red) writes the result values and coordinates back to memory, dropping any zero coordinates. Our work builds upon SAM and addresses compiler limitations by targeting fused applications beyond single sparse tensor kernels.

3 Forms of Fusion

Fusion in dense and sparse compilers can be categorized by scope and by technique. We describe three main types of fusion and provide a diagram of them in Figure 3.

Pattern-based operator fusion (POF) refers to merging operations based on recognized patterns, where sequences of operators are replaced by one kernel that fuses those operations. Often, the fused kernel is handcrafted. This operator fusion approach is common in dense compilers on CPUs, GPUs, and TPUs. Because POF is often completely automatic (all detected operator patterns are always replaced with the fused version), it is typically limited to localized patterns that are known in advance. Due to its localized nature, operator fusion is an intra-layer technique.

Intra-expression iteration fusion (IIF) merges the iteration space of a single tensor algebra expression—co-iterating its inputs—without crossing kernel boundaries. IIF manifests as loop fusion for dense computation, co-iteration for sparse computation, and dataflow iteration fusion in dataflow graphs. Existing sparse tensor compilers that target dataflow hardware [27, 28] employ this type of fusion, co-iterating to generate fused iteration spaces that elide zeros for one sparse expression at a time.

Inter-expression kernel fusion (EKF) fuses across different kernels or sub-computations into a single fused computation graph. This type of fusion can be implemented in conjunction with POF and IIF, but is not supported by existing sparse dataflow compilers [27, 28].

Of the forms of fusion above, Table 1 summarizes how prior frameworks fit into these categories. Existing frameworks thus leave a gap: no prior compiler automatically fuses across multiple sparse expressions in a general way. POF in dense compilers is limited to known templates and IIF in dense compilers is straightforward. IIF in sparse compilers stop at single-kernel fusion. However, EKF is necessary to enable fusion across multiple sparse expressions in a model. FuseFlow addresses this gap by providing a general algorithm for fusing entire sparse ML pipelines across kernel boundaries. Therefore, as highlighted by Table 1, FuseFlow is the first sparse compiler to provide an algorithmic approach focusing on inter-expression kernel fusion.

As FuseFlow is a sparse dataflow compiler, we contrast its fusion capabilities with the capabilities of the two prior sparse dataflow compilers Custard [28] and Stardust [27] (C+S) in Figure 4.[1] Custard and Stardust only support fusion within an expression and not across expressions. Although a user can combine expressions into a larger expression, which can then be fused, they must do so by hand and cannot fuse computations that have more than one result. The various fused regions (blue boxes) compare C+S fusion regions with our FuseFlow, which fuses all kernels within a fusion region.

FuseFlow's comprehensive fusion support leads to better performance. In GCN on the OGB-Collab dataset [29] (Figure 4b), fusion with Custard and Stardust using a handwritten rewrite yields 1.97× speedup over the unfused baseline. With less user effort, FuseFlow achieves another 1.33× speedup over C+S, leading to a ~2.63× speedup in total. FuseFlow's additional speedups come from its support for IIF during code generation. Along with EKF, efficient fused sparse computation also hinges on the design of IIF, which we discuss next.

[1] The motivation and description in Section 1 is also true for the Stardust compiler [27], another compiler from the same high-level sparse tensor algebra languages to a real dataflow accelerator [50].

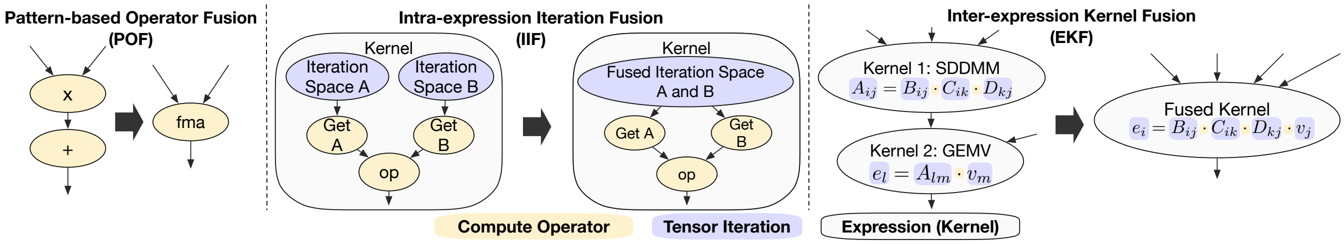
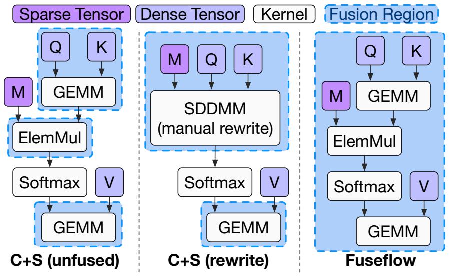

Pattern-based Operator Fusion (POF)
x
+
fma
Intra-expression Iteration Fusion (IIF)
Kernel
Iteration Space A
Iteration Space B
Get A
Get B
op
Kernel
Fused Iteration Space A and B
Get A
Get B
op
Compute Operator
Tensor Iteration
Inter-expression Kernel Fusion (EKF)
Kernel 1: SDDMM
$A_{ij} = B_{ij} \cdot C_{ik} \cdot D_{kj}$
Kernel 2: GEMV
$e_l = A_{lm} \cdot v_m$
Fused Kernel
$e_i = B_{ij} \cdot C_{ik} \cdot D_{kj} \cdot v_j$
Expression (Kernel)

Figure 3. Dataflow diagrams for the forms of fusion, showing how they differ and are related.

Tensor Compiler	Multi-expression	Sparsity	Fusion Strategy	Backends
TensorRT [45]	✓	✗	POF	GPU
XLA [51]	✓	✗	POF, IIF	CPU, GPU
DNNFusion [43]	✓	✗	POF, IIF	CPU, GPU
TVM [9]	✓	✗	POF, IIF	CPU, GPU, TPU
TACO [32]	✗	✓	IIF	CPU, GPU
SparseTIR [65]	✗	✓	IIF	CPU, GPU
ReACT [70]	✗	✓	IIF	CPU, GPU
Stardust [27]	✗	✓	IIF	Dataflow
Custard [28]	✗	✓	IIF	Dataflow
This Work	✓	✓	EKF, IIF	Dataflow

Table 1. Landscape of tensor compilers. **EKF (Inter-Expression Kernel Fusion)** enables fusion across multiple sparse tensor expressions. Prior sparse compilers only support **IIF (Intra-Expression Iteration Fusion)** within single kernels and dense compilers primarily rely on limited **POF (Pattern-based Operator Fusion)** via pattern matching.



(a) Compiler fusion comparison with sparse attention. Custard+Stardust (C+S) support IIF and only support EKF manually where unsupported ops break EKF.

(b) Normalized GCN performance with compilers from (a).

Config	Speed-up
C+S (unfused)	1.00×
C+S (rewrite)	1.97×
FuseFlow	**2.63×**

Figure 4. Comparing fusion coverage and performance.

```
for(int i = 0; i < I; i++)
  for(int k = 0; k < K; k++)
    for(int j = 0; j < J; j++)
      for(int l = 0; l < L; l++)
        D[i,l] += A[i,k] * B[k,j] * C[j,l]
```

(a) Global iteration space with loops

```
for(int i = 0; i < I; i++)
  for(int k = 0; k < K; k++)
    for(int j = 0; j < J; j++)
      E[i,j] += A[i,k] * B[k,j]
  for(int j = 0; j < J; j++)
    for(int l = 0; l < L; l++)
      D[i,l] += E[i,j] * C[j,l]
```

(b) Factored iteration space with loops

Figure 5. Two iteration patterns for $\forall_{ikjl} C_{il} \mathrel{+}= A_{ik} B_{kj} C_{jl}$, that are represented via loop nests with higher-order reduction variables highlighted in blue [31]. FuseFlow lowers to a dataflow input iteration graph with a factored iteration space (b), whereas prior work produces dataflow graphs with fully fused iteration spaces (a).

The Iteration Space Problem. In both inter- and intra-expression fusion approaches for sparse computation, there are multiple equivalent ways to iterate through sparse tensors, each with fundamental tradeoffs. Two primary costs, coordinate processing and computation, define the tradeoff between globally fused and factored iteration spaces.

A globally fused iteration space, shown in Figure 5a, iterates over every index variable, creating an n-dimensional iteration space, where n is the number of index variables (e.g., 4-dimensional in Figure 5a). It efficiently filters unnecessary numerical computations but incurs significant coordinate processing overhead, causing coordinate explosion as expressions grow. Prior work on sparse tensor algebra compilation to dataflow accelerators by default generates code that traverses a global iteration space [27, 28]. In contrast, factored iteration iterates pairwise over input tensors (see Figure 5b). It generates multiple smaller sub-spaces, one per binary operation (i.e. two 3-dimensional iteration spaces in Figure 5b). Each sub-space independently handles coordinate processing, significantly reducing overhead by limiting coordinate analysis to binary operations. However, as this analysis is local rather than global, factored iteration may miss opportunities to skip unnecessary computations, potentially increasing operations.

Global iteration spaces often perform poorly for sparse ML applications for two key reasons. First, these applications typically contain numerous higher-order tensors and indices, leading to a dimensionality explosion. Second, the mixture of sparse and dense tensors increases iteration points within each dimension. This combination makes traversing sparse ML models with global iteration significantly less efficient than factored iteration approaches. Hence, we opt for factored input-iteration in FuseFlow, by design, when pushing fusion through our lowering algorithm.

Avoiding global iteration space materialization requires a complete restructuring of the dataflow graph and its compiler lowering. Factored iteration preserves the order of reduction operations, unlike global iteration. Figure 5 highlights these behaviors. Recall that SAM graphs comprise three sequential regions—input iteration, computation, and tensor construction (see Section 2). Global iteration computations occur at the innermost loop (line 5 in Figure 5a), while factored iteration interleaves loops and computations (lines 4 and 7 in Figure 5b). From a dataflow perspective, rather than loop transformations, the graph input iteration and computation pipelines need to be interleaved, which we show later in Figure 10. Specifically, higher-order reducers produce coordinate streams that must interact with the input iterations of subsequent operations to enable fusion. Therefore, we need a new compiler approach, like FuseFlow's, to handle efficient sparse ML workloads on dataflow hardware.

4 Overview of FuseFlow

Figure 6 summarizes FuseFlow's compilation flow from PyTorch to a fused, hardware-ready sparse dataflow graph. Models are first lowered from PyTorch [3] to MLIR (Linalg + SparseTensor dialects) using either Torch-MLIR [38] for dense model components or MPACT [20] for sparse model components, with user-specified sparse formats and optional schedules. This process yields a graph of Einsum expressions extended with non-algebraic operators as shown by Figure 6a to Figure 6b. This stage in FuseFlow preserves sparsity semantics from the frontend and provides the knobs (schedules) that guide downstream optimization.

FuseFlow then applies cross-expression kernel fusion to user-marked fusion regions (denoted by `Fuse{}` in Figure 6b), producing a fused Einsum representation (Section 5). The fused Einsum representation includes fused components that form connected subgraph (as shown in the dashed fused region in Figure 6c) along with a partial order graph that encodes index constraints. Within the connected subgraph, the arrows indicate direct producer-consumer expression fusion. To generate this representation, FuseFlow inlines producer results into their consumers across kernel boundaries while building the partial order graph (Figure 6c). Partial order graph constraints are derived from the user-schedules (red) and sparse storage formats (green) (from Figure 6b).

To lower the fused expressions, FuseFlow introduces a new IR called *fusion tables* (Figure 6d). This representation addresses the complexity of lowering multiple fused expressions to fused dataflow graphs. The fusion table encodes: the fused iteration order based on the partial order graph in its rows (shown in Figure 6d in light pink), the fused expression which is implicit in the tensor order of its columns (shown in off-white), and IR nodes along with named pointers to intermediate streams before they are materialized in their cells (shown in cool gray). Therefore, fusion tables represent the fused tensor computation in a tabular format before code generation, allowing for factored iteration with interleaved input iteration and computation. Because prior work [27, 28] compiles single kernels, they do not scale to face similar challenges (i.e. requiring references to temporary streams) that FuseFlow address with its fusion table IR.

FuseFlow generates SAM dataflow graphs with ML primitives, which we call SAMML. FuseFlow applies user-guided or autotuned parallelization, sparsity blocking, and dataflow-order selection, and provides a fast heuristic to estimate FLOPs/bytes for early pruning. It then executes in Comal, a cycle-accurate simulator within the open-source DAM simulation framework [69], or maps to FPGA backends.

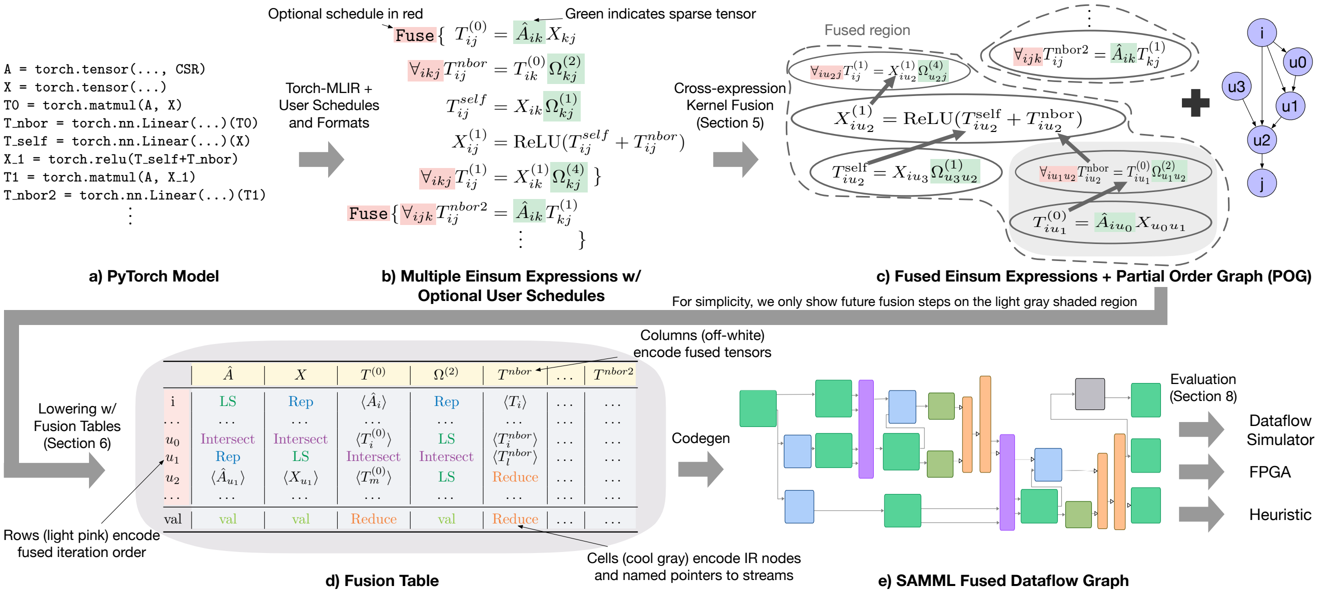
Scheduling Language. FuseFlow exposes fusion granularity, dataflow ordering, parallelization, and a performance heuristic through explicit user schedules, enabling the design-space exploration in Section 8. Users specify fusion via *Fuse{}* regions (i.e. denoted in MLIR with functions), dataflow order via modifying MLIR Linalg affine maps, and other parameters via a command line interface. While this control is essential for optimal performance, future work includes autoscheduling to determine a fusion plan and schedules for common sparse ML patterns.

5 Cross-Expression Fusion Algorithm

Our solution for automatically generating fused code relies on FuseFlow's cross-expression fusion algorithm. It fuses across distinct expressions while preserving correctness and efficiency in sparse iteration. Once fusion regions are scheduled, FuseFlow's algorithm produces a collection of fused Einsum expressions and a partial order graph for each fusion region that serves as the foundation for further optimization.

Sparsity-specific challenges arise when multiple expressions each impose their own traversal order over many sparse tensors. Efficient sparse computation requires *concordant traversal*, a fused iteration order compatible with each operand's native mode order (storage order) [68]. Traversing a sparse tensor against its storage format (e.g., column-wise over a CSR matrix) is *discordant* and is often asymptotically worse. Ignoring this critical aspect of sparse computation can lead to incorrect code [32] or suboptimal performance due to expensive indirect lookups and tensor reformatting [32, 68].

Our approach treats ordering constraints as first-class: (i) user-specified dataflow order constraints of each local



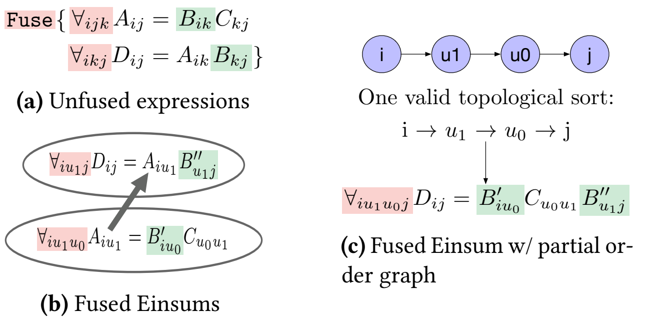
```
A = torch.tensor(..., CSR)
X = torch.tensor(...)
T0 = torch.matmul(A, X)
T_nbor = torch.nn.Linear(...)(T0)
T_self = torch.nn.Linear(...)(X)
X_1 = torch.relu(T_self+T_nbor)
T1 = torch.matmul(A, X_1)
T_nbor2 = torch.nn.Linear(...)(T1)
                ⋮
```

$$
\begin{aligned}
\texttt{Fuse}\{\ T^{(0)}_{ij} &= \hat{A}_{ik}X_{kj} \\
\forall_{ikj}T^{nbor}_{ij} &= T^{(0)}_{ik}\Omega^{(2)}_{kj} \\
T^{self}_{ij} &= X_{ik}\Omega^{(1)}_{kj} \\
X^{(1)}_{ij} &= \mathrm{ReLU}(T^{self}_{ij} + T^{nbor}_{ij}) \\
\forall_{ikj}T^{(1)}_{ij} &= X^{(1)}_{ik}\Omega^{(4)}_{kj}\ \} \\
\texttt{Fuse}\{\forall_{ijk}T^{nbor2}_{ij} &= \hat{A}_{ik}T^{(1)}_{kj} \\
&\ \vdots\qquad \}
\end{aligned}
$$

	$\hat{A}$	X	$T^{(0)}$	$\Omega^{(2)}$	T^{nbor}	...	T^{nbor2}
i	LS	Rep	$\langle \hat{A}_i \rangle$	Rep	$\langle T_i \rangle$	...	...
...	...	...	...	...	...	...	...
u_0	Intersect	Intersect	$\langle T^{(0)}_i \rangle$	LS	$\langle T^{nbor}_i \rangle$	...	...
u_1	Rep	LS	Intersect	Intersect	$\langle T^{nbor}_l \rangle$	...	...
u_2	$\langle \hat{A}_{u_1} \rangle$	$\langle X_{u_1} \rangle$	$\langle T^{(0)}_m \rangle$	LS	Reduce	...	...
...	...	...	...	...	...	...	...
val	val	val	Reduce	val	Reduce	...	...

Optional schedule in red
Green indicates sparse tensor
Torch-MLIR + User Schedules and Formats
Cross-expression Kernel Fusion (Section 5)
Fused region
$\forall_{iu_2j}T^{(1)}_{ij} = X^{(1)}_{iu_2}\Omega^{(4)}_{u_2j}$
$\forall_{ijk}T^{\text{nbor2}}_{ij} = \hat{A}_{ik}T^{(1)}_{kj}$
$X^{(1)}_{iu_2} = \mathrm{ReLU}(T^{\text{self}}_{iu_2} + T^{\text{nbor}}_{iu_2})$
$T^{\text{self}}_{iu_2} = X_{iu_3}\Omega^{(1)}_{u_3u_2}$
$\forall_{iu_1u_2}T^{\text{nbor}}_{iu_2} = T^{(0)}_{iu_1}\Omega^{(2)}_{u_1u_2}$
$T^{(0)}_{iu_1} = \hat{A}_{iu_0}X_{u_0u_1}$
i
u0
u3
u1
u2
j
a) PyTorch Model
b) Multiple Einsum Expressions w/ Optional User Schedules
c) Fused Einsum Expressions + Partial Order Graph (POG)
For simplicity, we only show future fusion steps on the light gray shaded region
Lowering w/ Fusion Tables (Section 6)
Columns (off-white) encode fused tensors
Rows (light pink) encode fused iteration order
Cells (cool gray) encode IR nodes and named pointers to streams
d) Fusion Table
Codegen
Evaluation (Section 8)
Dataflow Simulator
FPGA
Heuristic
e) SAMML Fused Dataflow Graph

Figure 6. Compilation flow of FuseFlow. (a) PyTorch model. (b) Einsum expressions with optional user schedules (red) and sparse formats (green). (c) Cross-expression fusion of Fuse regions yields fused Einsum subgraphs and a partial-order graph. (d) Fusion tables encode iteration (rows), tensors (columns), and IR nodes/streams (cells). (e) Codegen emits a SAMML fused dataflow graph; evaluation targets a dataflow simulator, FPGA, or a heuristic model (Section 8).



$$\texttt{Fuse}\{\forall_{ijk}A_{ij} = B_{ik}C_{kj}$$
$$\forall_{ikj}D_{ij} = A_{ik}B_{kj}\}$$

(a) Unfused expressions

$\forall_{iu_1j}D_{ij} = A_{iu_1}B''_{u_1j}$
$\forall_{iu_1u_0}A_{iu_1} = B'_{iu_0}C_{u_0u_1}$

(b) Fused Einsums

i
u1
u0
j
One valid topological sort:
i → u_1 → u_0 → j
$\forall_{iu_1u_0j}D_{ij} = B'_{iu_0}C_{u_0u_1}B''_{u_1j}$

(c) Fused Einsum w/ partial order graph

Figure 7. The input (a) with sparse matrix B stored in CSR format and equivalent output representations (b) and (c) of our automated cross-expression fusion algorithm.

expression—unspecified orders remain free—and (ii) per-tensor mode order required by storage format (e.g. CSR requires $i \longrightarrow j$, denoted as $[i, j] = [0, 1]$). For example, suppose we fuse: $A_{ij} = B_{ik}C_{kj}$ and $E_{ij} = B_{ik}A_{kj}$, both with inner product dataflow ($i \longrightarrow j \longrightarrow k$). Simple index substitution conflicts with A's required mode orders $[0, 1]$ vs. $[1, 0]$, forcing discordant traversal. When a tensor is used multiple times, FuseFlow treats each use as a distinct *tensor view* (denoted with primes (in Figure 7b); each view is annotated with its required mode order.

We therefore introduce a fusion algorithm that extends index substitution [13] with ordering constraints maintained in a directed graph called a partial order graph (POG).

While processing each local expression, FuseFlow inserts edges into the POG to represent both dataflow order and mode order constraints so that global consistency is maintained across the fused region.

The POG allows FuseFlow to track index order constraints of a fused expression based on the mode order of the input sparse tensors, and final output tensor as well as the local dataflow order of each expression to be fused. This cross-expression fusion is achieved in several steps, as shown in Figure 7. For each expression to be fused:

1) Rename local index variables: For each tensor in the expression, FuseFlow replaces all local reduction indices (those not on the left-hand side) with fresh indices (denoted as u-indices in Figure 7b) and adds POG edges to enforce each sparse tensor view's mode order constraints (i.e. $i \longrightarrow u_0$ based on B' in Figure 7b).

2) Build fused Einsum producer-consumer edges: For each tensor, FuseFlow connects producer uses with their consumers, as shown by the arrows in Figure 7b, while substituting index variables. This fused Einsum expressions representation is similar to the ideas presented by [70].

3) Propagate order constraints: As each producer-to-consumer edge is added, FuseFlow inserts directed edges in the POG between indices that have an outer-to-inner ordering relationship.

4) Handle multiple tensor uses: For any tensor used multiple times, FuseFlow assigns a distinct view to each use, annotating it with the required mode order. Equivalent views are merged, where equivalence means: (i) identical mode-order sequences and (ii) equivalent index maps. If distinct views of the same tensor induce conflicting ordering constraints (detected as a cycle in the POG) and no concordant topological order exists, FuseFlow materializes a permuted copy of the tensor (a higher-order transpose) for one of the views to break the cycle.

By applying the above steps to every expression to be fused, we accumulate a unified index-space representation with all necessary constraints. Throughout this process, the POG ensures that global index ordering remains consistent with all mode orders and user-scheduled dataflow orders encountered. If the graph remains acyclic, FuseFlow performs a topological sort to produce all valid global dataflow orders that respects all constraints. FuseFlow can traverse the fused Einsum representation and emit a single, fully fused Einsum (Figure 7c), equivalent to the fused representation in Figure 7b The full algorithm can be found in Section A.5.

6 Lowering with Fusion tables

To facilitate code generation, we introduce *fusion tables*, a tabular lowering IR that memoizes intermediate streams and defers node creation. It provides named pointers to each component in the final dataflow graph, allowing for references to components that have not been created yet. Programming a dataflow machine differs from typical loop-based paradigms in that it relies on a spatial connection topology of operators/nodes and data rather than iterative control flow. During lowering, a dataflow compiler maps how dataflow primitives are connected to assemble the final dataflow graph. Fusion tables capture these connections by treating each operator as a cell in a table with pointers representing data movement through nodes. Fusion tables also enable fusion across multiple expressions to target a dataflow system. We first provide details on the fusion tables (Section 6.1) and show how they are used by FuseFlow for code generation (Section 6.2).

6.1 Fusion Table IR

As discussed in Section 3, our compiler must dynamically adjust and interleave the topology of iteration and computation pipelines. We use a fusion table IR to accomplish this task. A fusion table allows the compiler to defer materializing the final graph and instead work with a named, structured representation. The compiler can assign placeholders to dataflow nodes that are not yet created, enabling later pointers (references) to those future nodes by name. In essence, the fusion table provides a spatially organized plan of the fused index iteration and operations, which can be manipulated freely before committing to a final dataflow graph.

Fusion Table Structure. A fusion table can be thought of as a two-dimensional grid. Rows represent index variables or value results, which are ordered by fused iteration index order. For example, in Figure 8a the table contains rows that represent the iteration order of the expression ($i \rightarrow k \rightarrow j$) with the last row for value computation. Columns represent tensor expressions, with each operand or intermediate result in the fused expression assigned to its own column. By reading across a given row, we see all tensors involved at that loop level. In other words, rows slice the computation by control (loop levels), while columns slice it by data (tensors). Cells occupy the intersection of a row and a column; each cell represents either an operation to perform or a pointer to another cell's operation. A cell can be one of two types:

1) Primitive cell: creates a new dataflow node corresponding to a fundamental operation as defined in Section 2. Placing a primitive cell in the table is akin to planning the instantiation of that dataflow node in the SAMML graph.

2) Reference cell: points to an existing cell that reuses an already generated node (e.g., $\langle T^0_i \rangle$ in Figure 8c) or passes through values when an index variable is not needed for the current operation (e.g. $\langle T^0_k \rangle$ in Figure 8c).

By ordering indices and operations in this structured grid, the compiler spatially captures the relationships between tensor computations and their iteration space. Figure 8 illustrates this concept using a sparse matrix multiplication (SpMM) example. Figure 8a shows an empty fusion table for the SpMM kernel $T^0_{ij} = \sum_k A_{ik} \times X_{kj}$ with $i \rightarrow k \rightarrow j$ iteration order. Figure 8b shows the table partially filled as the compiler processes the fused Einsum expressions step by step. Note that some cells are already referencing nodes (marked by angle brackets) that have not been materialized yet, indicating future connections (i.e. $\langle A_{val} \rangle$ referencing $\langle A_j \rangle$). In Figure 8c, the fusion table is fully populated after handling all operations. Finally, Figure 8d depicts the final dataflow graph generated from the completed fusion table, with the color coding showing how each cell in the table maps to a component in the final graph.

Fusion Table Construction. To understand how fusion tables are constructed by FuseFlow, we walk through the fusion table construction for the SpMM example shown in Figure 8. FuseFlow populates fusion tables by processing each operation in the input program one by one. FuseFlow uses several steps for each operation, as detailed below:

1) Insert level scanners and value nodes: For every input tensor view, FuseFlow assigns level scanner cells and value cells in a top-down fashion following the dataflow order. If an input tensor is not the result of a prior operation, a value cell is placed. In Figure 8c, LS cells (in dark green) for $\hat{A}_i$, $\hat{A}_k$, $\hat{X}_k$, and $\hat{X}_j$ are created, along with the corresponding value cells (labeled "Val" in light green).

2) Insert repeat and compute nodes: When processing intermediate tensor views, FuseFlow identifies index variables missing from each input operand's tensor view and assigns Rep nodes for each of these cases. Repeat nodes' inputs include the stream being repeated and the repeat signal. The computation pipeline for the intermediate tensor view

	$\hat{A}_{ik}$	X_{kj}	T^0_{ij}
i			
k			
j			
val			

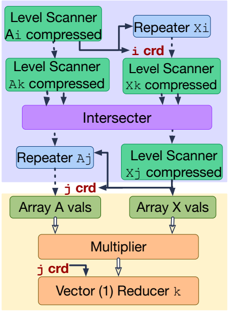
(a) Initial empty fusion table

	$\hat{A}_{ik}$	X_{kj}	T^0_{ij}
i	LS(root)		
k	LS($\langle \hat{A}_i \rangle$)		
j			
val	Val($\langle \hat{A}_j \rangle$)		

(b) Partially filled table

	$\hat{A}_{ik}$	X_{kj}	T^0_{ij}
i	LS(root)	Rep(root, $\langle \hat{A}_i \rangle$)	$\langle \hat{A}_i \rangle$
k	LS($\langle \hat{A}_i \rangle$) Intersect$_k$	LS($\langle X_i \rangle$)	$\langle T^0_i \rangle$
j	Rep($\langle \hat{A}_k \rangle$, $\langle X_j \rangle$)	LS($\langle X_k \rangle$)	$\sum_k (\langle \hat{A}_{\text{val}} \rangle \times \langle X_{\text{val}} \rangle)_{\text{crd}_0}$
val	Val($\langle \hat{A}_j \rangle$)	Val($\langle X_j \rangle$)	$\sum_k (\langle \hat{A}_{\text{val}} \rangle \times \langle X_{\text{val}} \rangle)_{\text{val}}$

(c) Fully constructed table



(d) Partial SAMML graph.

Figure 8. Fusion table construction for sparse matrix multiplication $\forall_{ikj}\ T^0_{ij} = \hat{A}_{ik} X_{kj}$: (a) an empty fusion table; (b) a partially filled table as the compiler walks the expression DAG—highlighting how it can reference cells (nodes) not yet materialized; (c) the fully populated fusion table; (d) the generated dataflow graph generated from the completed table. Colors in the table visually indicate how cell components map to nodes in Figure 8d. Shaded column mark which column FuseFlow has processed.

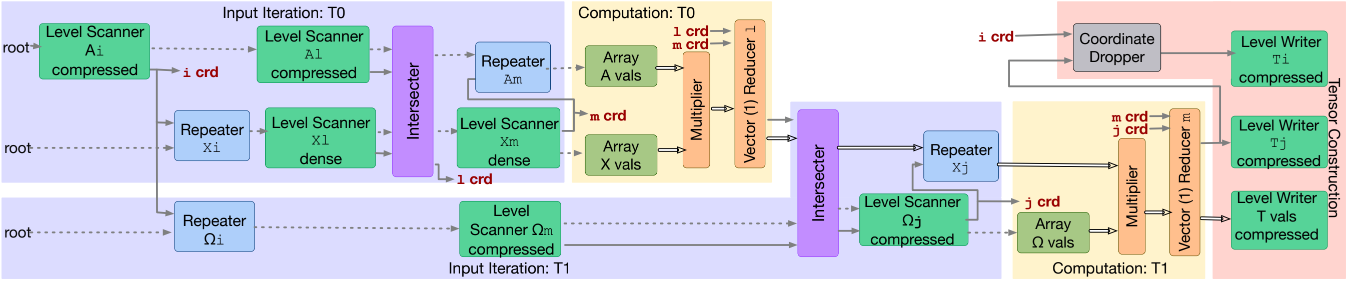


Figure 9. SAMML graph for the neighborhood subcomputation $T^{nbor}_{ij} = \hat{\mathbf{A}}_{il} X_{lm} \Omega^{(2)}_{mj}$ from GraphSAGE with $i \rightarrow l \rightarrow m \rightarrow j$ order. See Figure 19 in Section A.2 for the full fusion table that generates this graph.

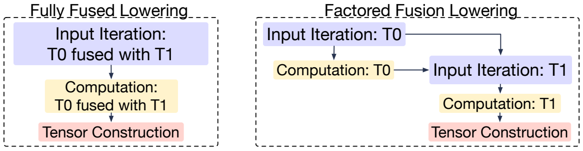


Figure 10. How fully fused input iteration (Figure 5a) vs. factored input iteration (Figure 5b) manifests spatially in sparse dataflow graphs. Factored fusion splits the iteration sub-graphs (two blue regions), while fully-fused iteration keeps the iteration sub-graph intact (one blue region).

is also assigned at this point, meaning ALU and reduction nodes (if applicable) are inserted. In Figure 8c, the compute pipeline cell (in orange) is inserted for T^0.

3) Handle higher-order reductions: When a higher-order reduction is encountered, the compiler updates the relevant cells with the reduction outputs. For example, in Figure 8c, a first-order higher-order reduction (e.g., $\langle T^0_j \rangle$) is applied: it consumes a value stream and two coordinate streams. It produces a reduced value stream along with a final reduced output coordinate.

4) Insert stream merging nodes: After processing all tensors, the compiler identifies index variables shared across multiple tensor views. It then creates stream merging nodes (intersect or union) by relocating existing level scanner cells into newly created merged cells, effectively rewiring the graph before code generation. In Figure 8c, FuseFlow merges cells for $\hat{A}_k$ and X_j into an intersect (shown in purple). If output coordinates coming from a higher-order reducer need merging, the corresponding reduction node's cell is moved instead. This step shows how the compiler modifies the input iteration pipeline through simple cell movement.

Why use a fusion table? Conventional compiler representations, such as dependency graphs or value-numbering approaches, represent computations as fixed nodes and edges [28]. However, these approaches lack flexibility: once nodes are instantiated, it becomes challenging to reorder or restructure them dynamically without cumbersome graph transformations. In contrast, the fusion table is designed to be a malleable blueprint that the compiler can adjust before final code generation. This brings two key benefits:

1) Deferred graph construction for flexibility: As described previously, fusion tables defer node creation and memoize intermediate streams, allowing for references before creation. This feature lets the compiler rewire node connections without complex graph manipulation.

2) Explicit grid layout: Rows encode fused iteration (control) and columns encode tensor views (data), with cells as operations or references. This grid makes dependencies and reuse obvious, simplifying fused graph generation and mapping cleanly to sparse dataflow hardware.

6.2 Code Generation

FuseFlow generates the final dataflow graph by traversing the fusion table top-down, instantiating nodes for coordinate iteration and computation as dictated by the table structure. Starting from the output tensor value cell, FuseFlow recursively expands dependent cells upward, constructing the graph nodes and streams that correspond to the fused loops. Finally, tensor construction nodes (level writers, coordinate droppers) are added to finalize outputs. Figure 9 shows the final dataflow graph generated for a fused GraphSAGE kernel (see Section A.2 for its corresponding fusion table). The result is a hardware-efficient dataflow graph in the factored-iteration style. By design, our fusion table lowering yields a factored (not global) iteration space, interleaving input iteration and computation, as motivated in Section 3 and illustrated in Figure 10 on the right. A direct comparison of the SAM graph with global iteration space and the SAMML graph with factored iteration space is shown in Section A.4. These lowering algorithm implications are further discussed in Section A.4. The full lowering algorithm can be found in Section A.6. Lowering the FuseFlow generated SAMML graph to real hardware follows prior work [35], as each node lends itself to VLSI implementations. Therefore, SAMML graphs compose to represent sparse deep learning workloads on dataflow accelerators.

7 FuseFlow Implementation

We implement FuseFlow within the MLIR compiler framework. We reimplement SAM as an MLIR dialect with a new FuseFlow MLIR compilation path as described in Section 4. Specifically, FuseFlow compiles the MLIR Sparse and Linalg dialects [5] to SAM graphs and lets users control fusion and dataflow ordering through a scheduling language as shown in Figure 11. FuseFlow also includes additional optimizations, which were discussed in the context of the SAM IR but not previously present in any SAM compiler [28]. These optimizations—such as parallelization, sparsity blocking, dataflow ordering, and an analytical fusion heuristic—are necessary for efficient, large-scale ML. Users can guide these optimizations through a command-line scheduling interface.

Parallelization. Our compiler applies vectorization and loop unrolling optimizations inspired by SAM [28], concretizing them via stream parallelizer and serializer primitives. Users specify parallelization by selecting index variables and parallelization factors. The compiler partitions tensor coordinates and duplicates compute subgraphs, distributing work across parallel streams and merging results upon completion.

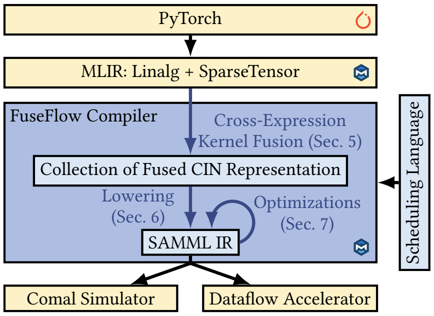


Figure 11. System overview of FuseFlow, where blue boxes and arrows denote contributions.

Sparsity Blocking. FuseFlow efficiently targets structured sparsity (e.g., block-sparsity [12, 67]) by mapping dense blocks onto the innermost coordinates of tensors. Sparse iteration occurs at outer levels, with dense blocks streamed directly to vectorized ALUs, maintaining sparsity-driven dataflow benefits while enhancing computational density.

Dataflow Ordering. FuseFlow enumerates valid dataflow orders that do not break fusion, enabling users or autotuning frameworks to select schedules to optimize performance. Each dataflow order yields different SAMML graphs and asymptotic efficiencies.

Fusion Heuristic. Our heuristic rapidly estimates FLOPs and memory transfers of fused programs without full simulation. Users input tensor dimensions, sparsity percentages, and intersection rates. The fusion heuristic enables lightweight analysis to quickly prune suboptimal schedules, significantly reducing the optimization search space.

8 Evaluation

To evaluate the techniques presented in this paper, we use FuseFlow to compile various real-world sparse machine learning applications to our SAMML IR and simulate their end-to-end cycle-accurate performance. We showcase our compiler's generality by using four different model classes. We also perform ablation studies on various key features of FuseFlow in Section 8.3 to Section 8.6. FuseFlow's general algorithmic fusion mechanism allows us to explore a vast space of fusion and dataflow schedules, unlocking speedups that were previously unattainable with existing frameworks for sparse workloads on dataflow hardware.

8.1 Methodology

Benchmark Applications and Datasets. We evaluate FuseFlow on four sparse machine learning model classes across different domains: Sparse Autoencoder (SAE) [41] (3 layers), Graph Convolutional Networks (GCN) [30] (2 layers), GraphSAGE [22] (2 layers), and GPT-3 Small (125M parameters) with BigBird attention [67] (sequence length of 1024). For SAE, we randomly sampled 5 images. Real world datasets used for each model as shown in Section A.3.

FuseFlow Compiler. FuseFlow is implemented in MLIR within LLVM 19.1.0, with dependencies including Protobuf 28.1 and OR-Tools 9.10, and compiled using GCC 14.2.0. For all evaluated benchmarks, we select by default the first valid topological sort provided by FuseFlow (see Section 5).

Compilation Overhead. All models compile in < 750ms.

Simulator Framework. Our Comal simulator models the architectural behavior of each IR node and tracks cycles based on fully pipelined dataflow graphs, as in SAM [28]. It incorporates HBM2 memory simulation via Ramulator 2.0 [39], a cycle-accurate DRAM simulator, and provides instrumentation to estimate operations and memory accesses. Comal uses the DAM simulation framework [69] in Rust 1.87.0 with all simulation functional results verified against a dense PyTorch implementation.

8.2 Hardware Validation

As in SAM [28], our primary evaluation uses a cycle-accurate simulator. At the time of writing, no existing accelerator broadly supported end-to-end sparse ML. The closest, Onyx [35], a coarse-grained reconfigurable array (CGRA) targeting sparse tensor algebra, is insufficient for sparse ML as it lacks support for nonlinear and masking operations. Therefore, we validate simulator fidelity against a post-synthesis RTL simulation of a Xilinx VU9P (AWS F1) design generated from FuseFlow's SAMML. FuseFlow lowers to Comal for simulation and to Vitis HLS for FPGA, using a minimal, proof-of-concept HLS library to instantiate streaming operators. We select kernels that fit entirely in on-chip BRAM to isolate compute, including partially fused (one-layer) GCN and GraphSAGE, fully fused GCN, and BigBird attention. Concretely, GCN (11 kernels) and GraphSAGE (13 kernels) on KarateClub [66] and GPT-3 (17 kernels) with sequence length 64. For each model, we normalize kernel cycle counts by the best across both backends and report trend agreement via R^2 over kernels. We observe a strong agreement of R^2=0.991 in Figure 13. Fairly recently, Chen et al. [8] introduced Opal, a follow-on CGRA to Onyx with sparse ML support and improved dataflow orderings, which we plan to target as future work.

8.3 Fusion

We use FuseFlow to generate fused graphs for each model comparing the performance at different fusion levels against unfused configurations. Accelerating unoptimized code is rarely useful, making fusion an essential avenue for optimization. We show that it is important to identify the right fusion granularity to ensure meaningful performance gains. For SAE, GCN, and GraphSAGE, full fusion marks fusion regions around the end-to-end computations, while partial fusion focuses on fusion regions within encoder/decoder or individual layers. For GPT-3 with BigBird Attention, shape operators (e.g., reshape, transpose) naturally limit fusion, dividing decoders into multiple subsets. Full fusion across decoders combines these subsets across decoder boundaries.

Model class	Avg % Error FLOPs	 Bytes
GPT-3 (block=16)	1.8	5.7
GCN	2.8	9.6
GraphSAGE	2.6	11.5

Table 2. Average percent error of FLOPs and memory accesses on OGB-Collab.

Model	Unconstr.	Constr.
GCN	$2.0 \cdot 10^{8*}$	$6.3 \cdot 10^7$
GraphSAGE	$3.9 \cdot 10^7$	$1.1 \cdot 10^3$

Table 3. Number of dataflow orders with and without local constraints (*capped, estimated up to ~10^{15}).

As demonstrated in Figure 12, GPT-3 achieves up to ~2.7× improvement with full fusion. Conversely, SAE, GCN, and GraphSAGE experience performance degradation under full fusion due to increased computational overhead from nested matrix multiplications. Partial fusion is more effective for these models, improving performance by up to ~2.6× (GCN on OGB-collab) and ~3.9× (GraphSAGE on OGB-mag). These results illustrate that optimal fusion granularity depends heavily on the model.

Analyzing GCN further, partially fusing the first layer significantly reduces bytes transferred compared to unfused versions, improving operational intensity (Figure 14). Fully fused GCN, while having higher operational intensity, suffers from recomputation, degrading overall performance. Thus, optimal fusion must carefully balance reduced data movement against additional computation.

Finally, our heuristic effectively estimates computational and memory costs. It correctly predicts optimal fusion configurations and enables early pruning of suboptimal strategies as shown in Table 2, which shows the average percentage errors for FLOPs and memory accesses on GPT-3 (w/ block size 16), GCN, and GraphSAGE on OGB-Collab.

8.4 Parallelization

We evaluate FuseFlow's capacity to enhance performance by generating parallel dataflow graphs. We first perform a sweep of parallelization factors from 1 to 64 to parallelize a single index variable in BigBird attention, as shown in Figure 15a. We find that FuseFlow's generated program scales well with the amount of added parallelism. We also demonstrate FuseFlow's ability to parallelize across different index variables, and its support for nested parallelism. Figure 15b shows the impact of parallelizing two different index variables in BigBird attention, highlighting the performance effects when sweeping across various parallelization factors, as well as applying a constant factor of 4 across both levels at the same time. While varying parallelization location, we find that FuseFlow's generated programs are able to obtain performance improvements relative to the parallelization

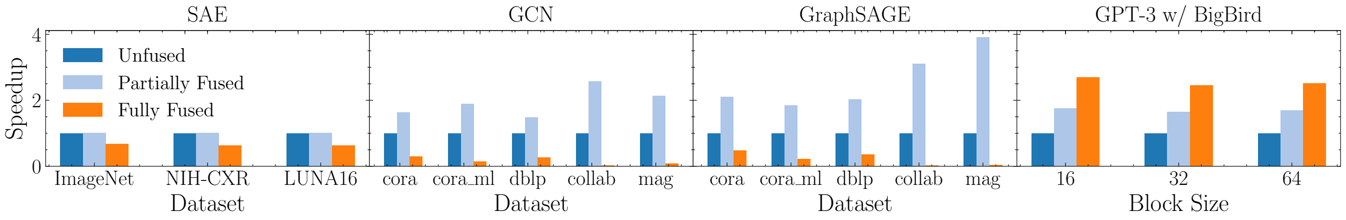


Figure 12. The effect of fusion on dataflow performance across various models.

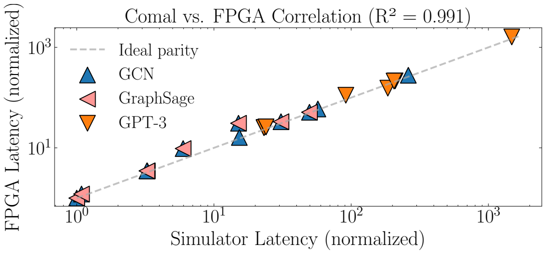


Figure 13. Latency correlation across kernels across various models. Colors denote models; the dashed line shows parity.

factor. Parallelizing both levels at the same time by a parallel factor of 4 results in ~15.9x speedup.

8.5 Block Sparse Computation

We evaluate FuseFlow's sparsity blocking on the BigBird attention module for all three block configurations. We compare the performance of this blocked approach with our results in Figure 12, which treats the tensors as unstructured sparse computation. The results in Figure 16 show that the speedup obtained is proportional to the block size.

8.6 Dataflow Ordering

We evaluate the impact of dataflow ordering by varying the order of nested matrix multiplication (matmul)—a core operation in GCN and GraphSAGE—using KarateClub [66]. As shown in Figure 17, suboptimal orders cause up to ~29× slowdown compared to the best. Leveraging the best order thus provides an end-to-end speedup of ~29× for fused GCN and GraphSAGE models. Additionally, constraining each matmul kernel to the best dataflow order significantly reduces the design-space size by 68.5%-99.9%, as shown in Table 3. Without these constraints, GCN alone has an impractically large number of possible dataflows (estimated up to ~10^{15}), so we limit the search space to 2×10^8 configurations in FuseFlow.

9 Related Work

This paper shows how to compile sparse ML models to a sparse dataflow abstract machine. We review related work on sparse compilation to dataflow hardware, fusion in sparse tensor algebra frameworks, and sparse ML systems.

9.1 Compiling Sparse Tensor Algebra to Dataflow

Several techniques have been proposed for compiling sparse tensor algebra to dataflow hardware. Closest to our work is the Custard compiler [28]. Custard compiles sparse tensor algebra expressions to SAM graphs with intra-layer iteration fusion (IIF). Moreover, the compiler for the Onyx chip [35] maps SAM graphs to physical sparse CGRA hardware. The SAMML dataflow graphs we target are an extension of SAM graphs with additional ML primitives. Unlike Custard, our work supports a different form of IIF through factored iteration and introduces cross-expression fusion (EKF).

An extension to the Spatial compiler [34, 50] for Capstan hardware [47, 50] compiles computations written in parallel patterns—a loop-based declarative language—to sparse dataflow hardware. Moreover, the Stardust [27] compiler can compile high-level sparse tensor algebra languages to these parallel patterns. Stardust, like Custard, only supports tensor expressions and cannot compile entire sparse ML models or provide cross-expression fusion.

Finally, there is a class of work on compiling general-purpose C code to reconfigurable dataflow accelerators. Weng et al. [59] describe a compiler from annotated sparse loops to their SPU hardware [11], and Gobieski et al. [19] presents a co-designed compiler from general code to a CGRA. Both works support sparse loops in their general-purpose input code but do not compile from higher-level sparse languages.

9.2 Fusion in Sparse Tensor Frameworks

The TACO compiler [32] showed how to generate fused loops for sparse tensor algebra expressions on CPUs and GPUs [52]. Later sparse tensor algebra compilers have expanded such fusion capabilities [70] to expressions over additional data structures [1, 10, 37, 65] and operations [25, 36, 49, 55]. The fusion support in these compilers, however, is limited to tensor algebra expressions and CPU/GPU compilation [62]. Other specialized frameworks such as FusedMM [48] and SeaStar [60] support sparse operator fusion but are limited to specific patterns (e.g., SDDMM+SpMM operations or GNN message-passing patterns). Zhou et al. [70] introduce techniques that identify and avoid four common redundancy types in IIF for sparse tensor algebra. Their compiler, ReACT, is most similar to ours as it introduces a representation close to our fused Einsums to and generates factored iteration code.

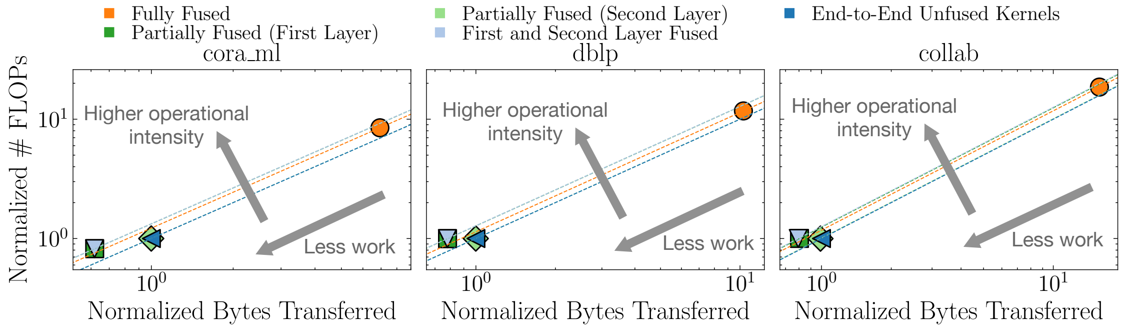
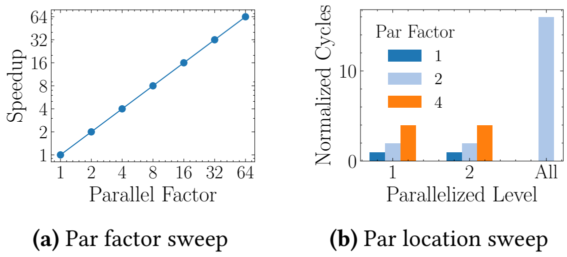


Figure 14. GCN FLOPs and memory accesses normalized to the unfused baseline across datasets. Dashed lines show operational intensity. Fusion increases operational intensity, but full fusion's recomputation increases both FLOPs and memory.



(a) Par factor sweep

(b) Par location sweep

Figure 15. The effect of parallelization factor and parallelization location for BigBird attention.

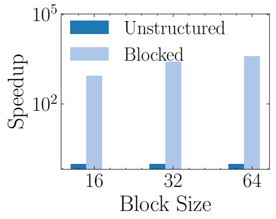


Figure 16. Performance of block sparse computation for BigBird attention.

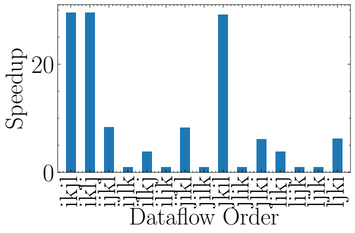


Figure 17. Dataflow order sweep for nested matmul normalized by worst dataflow.

However, ReACT cannot fuse across multiple independent expressions. Building upon these works, we show how to fuse across independent Einsum expressions in an ML model and how to do so when compiling to dataflow machines.

The TeAAL framework [40] presents a declarative language of cascaded Einsums to describe sparse tensor algebra accelerators and generates an accelerator simulator and performance model from that language. TeAAL represents multiple Einsum expressions similar to our work, but our work generates fused code across those Einsum expressions. While TeAAL is a tool for modeling dataflow accelerators, FuseFlow generates a program configuration or mapping to dataflow accelerators through the SAMML IR along with a simulation of that program.

9.3 Sparse ML Compilation and Frameworks

A few ML frameworks have been designed with support for sparse tensors and, hence, sparse ML models. Scorch [63] describes several techniques needed to implement a version of the PyTorch API that supports sparse as well as dense tensors. The MLIR Linalg + SparseTensor dialects [5] combined with the MLIR lowering from PyTorch to Linalg [20] also provides a sparse ML framework for CPUs. Our compilation techniques complement these frameworks with a compilation path to sparse dataflow hardware. Domain-specific libraries like PyTorch Geometric (PyG) [15] and Deep Graph Library (DGL) [56] integrate sparse computation into specific applications, but they lack the generality needed for targeting a broader range of sparse models.

10 Conclusion

FuseFlow introduces key pieces in compiling large-scale sparse ML models expressed in PyTorch to dataflow architectures. We believe such frameworks are essential for making productive use of these architectures. Our work opens up several avenues of future compiler work to develop further optimizations on sparse dataflow and to map from SAMML to physical RDA hardware.

Acknowledgments

We would like to thank James Dong, Benjamin Driscoll, Chris Gyurgyik, Konstantin Hossfeld, Jungwoo Kim, Scott Kovach, Devanshu Ladsaria, Sai Gautham Ravipati, AJ Root, Alex Rucker, Nathan Sobotka, Gina Sohn, Bala Vinaithirthan, Rohan Yadav, Bobby Yan, Genghan Zhang, and Qizheng Zhang for their feedback on this paper. We would also like to thank Bo Wun Cheng and Zhouhua Xie for help on technical ideas. We would especially like to thank Shiv Sundram and Mark Horowitz for their feedback on both the work and the paper. This work was supported in part by the National Science Foundation under grant number 2216964, DARPA under the Machine learning and Optimization-guided Compilers for Heterogeneous Architectures (MOCHA) program (award number HR00112520038), and by the Naval Surface Warfare Center under Agreement No. N00164-23-9-G057-01. This research was also supported in part by the Stanford Data Analytics for What's Next (DAWN) Affiliate Program and the PRISM center, one of seven centers in JUMP 2.0, a Semiconductor Research Corporation (SRC) program sponsored by DARPA. Olivia Hsu was supported in part by an NSF GRFP. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of the aforementioned funding agencies.

References

[1] 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* (Montréal, QC, Canada) *(CGO '23)*. Association for Computing Machinery, New York, NY, USA, 41–54. doi:10.1145/3579990.3580020

[2] Willow Ahrens, Fredrik Kjolstad, and Saman Amarasinghe. 2022. Autoscheduling for sparse tensor algebra with an asymptotic cost model. In *Proceedings of the 43rd ACM SIGPLAN International Conference on Programming Language Design and Implementation* (San Diego, CA, USA) *(PLDI 2022)*. Association for Computing Machinery, New York, NY, USA, 269–285. doi:10.1145/3519939.3523442

[3] Jason Ansel, Edward Yang, Horace He, Natalia Gimelshein, Animesh Jain, Michael Voznesensky, Bin Bao, Peter Bell, David Berard, Evgeni Burovski, Geeta Chauhan, Anjali Chourdia, Will Constable, Alban Desmaison, Zachary DeVito, Elias Ellison, Will Feng, Jiong Gong, Michael Gschwind, Brian Hirsh, Sherlock Huang, Kshiteej Kalambarkar, Laurent Kirsch, Michael Lazos, Mario Lezcano, Yanbo Liang, Jason Liang, Yinghai Lu, C. K. Luk, Bert Maher, Yunjie Pan, Christian Puhrsch, Matthias Reso, Mark Saroufim, Marcos Yukio Siraichi, Helen Suk, Shunting Zhang, Michael Suo, Phil Tillet, Xu Zhao, Eikan Wang, Keren Zhou, Richard Zou, Xiaodong Wang, Ajit Mathews, William Wen, Gregory Chanan, Peng Wu, and Soumith Chintala. 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* (La Jolla, CA, USA) *(ASPLOS '24)*. Association for Computing Machinery, New York, NY, USA, 929–947. doi:10.1145/3620665.3640366

[4] Manya Bansal, Olivia Hsu, Kunle Olukotun, and Fredrik Kjolstad. 2023. Mosaic: An Interoperable Compiler for Tensor Algebra. *Proc. ACM Program. Lang.* 7, PLDI, Article 122 (June 2023), 26 pages. doi:10.1145/3591236

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

[6] Aleksandar Bojchevski and Stephan Günnemann. 2018. Deep Gaussian Embedding of Graphs: Unsupervised Inductive Learning via Ranking. arXiv:1707.03815 [stat.ML] https://arxiv.org/abs/1707.03815

[7] Alex Carsello, Kathleen Feng, Taeyoung Kong, Kalhan Koul, Qiaoyi Liu, Jackson Melchert, Gedeon Nyengele, Maxwell Strange, Keyi Zhang, Ankita Nayak, Jeff Setter, James Thomas, Kavya Sreedhar, Po-Han Chen, Nikhil Bhagdikar, Zachary Myers, Brandon D'Agostino, Pranil Joshi, Stephen Richardson, Rick Bahr, Christopher Torng, Mark Horowitz, and Priyanka Raina. 2022. Amber: A 367 GOPS, 538 GOPS/W 16nm SoC with a Coarse-Grained Reconfigurable Array for Flexible Acceleration of Dense Linear Algebra. IEEE Symposium on VLSI Technology & Circuits.

[8] Po-Han Chen, Bo Wun Cheng, Michael Oduoza, Zhouhua Xie, Rupert Lu, Sai Gautham Ravipati, Kalhan Koul, Alex Carsello, Yuchen Mei, Mark Horowitz, and Priyanka Raina. 2025. Opal: A 16-nm Coarse-Grained Reconfigurable Array SoC for Full Sparse Machine Learning Applications. *IEEE Solid-State Circuits Letters* 8 (2025), 293–296. doi:10.1109/LSSC.2025.3613245

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

[10] Stephen Chou, Fredrik Kjolstad, and Saman Amarasinghe. 2018. Format Abstraction for Sparse Tensor Algebra Compilers. *Proc. ACM Program. Lang.* 2, OOPSLA, Article 123 (October 2018), 30 pages.

[11] Vidushi Dadu, Jian Weng, Sihao Liu, and Tony Nowatzki. 2019. Towards General Purpose Acceleration by Exploiting Common Data-Dependence Forms. In *Proceedings of the 52nd Annual IEEE/ACM International Symposium on Microarchitecture* (Columbus, OH, USA) *(MICRO '52)*. Association for Computing Machinery, New York, NY, USA, 924–939. doi:10.1145/3352460.3358276

[12] Tri Dao, Dan Fu, Stefano Ermon, Atri Rudra, and Christopher Ré. 2022. Flashattention: Fast and memory-efficient exact attention with io-awareness. *Advances in Neural Information Processing Systems* 35 (2022), 16344–16359.

[13] N.G de Bruijn. 1972. Lambda calculus notation with nameless dummies, a tool for automatic formula manipulation, with application to the Church-Rosser theorem. *Indagationes Mathematicae (Proceedings)* 75, 5 (1972), 381–392. doi:10.1016/1385-7258(72)90034-0

[14] Jia Deng, Wei Dong, Richard Socher, Li-Jia Li, Kai Li, and Li Fei-Fei. 2009. ImageNet: A large-scale hierarchical image database. In *Proceedings of the IEEE conference on computer vision and pattern recognition (CVPR)*. 248–255.

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

[16] Amin Firoozshahian, Joel Coburn, Roman Levenstein, Rakesh Nattoji, Ashwin Kamath, Olivia Wu, Gurdeepak Grewal, Harish Aepala, Bhasker Jakka, Bob Dreyer, et al. 2023. Mtia: First generation silicon targeting meta's recommendation systems. In *Proceedings of the 50th Annual International Symposium on Computer Architecture*. 1–13.

[17] Trevor Gale, Erich Elsen, and Sara Hooker. 2019. The state of sparsity in deep neural networks. *arXiv preprint arXiv:1902.09574* (2019).

[18] Trevor Gale, Matei Zaharia, Cliff Young, and Erich Elsen. 2020. *Sparse GPU Kernels for Deep Learning*. IEEE Press, Chapter 17, 1–14.

[19] Graham Gobieski, Souradip Ghosh, Marijn Heule, Todd Mowry, Tony Nowatzki, Nathan Beckmann, and Brandon Lucia. 2023. RipTide: A Programmable, Energy-Minimal Dataflow Compiler and Architecture. In *Proceedings of the 55th Annual IEEE/ACM International Symposium on Microarchitecture* (Chicago, Illinois, USA) *(MICRO '22)*. IEEE Press, 546–564. doi:10.1109/MICRO56248.2022.00046

[20] Google. 2021. MLIR Sparsifier. https://developers.google.com/mlir-sparsifier

[21] Fred G. Gustavson. 1978. Two Fast Algorithms for Sparse Matrices: Multiplication and Permuted Transposition. *ACM Trans. Math. Softw.* 4, 3 (1978).

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

[23] Song Han, Jeff Pool, John Tran, and William J. Dally. 2015. Learning both Weights and Connections for Efficient Neural Networks.

arXiv:1506.02626 [cs.NE] https://arxiv.org/abs/1506.02626

[24] 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.

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

[26] Torsten Hoefler, Dan Alistarh, Tal Ben-Nun, Nikoli Dryden, and Alexandra Peste. 2021. Sparsity in deep learning: Pruning and growth for efficient inference and training in neural networks. *Journal of Machine Learning Research* 22, 241 (2021), 1–124.

[27] Olivia Hsu, Alexander Rucker, Tian Zhao, Varun Desai, Kunle Olukotun, and Fredrik Kjolstad. 2025. Stardust: Compiling Sparse Tensor Algebra to a Reconfigurable Dataflow Architecture. In *Proceedings of the 23rd ACM/IEEE International Symposium on Code Generation and Optimization* (Las Vegas, NV, USA) *(CGO '25)*. Association for Computing Machinery, New York, NY, USA, 628–643. doi:10.1145/3696443.3708918

[28] Olivia Hsu, Maxwell Strange, Ritvik Sharma, Jaeyeon Won, Kunle Olukotun, Joel S Emer, Mark A Horowitz, and Fredrik Kjølstad. 2023. The sparse abstract machine. In *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 3*. 710–726.

[29] Weihua Hu, Matthias Fey, Marinka Zitnik, Yuxiao Dong, Hongyu Ren, Bowen Liu, Michele Catasta, and Jure Leskovec. 2020. Open Graph Benchmark: Datasets for Machine Learning on Graphs. *arXiv preprint arXiv:2005.00687* (2020).

[30] Thomas N. Kipf and Max Welling. 2017. Semi-Supervised Classification with Graph Convolutional Networks. In *International Conference on Learning Representations*. https://openreview.net/forum?id=SJU4ayYgl

[31] Fredrik Kjolstad, Peter Ahrens, Shoaib Kamil, and Saman Amarasinghe. 2019. Tensor Algebra Compilation with Workspaces. *International Symposium on Code Generation and Optimization* (February 2019).

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

[33] Fredrik Berg Kjølstad. 2020. *Sparse tensor algebra compilation*. Ph. D. Dissertation. Massachusetts Institute of Technology.

[34] David Koeplinger, Matthew Feldman, Raghu Prabhakar, Yaqi Zhang, Stefan Hadjis, Ruben Fiszel, Tian Zhao, Luigi Nardi, Ardavan Pedram, Christos Kozyrakis, and Kunle Olukotun. 2018. Spatial: A Language and Compiler for Application Accelerators. In *Proceedings of the 39th ACM SIGPLAN Conference on Programming Language Design and Implementation* (Philadelphia, PA, USA) *(PLDI 2018)*. Association for Computing Machinery, New York, NY, USA, 296–311. doi:10.1145/3192366.3192379

[35] Kalhan Koul, Maxwell Strange, Jackson Melchert, Alex Carsello, Yuchen Mei, Olivia Hsu, Taeyoung Kong, Po-Han Chen, Huifeng Ke, Keyi Zhang, Qiaoyi Liu, Gedeon Nyengele, Akhilesh Balasingam, Jayashree Adivarahan, Ritvik Sharma, Zhouhua Xie, Christopher Torng, Joel Emer, Fredrik Kjolstad, Mark Horowitz, and Priyanka Raina. 2024. Onyx: A 12nm 756 GOPS/W Coarse-Grained Reconfigurable Array for Accelerating Dense and Sparse Applications. *IEEE Symposium on VLSI Technology and Circuits (VLSI)* (June 2024).

[36] Scott Kovach, Praneeth Kolichala, Tiancheng Gu, and Fredrik Kjolstad. 2023. Indexed Streams: A Formal Intermediate Representation for Fused Contraction Programs. *Proc. ACM Program. Lang.* 7, PLDI, Article 154 (June 2023), 25 pages. doi:10.1145/3591268

[37] Jie Liu, Zhongyuan Zhao, Zijian Ding, Benjamin Brock, Hongbo Rong, and Zhiru Zhang. 2024. UniSparse: An Intermediate Language for General Sparse Format Customization. *Proc. ACM Program. Lang.* 8, OOPSLA1, Article 99 (April 2024), 29 pages. doi:10.1145/3649816

[38] LLVM. [n. d.]. *Torch-MLIR*. https://github.com/llvm/torch-mlir

[39] Haocong Luo, Yahya Can Tuğrul, F. Nisa Bostancı, Ataberk Olgun, A. Giray Yağlıkçı, , and Onur Mutlu. 2023. Ramulator 2.0: A Modern, Modular, and Extensible DRAM Simulator.

[40] 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.

[41] Andrew Ng et al. 2011. Sparse autoencoder. *CS294A Lecture notes* 72, 2011 (2011), 1–19.

[42] Quan M. Nguyen and Daniel Sanchez. 2021. Fifer: Practical Acceleration of Irregular Applications on Reconfigurable Architectures. In *MICRO-54: 54th Annual IEEE/ACM International Symposium on Microarchitecture* (Virtual Event, Greece) *(MICRO '21)*. Association for Computing Machinery, New York, NY, USA, 1064–1077. doi:10.1145/3466752.3480048

[43] Wei Niu, Jiexiong Guan, Yanzhi Wang, Gagan Agrawal, and Bin Ren. 2021. Dnnfusion: accelerating deep neural networks execution with advanced operator fusion. In *Proceedings of the 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation*. 883–898.

[44] Tony Nowatzki, Vinay Gangadhar, Newsha Ardalani, and Karthikeyan Sankaralingam. 2017. Stream-dataflow acceleration. In *2017 ACM/IEEE 44th Annual International Symposium on Computer Architecture (ISCA)*. 416–429. doi:10.1145/3079856.3080255

[45] NVIDIA. [n. d.]. *TensorRT*. https://github.com/NVIDIA/TensorRT

[46] Angshuman Parashar, Michael Pellauer, Michael Adler, Bushra Ahsan, Neal Crago, Daniel Lustig, Vladimir Pavlov, Antonia Zhai, Mohit Gambhir, Aamer Jaleel, Randy Allmon, Rachid Rayess, Stephen Maresh, and Joel Emer. 2013. Triggered Instructions: A Control Paradigm for Spatially-Programmed Architectures. In *Proceedings of the 40th Annual International Symposium on Computer Architecture* (Tel-Aviv, Israel) *(ISCA '13)*. Association for Computing Machinery, New York, NY, USA, 142–153. doi:10.1145/2485922.2485935

[47] Raghu Prabhakar, Yaqi Zhang, David Koeplinger, Matt Feldman, Tian Zhao, Stefan Hadjis, Ardavan Pedram, Christos Kozyrakis, and Kunle Olukotun. 2017. Plasticine: A Reconfigurable Architecture For Parallel Paterns. *SIGARCH Comput. Archit. News* 45, 2 (June 2017), 389–402. doi:10.1145/3140659.3080256

[48] Md Khaledur Rahman, Majedul Haque Sujon, and Ariful Azad. 2021. Fusedmm: A unified sddmm-spmm kernel for graph embedding and graph neural networks. In *2021 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*. IEEE, 256–266.

[49] Alexander J Root, Bobby Yan, Peiming Liu, Christophe Gyurgyik, Aart J.C. Bik, and Fredrik Kjolstad. 2024. Compilation of Shape Operators on Sparse Arrays. *Proc. ACM Program. Lang.* 8, OOPSLA2, Article 312 (Oct. 2024), 27 pages. doi:10.1145/3689752

[50] Alexander Rucker, Matthew Vilim, Tian Zhao, Yaqi Zhang, Raghu Prabhakar, and Kunle Olukotun. 2021. Capstan: A Vector RDA for Sparsity. In *MICRO-54: 54th Annual IEEE/ACM International Symposium on Microarchitecture* (Virtual Event, Greece) *(MICRO '21)*. Association for Computing Machinery, New York, NY, USA, 1022–1035. doi:10.1145/3466752.3480047

[51] Amit Sabne. 2020. XLA : Compiling Machine Learning for Peak Performance.

[52] Ryan Senanayake, Changwan Hong, Ziheng Wang, Amalee Wilson, Stephen Chou, Shoaib Kamil, Saman Amarasinghe, and Fredrik Kjolstad. 2020. A Sparse Iteration Space Transformation Framework for Sparse Tensor Algebra. *Proc. ACM Program. Lang.* 4, OOPSLA, Article 158 (Nov. 2020), 30 pages. doi:10.1145/3428226

[53] Arnaud Arindra Adiyoso Setio, Alberto Traverso, Thomas de Bel, Moira SN Berens, Cas van den Bogaard, Piergiorgio Cerello, Hao Chen,

Qi Dou, Maria Evelina Fantacci, Bram Geurts, et al. 2017. Validation, comparison, and combination of algorithms for automatic detection of pulmonary nodules in computed tomography images: The LUNA16 challenge. *Medical image analysis* 42 (2017), 1–13.

[54] Marco Siracusa, Víctor Soria-Pardos, Francesco Sgherzi, Joshua Randall, Douglas J Joseph, Miquel Moretó Planas, and Adrià Armejach. 2023. A tensor marshaling unit for sparse tensor algebra on general-purpose processors. In *Proceedings of the 56th Annual IEEE/ACM International Symposium on Microarchitecture*. 1332–1346.

[55] Shiv Sundram, Muhammad Usman Tariq, and Fredrik Kjolstad. 2024. Compiling Recurrences over Dense and Sparse Arrays. *Proc. ACM Program. Lang.* 8, OOPSLA1, Article 103 (April 2024), 26 pages. doi:10.1145/3649820

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

[57] Xiaosong Wang, Yifan Peng, Le Lu, Zhiyong Lu, Mohammadhadi Bagheri, and Ronald M Summers. 2017. ChestX-ray8: Hospital-scale chest X-ray database and benchmarks on weakly-supervised classification and localization of common thorax diseases. In *Proceedings of the IEEE conference on computer vision and pattern recognition (CVPR)*. 2097–2106.

[58] Wei Wen, Chunpeng Wu, Yandan Wang, Yiran Chen, and Hai Li. 2016. Learning structured sparsity in deep neural networks. *Advances in neural information processing systems* 29 (2016).

[59] Jian Weng, Sihao Liu, Dylan Kupsh, and Tony Nowatzki. 2022. Unifying spatial accelerator compilation with idiomatic and modular transformations. *IEEE Micro* 42, 5 (2022), 59–69.

[60] Yidi Wu, Kaihao Ma, Zhenkun Cai, Tatiana Jin, Boyang Li, Chenguang Zheng, James Cheng, and Fan Yu. 2021. Seastar: vertex-centric programming for graph neural networks. In *Proceedings of the sixteenth european conference on computer systems*. 359–375.

[61] Yannan Nellie Wu, Po-An Tsai, Angshuman Parashar, Vivienne Sze, and Joel S. Emer. 2022. Sparseloop: An Analytical Approach To Sparse Tensor Accelerator Modeling. doi:10.48550/ARXIV.2205.05826

[62] Rohan Yadav, Alex Aiken, and Fredrik Kjolstad. 2022. SpDISTAL: Compiling Distributed Sparse Tensor Computations. *arXiv preprint arXiv:2207.13901* (2022).

[63] Bobby Yan, Alexander J Root, Trevor Gale, David Broman, and Fredrik Kjolstad. 2024. Scorch: A Library for Sparse Deep Learning. *arXiv preprint arXiv:2405.16883* (2024).

[64] Zhilin Yang, William W. Cohen, and Ruslan Salakhutdinov. 2016. Revisiting Semi-Supervised Learning with Graph Embeddings. arXiv:1603.08861 [cs.LG] https://arxiv.org/abs/1603.08861

[65] Zihao Ye, Ruihang Lai, Junru Shao, Tianqi Chen, and Luis Ceze. 2022. SparseTIR: Composable Abstractions for Sparse Compilation in Deep Learning. doi:10.48550/ARXIV.2207.04606

[66] Wayne W Zachary. 1977. An information flow model for conflict and fission in small groups. *Journal of anthropological research* 33, 4 (1977), 452–473.

[67] Manzil Zaheer, Guru Guruganesh, Kumar Avinava Dubey, Joshua Ainslie, Chris Alberti, Santiago Ontanon, Philip Pham, Anirudh Ravula, Qifan Wang, Li Yang, et al. 2020. Big bird: Transformers for longer sequences. *Advances in neural information processing systems* 33 (2020), 17283–17297.

[68] Genghan Zhang, Olivia Hsu, and Fredrik Kjolstad. 2024. Compilation of Modular and General Sparse Workspaces. *Proceedings of the ACM on Programming Languages* 8, PLDI (2024), 1213–1238.

[69] Nathan Zhang, Rubens Lacouture, Gina Sohn, Paul Mure, Qizheng Zhang, Fredrik Kjolstad, and Kunle Olukotun. 2024. The Dataflow Abstract Machine Simulator Framework. In *2024 ACM/IEEE 51st Annual International Symposium on Computer Architecture (ISCA)*. 532–547. doi:10.1109/ISCA59077.2024.00046

[70] Tong Zhou, Ruiqin Tian, Rizwan A Ashraf, Roberto Gioiosa, Gokcen Kestor, and Vivek Sarkar. 2022. ReACT: Redundancy-aware code generation for tensor expressions. In *Proceedings of the International Conference on Parallel Architectures and Compilation Techniques*. 1–13.

A Appendix

A.1 Intra-expression Iteration Fusion Details

Although we provide a diagram of intra-expression iteration fusion in Figure 3 for dataflow, we also want to tie it to imperative loops for better understanding. We provide an example IIF fusion transformation for dense loops in Figure 18. The transformation for sparse loops is similar but includes coiteration of sparse tensors and iteration of compressed tensor reference arrays.

```
for(int i = 0; i < I; i++)
  a[i] = ...
for(int i = 0; i < I; i++)
  b[i] = ...
```

(a) Unfused intra-expression iteration dense loops.

```
for(int i = 0; i < I; i++)
  a[i] = ...
  b[i] = ...
```

(b) Fused intra-expression iteration dense loops.

Figure 18. Code demonstrating IIF on dense iteration spaces for dense compilers. Figure 18a unfuses the dense iteration space for vectors a and b, while Figure 18b fuses the dense iteration space for a and b. This transformation is often equivalent to loop fusion on dense loops.

A.2 Full Fusion Table for GraphSAGE Example

Figure 19 shows the full fusion table for the GraphSAGE fused kernel.

A.3 Datasets

Table 4 shows a description of the datasets used to evaluate FuseFlow in Section 8.

Model	Benchmark/Dataset	MxN	Sparsity
GCN/GraphSAGE	Planetoid/Cora [64]	2708 x 1433	99.7%
GCN/GraphSAGE	CitationFull/Cora_ML [6]	2995 x 2879	99.8%
GCN/GraphSAGE	CitationFull/DBLP [6]	17716 x 1639	99.6%
GCN/GraphSAGE	OGB-Collab [29]	235868 x 128	99.9%
GCN/GraphSAGE	OGB-MAG [29]	1939743 x 128	99.9%
SAE	ImageNet [14]	224 x 224	50%
SAE	NIH-CXR [57]	1024 x 1024	50%
SAE	LUNA16 [53]	512 x 512	50%
GPT-3 w/ BigBird	IMDB [6]		53.9%-86.5%*

Table 4. Datasets used for each model class with their corresponding sparsity levels. *Refers to attention mask sparsity.

A.4 Lowering Algorithm Implications

Our proposed lowering method produces dataflow graphs with computations, along with their reductions, placed in their natural positions rather than deferring them to the end. In particular, The FuseFlow compiler generates factored iteration because it does not distribute multiplications across sums and does not construct a fully fused global iteration space. For our GraphSAGE example in Figure 9, we use background color shading to help visualize this placement and distinguish interleaved regions: blue shading highlights input iteration regions, while yellow shading highlights computation regions. Concretely, higher-order reducer primitives spatially appear earlier in the graph and generate coordinate streams that flow to stream joiners later in the graph. An abstracted version of this interleaving is shown in Figure 10 (right) with behavior equivalent to the factored fusion iteration space from Figure 5b. On the other hand, Figure 10 (left) shows the generated SAM graphs from prior work [28] with its behavior equivalent to the global iteration space in Figure 5a. In this case, all computation is combined at the end, rather than interleaved. Figure 10 demonstrates how our sparse abstract machine dataflow graphs changed given the new lowering algorithm presented in this section. Concretely, for our GraphSAGE example, the SAM graph with global iteration space is shown in Figure 20a, constrasting with the SAMML graph with factored iteration space as shown in Figure 20b.

A.5 Full Cross-Expression Fusion Algorithm

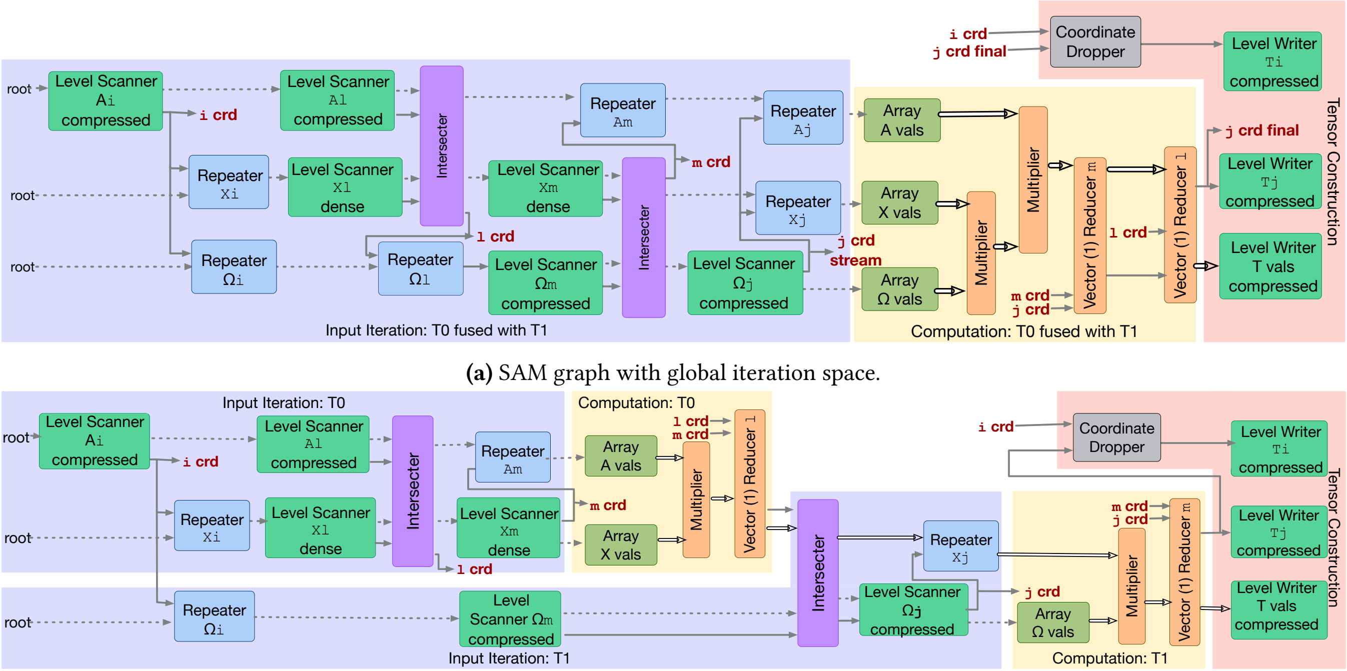
We present the cross-expression fusion algorithm as described in Section 5 below in Algorithm 1.

A.6 Full Fusion Table Lowering Algorithm

We present the full fusion table lowering algorithm as described in Section 6.1 below in Algorithm 2.

	$\hat{A}_{il}$	X_{lm}	T^0_{im}	Ω^2_{mj}	T^{nbor}_{ij}
i	LS(root)	Rep(root, $\langle \hat{A}_i \rangle$)	$\langle \hat{A}_i \rangle$	Rep(root, $\langle T^0_i \rangle$)	T^0_i
l	LS($\langle \hat{A}_i \rangle$) Intersect$_l$	LS($\langle X_i \rangle$)	$\langle T^0_i \rangle$	$\langle \Omega^{(2)}_i \rangle$	$\langle T^{nbor}_i \rangle$
m	Rep($\langle \hat{A}_l \rangle$,$\langle X_m \rangle$)	LS($\langle X_l \rangle$)	Red1$_l$ Intersect$_m$	LS($\langle \Omega^{(2)}_l \rangle$)	$\langle T^{nbor}_l \rangle$
j	$\langle \hat{A}_m \rangle$	$\langle X_m \rangle$	$\langle T^0_m \rangle$[crd$_0$]	LS($\langle \Omega^{(2)}_m \rangle$)	Red1$_m$[crd$_0$]
val	Val($\langle \hat{A}_j \rangle$)	Val($\langle X_j \rangle$)	Rep($\langle T^0_m \rangle$[val], $\langle \Omega^{(2)}_j \rangle$)	Val($\langle \Omega^{(2)}_j \rangle$)	Red1$_m$[val]

Figure 19. Fusion table for our T^{nbor} example from GraphSAGE where Red1$_l$[crd0, val] = $\sum_l \langle \hat{A}_{val} \rangle \times \langle X_{val} \rangle$ and Red1$_m$[crd$_0$, val] = $\sum_m \langle T^0_{val} \rangle \times \langle \Omega^{(2)}_{val} \rangle$.



(a) SAM graph with global iteration space.

(b) SAMML graph with factored iteration space.

Figure 20. SAM graph with global iteration space vs. SAMML graph with factored iteration space.

Algorithm 1 Cross-Expression Fusion with Ordering-Constraints

```
Require: List of kernel expressions $\mathcal{E} = \{e_1, \ldots, e_m\}$ in program order
Ensure: Fused Einsum expressions $\mathcal{F}$ and a global partial-order graph $P = (V, E)$
procedure FuseExpressions($\mathcal{E}$)                         ▷ Init partial order graph
    $P \leftarrow$ InitPOG                                        ▷ nodes = index variables
    $\mathcal{F} \leftarrow [\ ]$                                 ▷ accumulates fused kernels
    for all expression $e \in \mathcal{E}$ do
        for all tensor $T$ in $e$ do
            for all reduction indices $r$ of $T$ do
                $u \leftarrow$ getFreshIndexVar()
                $e \leftarrow e[r \leftarrow u]$                   ▷ substitution
            $E \leftarrow E \cup$ ModeOrderEdges($T$)              ▷ add $(\cdot \rightarrow \cdot)$ edges for $T$'s format
        InlineUses($e, \mathcal{F}$)                              ▷ replace all uses of $e$'s outputs
        $order \leftarrow$ DataflowOrder($e$)                      ▷ e.g. $j \rightarrow k \rightarrow i$
        for all outer → inner in $order$ do
            $E \leftarrow E \cup \{(outer, inner)\}$
        for all tensor uses $U$ in $e$ grouped by original tensor name do
            if CompatibleViews($U$) then
                MergeViews($U$)
            else
                TagDuplicate($U$)
    if CycleDetected($P$) then
        ResolveCycles((P))                                        ▷ insert permutations on offending views
    $\pi \leftarrow$ TopologicalSort($P$)                         ▷ global concordant order
    $\mathcal{F} \leftarrow$ EmitEinsum($\pi$)                    ▷ respecting $P$ and tensor views
    return $\mathcal{F}, P$
```

Algorithm 2 Lowering a Tensor Computation Graph with Fusion Tables

```
Require: Tensor-IR graph $G = (V, E)$
Ensure: Staged, hardware-ready stream graph $G'$
procedure LowerGraph($G$)
    $G' \leftarrow$ InitGraph
    for all tensor-views $T$ in $V$ (top-down) do                ▷ 1) Insert level scanners & value nodes
        if $T$ is InputTensor then
            InsertLSAndVal($T, G'$)
                                                                 ▷ 2) Insert repeat and compute nodes
        if $T$ is IntermediateTensor then
            $missing \leftarrow$ indices absent from $T$
            for all $i \in missing$ do
                InsertRep($T, i, G'$)
            InsertComputePipeline($T, G'$)
                                  ▷ 3) Handle higher-order reductions (modifies table by moving cells)
        if HasHigherOrderReduction($T$) then
            LowerReduction($T, G'$)
                                  ▷ 4) Stream-level merging across views (modifies table by moving cells)
    for all index-vars $i$ shared by $> 1$ view do
        MergeStreams($i, G'$)                                    ▷ intersect / union
                                                                 ▷ 5) Emit lambda table of cell evaluators
    $\Lambda \leftarrow$ EmitFusionTable($G'$)                    ▷ $\Lambda$: cell $\mapsto \lambda$
                                                                 ▷ 6) Trigger graph construction via output view
    $T_{\text{out}} \leftarrow$ OutputTensor($G$)
    Evaluate($\Lambda[T_{\text{out}}]$)
    return $G'$
```