Fast Autoscheduling for Sparse ML Frameworks

Bobby Yan
Stanford University
Stanford, USA
bobbyy@cs.stanford.edu

Alexander J Root
Stanford University
Stanford, USA
ajroot@cs.stanford.edu

Trevor Gale
Stanford University
Stanford, USA
tgale@cs.stanford.edu

David Broman
KTH
Stockholm, Sweden
dbro@kth.se

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

***Abstract*—The rapid growth in the size of deep learning models strains the capabilities of dense computation paradigms. Leveraging sparse computation has become increasingly popular for training and deploying large-scale models, but existing deep learning frameworks lack extensive support for sparse operations. Current approaches either require manual scheduling expertise or rely on exhaustive search taking hours to days, which are both incompatible with the interactive development essential to machine learning research. We present three algorithmic contributions that enable fast, automatic optimization of sparse tensor computations. First, we develop a heuristic-based loop ordering algorithm that avoids asymptotic performance cliffs while compiling in milliseconds rather than hours. Second, we introduce a tiling algorithm specialized for mixed sparse-dense computations that achieves performance comparable to hand-optimized kernels. Third, we present a format inference algorithm that automatically selects appropriate sparse tensor formats for intermediate and output tensors based on operation semantics. These algorithms are grounded in the computational properties of sparse tensor algebra, making them predictable and robust across diverse workloads. We implement these techniques in Scorch, a prototype sparse tensor compiler for PyTorch that demonstrates their practical effectiveness. With only minimal code changes, our approach achieves 1.05–5.80× speedups over PyTorch Sparse on end-to-end tasks including graph neural networks, sparse autoencoders, and sparse transformers, with compilation times fast enough for interactive ML development.**

I. Introduction

The last decade has seen significant advances in code generation approaches for sparse tensor algebra [25, 13, 32, 55]. These code generators allow users to separately specify tensor expressions and the data structures that store the tensors. They then compile the tensor expressions to fused code that iterates over the given data structures to compute the results. Researchers have also proposed sparse scheduling languages [26, 47, 54, 7, 57] that let users manually control how to optimize the generated code, through, e.g., loop ordering, tiling, and the introduction of temporary data structures.

However, we lack *fast* algorithms that automatically make decisions about how to schedule the code and what data structures to use for intermediate and output tensors. Without such optimization machinery, application developers cannot use sparse tensor algebra without also understanding how to optimize it, which limits its use to a small group of experts.

The absence of fast automatic optimization prevents us from extending ML frameworks like PyTorch and JAX with first-class sparse tensor support. These frameworks currently provide only a small set of hand-written sparse matrix operations wrapped in custom APIs, which is sufficient for some popular models but inadequate for exploring new models. A primary reason for the success of PyTorch is its focus on the needs of ML researchers, with support for immediate execution and predictable performance that avoids performance cliffs [42]. We need to maintain this contract if we are to provide support for sparsity in a framework like PyTorch without breaking its ease of use. This means users cannot be required to specify the data structure of every intermediate tensor (of which Python code has many), nor be required to provide schedules to optimize kernel code. It also means that, as argued by Bik et al. [8], sparsity should be a property of tensors and that sparse tensor operations should use the same APIs as dense tensor operations. More subtly, compilation has to be fast enough to support interactive model development with immediate execution, which precludes exclusively relying on search-based autoscheduling strategies [3].

The primary remaining challenge in developing a sparse ML framework is thus to make fast automatic decisions about schedules and storage, while avoiding performance cliffs where loop ordering or fusion strategy degrades *asymptotic complexity*. Additionally, achieving competitive performance with hand-written kernels requires tiling strategies to improve cache locality and load balancing in mixed dense-sparse computations.

The two existing autoscheduling approaches have fundamental limitations. Search-based autoscheduling [3] exhaustively explores the schedule space, taking hours to days. Although search-based approaches can yield great performance, they are incompatible with interactive development. Hand-written kernels, as in PyTorch Sparse and specialized libraries like PyG [16] and DGL [51], provide good performance for specific operations. Such performance, however, does not generalize to the diverse sparse computations needed for future ML, leading to performance cliffs for operations outside the optimized set.

Our approach is to develop fast heuristic algorithms that make principled scheduling decisions in milliseconds rather than hours. Instead of exhaustive search or manual optimization, we employ structural cost models that capture asymptotic complexity differences between schedules. The cost models also systematically identify tiling opportunities in mixed sparse-dense operations, and they automatically infer data structures for intermediate and output tensors based on operation semantics and algebraic properties. These algorithms are grounded in the execution properties of sparse tensor algebra, such as the relationship between loop ordering and iteration space size, making them predictable and robust across diverse workloads. By focusing on avoiding asymptotic performance cliffs and on making reasonably good use of caches, we achieve compilation times proportional to program size while generating code competitive with hand-tuned implementations.

This paper makes three technical contributions:

- *Fast online autoscheduling* (Section III): A heuristic-based algorithm that optimizes loop ordering, fusion, and insertion of temporaries in milliseconds by analyzing sparse filter placement and balancing computational complexity against transposition costs.
- *Mixed sparse-dense tiling* (Section IV): An automatic tiling algorithm for both dense and sparse (data-dependent) loops that achieves performance on par with hand-optimized tiling schemes.
- *Automatic format inference* (Section V): A data-structure inference algorithm for outputs and temporaries based on operator properties and input data structure formats, tailored to choose asymptotically-appropriate formats.

To demonstrate these ideas, we developed Scorch,[1] a prototype of PyTorch with support for sparse tensors that executes on multi-core CPUs. We extend PyTorch's tensor constructors to allow users to specify sparse tensors, and we override PyTorch's APIs to dispatch sparse tensor operations to our compiler. Building on the sparse code generation [25] and optimization [47] techniques from TACO, our compiler automates the choice of schedules and the data structures of intermediate tensors using the algorithms in this paper. We focus on CPUs as a practical and cost-effective platform for sparse workloads [48, 29]. Sparse computations are often memory-bound; CPUs offer larger and cheaper memories than GPUs, along with better support for irregular access patterns. Our evaluation shows that the kernels generated by our system using our autoscheduling techniques are competitive with hand-optimized code for both CPUs and GPUs.

II. The Design of Sparse ML Frameworks

Deep learning frameworks like PyTorch succeed because they optimize first for researcher productivity and pragmatic performance [42], with eager-mode execution and a rich set of operators. A sparse counterpart should preserve these design principles while making sparsity a first-class concern. This section articulates the properties we believe are essential in order for a general, PyTorch-like sparse framework to achieve usability, generality, and performance predictability (depicted in Figure 1). In later sections we present algorithms: fast autoscheduling (Section III), mixed sparse-dense tiling (Section IV), and format inference (Section V), that, when combined with sparse code generation approaches from the literature, operationalize these properties.

A. Design Requirements

The following design points lead to simple and intuitive integration of sparse tensors into modern ML frameworks.

[1]Scorch is available on GitHub at https://github.com/bobbyyyan/scorch.

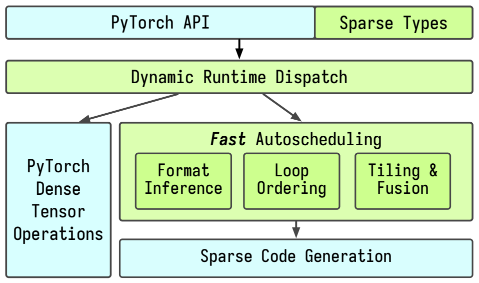


Fig. 1: Overview of a sparse ML framework that highlights the contributions of this paper in green.

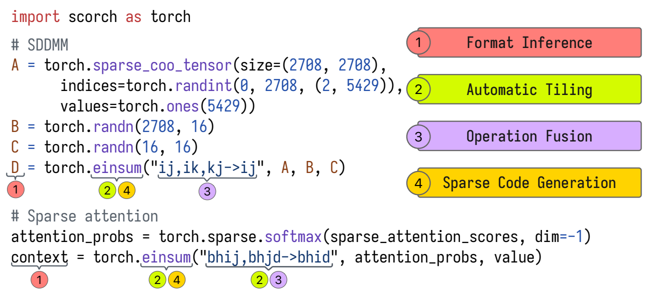
A unified tensor abstraction: A sparse ML framework should provide a unified tensor abstraction that seamlessly handles both dense and sparse data. Users should be able to write operations on any combination of dense and sparse tensors without needing to manually handle different code paths. The framework should automatically handle the complexity of mixed sparse-dense operations internally, allowing researchers to express computations in natural mathematical notation regardless of the underlying data sparsity.

Predictable and automated performance: The framework must automatically make performance-critical decisions without introducing performance cliffs such as the asymptotic slowdowns that can occur in sparse computation. For sparse computations, the framework must make three categories of decisions that fundamentally affect performance in sparse computation. First, loop orders that optimize asymptotic complexity must be chosen. Second, sparse data structure formats for intermediate and output tensors must be automatically chosen based on operation semantics and input formats. Third, memory hierarchy optimizations like tiling must be made to adapt to the irregular access patterns of sparse data. These optimizations should happen transparently during compilation or execution, requiring no manual tuning from users.

Interactive compilation speed: Although it is acceptable to incur high compilation cost before deploying a model, machine learning model development requires rapid iteration and the eager-mode execution of PyTorch is an important reason for its adoption. To facilitate interactive development, a sparse ML framework must support compiling sparse operations in milliseconds, rather than minutes and hours. This constraint precludes exhaustive search methods that explore millions of candidates for all but the last compilation stage before deployment. Instead, during development, the framework needs fast heuristic algorithms that make good decisions quickly, even if they occasionally miss the globally optimal configuration.

B. Programming Model Requirements

Minimal API changes: Adoption of sparse computation should not require learning a new programming paradigm. The framework should extend existing dense tensor APIs rather than replacing them, allowing users to leverage their existing knowledge and gradually adopt sparse features. Converting dense code to sparse should require only minimal changes—perhaps just specifying which initial tensors are sparse.



```
import scorch as torch

# SDDMM
A = torch.sparse_coo_tensor(size=(2708, 2708),
      indices=torch.randint(0, 2708, (2, 5429)),
      values=torch.ones(5429))
B = torch.randn(2708, 16)
C = torch.randn(16, 16)
D = torch.einsum("ij,ik,kj->ij", A, B, C)

# Sparse attention
attention_probs = torch.sparse.softmax(sparse_attention_scores, dim=-1)
context = torch.einsum("bhij,bhjd->bhid", attention_probs, value)
```

Fig. 2: Scorch SDDMM and sparse attention examples.

Automatic format management: Users should not need to specify storage formats for every tensor. While it may be reasonable to specify formats for input tensors (to match data from external sources), the framework should automatically infer appropriate formats for all intermediate and output tensors. This inference should consider both the operation being performed and the formats of input tensors to select formats that balance storage efficiency and computational performance.

Composable operations: The framework should support arbitrary composition and fusion of sparse operations. Users should be able to express complex computations as sequences of simpler operations, and the framework should automatically identify fusion opportunities to eliminate unnecessary materialization of intermediates (Figure 2).

C. Algorithmic Requirements

Asymptotically efficient scheduling: The framework must avoid scheduling decisions that lead to asymptotically inefficient algorithms. For sparse operations, different loop orderings can have vastly different complexities—what takes $O(\text{nnz})$ time in one ordering might take $O(n\cdot\text{nnz})$ in another. The scheduling algorithm must reliably identify and avoid such performance cliffs without exhaustive search.

Mixed sparse-dense optimizations: Real machine learning workloads frequently combine sparse and dense tensors. The framework needs specialized algorithms for optimizing these mixed computations. Dense dimensions within sparse operations present opportunities for cache optimization through tiling, but naive application of dense tiling strategies can degrade performance due to irregular sparse access patterns. The framework needs algorithms that identify which loops can be effectively tiled without introducing excessive overhead.

III. Automatic Loop Ordering and Workspace Insertion

Loop ordering can fundamentally change the worst-case asymptotic complexity and thus the observed performance of sparse computations [26]. We introduce a novel algorithm for automatically determining efficient loop orders that, unlike prior work, does not require an offline exhaustive search [3] or manual scheduling directives [47, 55].

Consider sparse-sparse matrix multiplication (SpMSpM), $C_{ik} = \sum_j A_{ij}B_{jk}$, where inputs A and B, and the output C are stored in the CSR (Compressed Sparse Row) format.

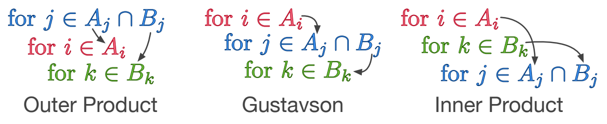


Fig. 3: SpMSpM loop orders.

Figure 3 illustrates three possible loop orderings for SpMSpM: outer product, Gustavson, and inner product. T_i denotes the set of coordinates in the dimension of tensor T indexed by index variable i. The loops iterate over coordinates of different matrix dimensions, and the arrows indicate the data structure traversal orders for each matrix. The inner product algorithm has worse asymptotic complexity than the other two algorithms because neither A nor B's data structures connect the coordinates in the two outer loops, forcing dense iteration. This example highlights three key insights that drive efficient loop ordering for sparse tensor operations:

First, placing sparse filters (Section III-B) earlier in the loop nesting reduces the total iteration space, similar to predicate pushdown optimizations in database query optimization. This ensures that sparse data structures effectively prune the iteration space as early as possible, avoiding unnecessary computation.

Second, intermediate workspaces are essential when scattering into sparse output tensors, particularly for higher-order tensors where dense alternatives would require prohibitive amounts of memory. For operations on real-world sparse tensors, where typically less than 1% of values are non-zero, the memory requirement difference between sparse and dense workspaces can be orders of magnitude. Loop ordering decisions directly impact the dimensionality and size requirements of these workspaces.

Third, transpositions of sparse tensors are computationally expensive operations that require constructing new data structures. They incur both time and space overhead proportional to the number of non-zeros. Therefore, an effective algorithm must balance the benefits of early filtering against the costs of tensor transpositions.

A. Algorithm Overview

Our algorithm (Algorithm 1) implements these insights through a cost-based approach that balances computational complexity, workspace overhead, and transposition costs. The algorithm proceeds in three phases: initialization, loop order optimization, and workspace insertion.

In the initialization phase (lines 3–5), the algorithm collects and sorts index variables in the tensor expression by their sparsity level, which we define formally as the expected cardinality reduction achieved by the corresponding filter. Consider SpMM, $C_{ik} = \sum_j A_{ij}B_{jk}$, where A is stored in the CSR format and B and C are dense. Here, $j \in A_j \cap B_j$ is the only sparse filter, as it involves an intersection between the sparse level A_j and the dense level B_j. When sparsity levels are equivalent (as with i and k in this example), ties are resolved by selecting the order that minimizes tensor transpositions, yielding an initial loop order $\mathcal{L} = [j, i, k]$.

The loop order optimization phase (lines 6–10) employs a greedy approach that evaluates the cost of pushing each sparse filter to different positions in the loop nest. For each index variable i, we consider moving it from its current position to each subsequent position in the loop order. The cost function $\textsc{CostToPush}(\mathcal{L}, i, \mathit{pos})$ is defined as $\alpha\Delta C_{\text{comp}} + \beta\Delta C_{\text{ws}} + \gamma\Delta C_{\text{trans}}$ where ΔC_{comp} represents the change in computational complexity, ΔC_{ws} accounts for workspace overhead, and ΔC_{trans} captures the cost of any required tensor transpositions.

Algorithm 1 Loop Ordering and Workspace Insertion

```
1: Input: Tensor expression $E$, input tensors $\mathcal{T}$, output tensor $T_{out}$.
2: Output: Loop order $\mathcal{L}$ for efficient computation.
3: $\mathcal{I} \leftarrow \textsc{GetIndexVariables}(E)$
4: $\mathcal{S} \leftarrow \textsc{SortBySparsityDescending}(\mathcal{I}, E)$
5: $\mathcal{L} \leftarrow \textsc{InitLoopOrder}(\mathcal{S})$
6: for all $i \in \mathcal{S}$ do                                  ▷ Loop ordering
7:     $\mathit{currentPos} \leftarrow \textsc{GetPosition}(\mathcal{L}, i)$
8:     for $\mathit{pos} \leftarrow \mathit{currentPos}$ to $|\mathcal{L}| + 1$ do
9:         if $\textsc{CostToPush}(\mathcal{L}, i, \mathit{pos}) < 0$ then
10:            $\mathcal{L} \leftarrow \textsc{MoveToPosition}(\mathcal{L}, i, \mathit{pos})$
11: $G \leftarrow \textsc{InitGraph}(\mathcal{I})$              ▷ Directed graph with nodes $\mathcal{I}$
12: for all $T \in \mathcal{T} \cup \{T_{out}\}$ do
13:     $\pi \leftarrow \textsc{GetModeOrdering}(T)$
14:     for $k \leftarrow 1$ to $|\mathcal{I}(T)| - 1$ do          ▷ Add order constraints
15:         $\textsc{AddEdgeToGraph}(G, \pi[k], \pi[k+1])$
16: while $\textsc{ContainsCycles}(G)$ do
17:     $e \leftarrow \textsc{RemoveCheapestEdge}(G)$
18:     $\mathcal{L} \leftarrow \textsc{UpdateLoopOrder}(\mathcal{L}, e)$    ▷ Update the loop order by transposing the tensor
19: if $\textsc{HasSparseLevels}(T_{out})$ then                  ▷ Workspace insertion
20:     $\mathcal{I}_{red} \leftarrow \textsc{GetReductionVariables}(\mathcal{T}, T_{out})$
21:     if $\textsc{ShouldInsertWSpace}(\mathcal{I}_{red}, \mathcal{I}(T_{out}), \mathcal{L})$ then
22:         $W \leftarrow \textsc{InsertWorkspace}(\mathcal{L}, \mathcal{I}_{red})$
23:         $(\mathcal{L}_p, \mathcal{L}_c) \leftarrow \textsc{SplitLoopOrder}(\mathcal{L}, \mathcal{I}_{red})$
24:         $\mathcal{L}_p \leftarrow \textsc{UpdateProducerLoop}(\mathcal{L}_p, W, E)$
25:         $\mathcal{L}_c \leftarrow \textsc{UpdateConsumerLoop}(\mathcal{L}_c, T_{out}, W)$
26: return $\mathcal{L}$
```

For SpMSpM, when evaluating the net cost of moving j to position 1, we see a bigger ΔC_{comp} due to the filter no longer applying to rows of A in the i loop, a smaller ΔC_{ws} from the 2D workspace being reduced to a 1D workspace (no longer scattering into the i dimension), and a smaller ΔC_{trans} as A no longer requires transposition to be iterated in j, i order. The negative net cost means the loop order gets updated to $[i, j, k]$. Conversely, moving j to position 2 would yield the inner product algorithm $[i, k, j]$, where it incurs a positive net cost due to a much bigger ΔC_{comp} as no input data structures allow sparse iteration from i to k and ΔC_{trans} from transposing B to iterate in k, j order, more so than the savings in ΔC_{ws} from eliminating the need for temporaries.

The mode ordering constraints from tensor storage formats are captured in a directed graph (lines 11–15), with cycles indicating required transpositions (handled in lines 16–18). Finally, the workspace insertion phase (lines 19–25) determines whether intermediate workspace tensors are needed to efficiently scatter into sparse output formats.

In the following subsections, we provide detailed analyses of each component in this algorithm. Section III-B formalizes the identification and ranking of sparse filters that guide loop ordering decisions. Section III-C presents our cost model for evaluating candidate loop orders, which balances the computational, workspace, and transposition costs.

B. Sparse Filters

While concepts similar to sparse filters have been used in prior work on sparse tensor algebra [25], our approach introduces a systematic method for comparing filter sparsity across complex expressions. This subsection explains our approach to identifying and ranking sparse filters for loop ordering decisions.

Given a tensor expression, each index variable corresponds to a set expression involving the coordinates of each tensor. For example, in the matrix multiplication expression $C_{ik} = \sum_j A_{ij} B_{jk}$, the index variables i, j, and k correspond to the set expressions A_i, $A_j \cap B_j$, and B_k, where T_i denotes the set of coordinates in the dimension of tensor T indexed by index variable i.

A sparse filter is an index variable whose set expression involves the intersection of a sparse set with a dense set (or the universe). In the context of sparse-dense matrix multiplication (SpMM), where A is a sparse matrix in the CSR format and B is a dense matrix, A_i is a dense set, A_j is a sparse set, and both B_j and B_k are dense sets. Therefore, index variables i and k are not sparse filters as they each correspond to a dense set (A_i and B_k respectively), while index variable j is a sparse filter because its set expression ($A_j \cap B_j$) involves the intersection of a sparse set (A_j) with a dense set (B_j), resulting in a sparse set.

In the case of sparse-sparse matrix multiplication (SpMSpM), where both A and B are sparse matrices in the CSR format, both j and k are sparse filters. To determine the relative sparsity of j and k, we assume that each sparse level is equally sparse without inspecting the actual data. Thus, A_j and B_k are considered equally sparse. The set expression for j evaluates to $A_j \cap B_j$, which simplifies to A_j (a sparse set), while the set expression for k evaluates to B_k (a sparse set). As a result, there is a tie in the sparsity of j and k.

In complex fused expressions involving more than two tensors, the sparsity comparisons of index variables may involve unions and intersections of multiple sparse sets. For instance, consider the set expressions $(A_i \cap B_i) \cup C_i$ and $(A_j \cup B_j) \cap C_j$ for index variables i and j, where each of the sets A_i, B_i, C_i, A_j, B_j, and C_j is sparse. To compare the sparsity of i and j, we analyze and compare the regions in the Venn diagrams for the set expressions corresponding to each index variable, as shown in Figure 4.

The set expression for i is the union of a sparse set with the intersection of two sparse sets, while the set expression for j is the intersection of the union of two sparse sets with another

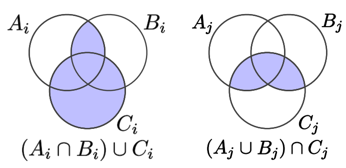


Fig. 4: Sparse filters.

sparse set. Since the former is a superset of the latter, we can conclude that j is more sparse than i.

C. Cost Modeling for Loop Ordering Selection

The autoscheduling algorithm in Scorch uses a structural cost model to evaluate candidate loop orders. For a loop order $\mathcal{L}$ and a proposal to move index variable i to position *pos*, the cost function $\textsc{CostToPush}(\mathcal{L}, i, pos)$ computes $\textsc{CostToPush}(\mathcal{L}, i, pos) = \alpha \Delta C_{\text{comp}} + \beta \Delta C_{\text{ws}} + \gamma \Delta C_{\text{trans}}$, where $\Delta C_{\text{comp}} = C_{\text{comp}}(\mathcal{L}') - C_{\text{comp}}(\mathcal{L})$, $\Delta C_{\text{ws}} = C_{\text{ws}}(\mathcal{L}') - C_{\text{ws}}(\mathcal{L})$, $\Delta C_{\text{trans}} = C_{\text{trans}}(\mathcal{L}') - C_{\text{trans}}(\mathcal{L})$ and ΔC_{comp}, ΔC_{ws}, and ΔC_{trans} represent the changes in the computational, workspace, and transposition costs, respectively, when moving from loop order $\mathcal{L}$ to a new order $\mathcal{L}'$ with index i at position pos.

This model aims to capture the asymptotic complexity differences between loop orderings rather than predicting exact runtimes. While this approach enables fast scheduling decisions without runtime profiling, it can produce suboptimal schedules for tensors whose actual sparsity patterns significantly diverge from our structural assumptions—for example, when a dense tensor is stored in a sparse format or when the distribution of non-zeros is highly skewed. In practice, we find that these edge cases are rare in real-world workloads, as demonstrated in our evaluation.

a) Computational Cost, C_{comp}: The computational cost $C_{\text{comp}}(\mathcal{L})$ estimates the total number of operations required to execute the loop nest for loop order $\mathcal{L}$. This cost depends on the size of iteration space at each loop level, which is affected by the position of sparse filters.

Let $\mathcal{L} = [l_1, l_2, \ldots, l_n]$ be the loop order. The total computational cost is modeled as $C_{\text{comp}}(\mathcal{L}) = \prod_{k=1}^{n} |\mathcal{I}_{l_k}|$, where $|\mathcal{I}_{l_k}|$ is the effective iteration count of loop l_k, adjusted for the filtering effects of prior loops. To accurately capture the effect of early or late filtering, we model $|\mathcal{I}_{l_k}|$ by considering the filters applied before loop l_k and how they reduce the iteration space at level l_k.

b) Modeling Effective Iteration Counts: For each loop l_k, the effective iteration count $|\mathcal{I}_{l_k}|$ is determined by $|\mathcal{I}_{l_k}| = \left|\bigcap_{T \in \mathcal{T}_{l_k}} T_{l_k}(\mathbf{c}_T^{<k})\right|$, where $\mathcal{T}_{l_k}$ is the set of tensors that involve index l_k, and $T_{l_k}(\mathbf{c}_T^{<k})$ is the set of feasible indices of T at level l_k given the previous indices $\mathbf{c}_T^{<k}$ chosen for T in the parent loops before l_k. Since we cannot examine actual tensor values during compilation, we estimate these cardinalities using only structural information. For sparse dimensions, we use a fixed sparsity ratio parameter ρ to approximate the expected number of non-zeros per dimension.

c) Workspace Cost, C_{ws}: The workspace cost accounts for the time and space complexity of using an intermediate workspace: $C_{\text{ws}}(\mathcal{L}) = c_{\text{insert}} N_{\text{insert}} D + c_{\text{sort}} N_{\text{entries}} D \log N_{\text{entries}}$, where c_{insert} and c_{sort} are constants representing the cost per insertion and per comparison during sorting, respectively. N_{insert} is the total number of insertions into the workspace. N_{entries} is the number of unique entries in the workspace. D is the dimensionality of the workspace. The insertion cost reflects the overhead of managing the workspace's hash map and coordinate arrays, while the sorting cost accounts for ordering the entries before writing back to the output tensor.

d) Transposition Cost, C_{trans}: The transposition cost represents the overhead of transposing tensors to match the loop order, which is proportional to the number of non-zero elements in the tensor [39]: $C_{\text{trans}}(\mathcal{L}) = \sum_{T \in \mathcal{T}} c_{\text{trans}} \hat{\text{nnz}}_T \delta(T, \mathcal{L})$, where c_{trans} is the constant cost per non-zero element for transposition, $\hat{\text{nnz}}_T$ is the estimated number of non-zeros in T, and $\delta(T, \mathcal{L}) = 1$ if T needs to be transposed to align with $\mathcal{L}$ and 0 otherwise.

Since we cannot inspect the actual data values during compilation, we estimate $n\hat{n}z_T$ based on tensor dimensions and storage format. For a tensor with dimensions $[d_1, d_2, ..., d_n]$, we approximate $n\hat{n}z_T \approx \prod_{i=1}^{n} \rho_i \cdot d_i$, where $\rho_i = 1$ for dense dimensions and $\rho_i = \rho$ for sparse dimensions.

e) Constant Selection via Autotuning: The constants in our cost model (α, β, γ, c_{insert}, c_{sort}, c_{trans}, and ρ) are determined through autotuning. We use a representative subset of matrices from the SuiteSparse Matrix Collection [14], sampling across different matrix sizes, domains, and sparsity patterns. For each core sparse kernel (SpMV, SpMM, SpMSpM, and SDDMM), we generate multiple valid schedules, measure their actual performance, and apply Bayesian optimization to identify constant values that predict the optimal schedule. The constants were tuned once on the ARM machine and used unchanged on x86. This cross-architecture portability stems from the cost model's focus on fundamental algorithmic costs (computational work, memory allocation, data layout transformations) rather than microarchitectural details.

IV. Tiling Mixed Dense-Sparse Loops

Tiling, or partitioning iteration spaces into smaller blocks that fit in faster memory, is a fundamental optimization that improves cache utilization for mixed sparse-dense operations, as it does for dense computations. For dense computations, tiling techniques are well-understood and have been studied extensively [53, 11, 43, 2, 60, 61]. However, mixed sparse-dense computations present unique challenges that make tiling decisions more complex. In these computations, the irregular access patterns from sparse dimensions interact with regular access patterns from dense dimensions, creating tension between different optimization goals. Dense dimensions benefit from tiling to improve cache locality, but the optimal tiling strategy depends on how they interact with sparse dimensions. Naively applying dense tiling strategies to mixed sparse-dense computations can actually make performance worse.

While prior work like TACO [25] established foundations for iteration space transformations and SparseTIR [55] introduced composable formats, neither system addressed the challenge of automatically determining which loops to tile in mixed sparse-dense computations. Pigeon [3] makes loop ordering but not tiling decisions. Our approach introduces a systematic method for identifying tiling opportunities that accounts for unique characteristics of mixed sparse-dense loop nests to provide a principled solution to a problem that previous systems left to manual tuning or operation-specific heuristics.

Our tiling strategy is based on three key observations derived from analyzing the behavior of sparse tensor operations. First, we note that reuse opportunities can be identified by examining the index variables present (and more importantly, missing) in each tensor access. Second, we recognize that tiling sparse dimensions often degrades performance due to the overhead of traversing sparse data structures in order, which requires performing expensive searches. Third, we observe that tiling some dense dimensions can be counterproductive, such as those that are parents of a sparse dimension in the loop nest.

Algorithm 2 Tiling Sparse Tensor Operations

```
1:  **Input:** A tensor expression $E$ containing input tensors $\mathcal{T}$, and a loop structure $\mathcal{L}$.
2:  **Output:** Tiled loop structure $\mathcal{L}_{\text{tiled}}$.
3:  $\mathcal{W} \leftarrow \emptyset$                      ▷ Initialize the working set of index variables
4:  $\mathcal{S} \leftarrow$ GETALLINDEXVARIABLES($E$)
5:  for each tensor access $T[\mathcal{I}]$ in the expression $E$ do
6:      if $\mathcal{I} \subset \mathcal{S}$ then                ▷ $\mathcal{I}$ are the indices of $T$
7:          $\mathcal{W} \leftarrow \mathcal{W} \cup \mathcal{I}$           ▷ Add indices to the working set
8:  for all $i \in \mathcal{W}$ do                  ▷ Remove sparse index variables
9:      if $\exists\, T \in \mathcal{T} \mid T_i$ is sparse then
10:         $\mathcal{W} \leftarrow \mathcal{W} \setminus \{i\}$
11: for all $i \in \mathcal{W}$ do         ▷ Remove parents of sparse dimensions
12:     if $\exists\, j \mid j$ is sparse and $i =$ PARENT($j$) in $\mathcal{L}$ then
13:         $\mathcal{W} \leftarrow \mathcal{W} \setminus \{i\}$
14: for all $i \in \mathcal{W}$ do                                    ▷ Tiling
15:     $(i_{\text{outer}}, i_{\text{inner}}) \leftarrow$ TILEINDEX($i$)                 ▷ Split loop $i$
16:     $\mathcal{L} \leftarrow$ REORDERLOOPS($\mathcal{L}, i_{\text{outer}}$)      ▷ Move $i_{\text{outer}}$ outermost
17: return $\mathcal{L}_{\text{tiled}} \leftarrow \mathcal{L}$
```

Consider the SpMM expression $C_{ik} = \sum_j A_{ij} B_{jk}$ with A as a CSR matrix and B, C as dense matrices. When a tensor access is missing an index variable present in the full expression, that tensor is reused across the loop corresponding to the missing index variable. Examining the tensor accesses of SpMM reveals that C_{ik} is missing the index variable j, indicating reuse across the j loop. This suggests potential benefits from tiling the i and k loops if they were both dense. However, tiling both i and k dimensions leads to a larger working set (area covered by the arrows in Figure 5) for B, as the tiles of k are exhausted more frequently. If all the dimensions were dense, loop j can be tiled to mitigate this issue, but j indexes a sparse dimension of A and cannot be efficiently tiled as a result. Tiling i, which is a parent of the sparse dimension j, should be avoided because without tiling j, we cannot limit B's working set between i tiles. In dense computation, tiling all three dimensions creates bounded working sets, but in mixed sparse-dense computation, the inability to tile the sparse j dimension means that tiling i

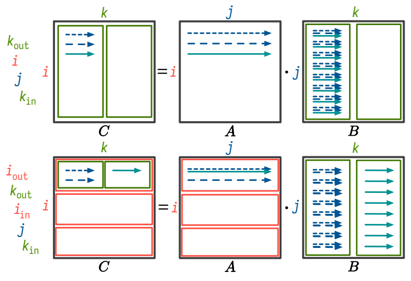


Fig. 5: Tiling SpMM ($C_{ik} = \sum_j A_{ij} B_{jk}$)

multiplies the number of times we must load each k-tile of B—once for every i-tile—destroying the cache efficiency that tiling is meant to provide. Our evaluation results in Figure 7b provide empirical validation of this insight and show that tiling both i and k dimensions (i-k-Tiled) degrades performance compared to only tiling the k dimension (k-Tiled). These observations are formalized in our tiling algorithm (Algorithm 2), which systematically identifies potential loops for tiling by analyzing tensor access patterns, excludes loops corresponding to sparse dimensions and loops that are parents of sparse dimensions, and applies tiling transformation to the remaining candidate loops. This algorithm automates a decision process that previously required extensive expertise in both sparse tensor algebra and cache optimization, making efficient sparse computation more accessible to general users of machine learning frameworks.

V. Format Inference

A key challenge when working with sparse tensor algebra operations is determining the data structures of different tensors. In prior work on sparse tensor algebra compilation [25, 4, 55] the user of the system has to specify the data structures (formats) of every tensor. We believe it is reasonable and even desirable to ask the users to specify the data structures of initial tensors, so that they can control what tensors are sparse and make sure the data structures going into a model matches the data structures produced by surrounding libraries outside of the control of our system. However, requiring a user to specify the data structures of intermediate (temporary) tensors is an unreasonable burden and would riddle the code with data structure definitions.

We address this challenge through a novel automatic tensor format inference algorithm, which to the best of our knowledge, makes Scorch the first sparse tensor compiler to automatically determine output formats.

A. *Format Inference Overview*

Format inference determines the storage format of each dimension of the output tensor based on the storage formats of the input tensors and the semantics of the tensor expression. A key property of the inference is that it operates on a per-dimension basis, analyzing how operations affect the sparsity patterns of each dimension of the output independently.

This dimension-wise approach is essential because real-world tensors frequently exhibit non-uniform sparsity patterns. Many common tensor formats like CSR have different storage strategies for different tensor dimensions—using dense storage for rows as they all have some values and compressed storage for columns as each row only has a few values. By analyzing each dimension independently, the inference system can select the most appropriate format for each dimension, enabling mixed-format tensors that balance storage efficiency and computational performance.

We classify tensor dimension formats into two categories: DENSE and SPARSE, the latter of which includes both *compressed* and *coordinate* level formats. Our inference algorithm uses a pattern-matching approach that recursively analyzes tensor expressions to determine the appropriate output format.

B. Inference Algorithm

The inference algorithm follows a set of rules that reflect the mathematical properties of operations on tensor sparsity:

- *For additions ($A + B$)*: If either operand is dense in a dimension, the result will be dense in that dimension since the non-zeros from the dense operand will fill the zero positions in the sparse operand. If both operands are sparse, we conservatively choose a sparse representation, as this avoids the memory and computation cost of unnecessarily using a dense format.
- *For multiplications ($A * B$)*: If either operand is sparse in a dimension, the result will be at least as sparse in that dimension, as multiplication with zero produces zero ($0 \cdot x = 0$). Therefore, we choose a sparse representation. If both operands are dense, the result is dense.
- *For unary operations (op(A))*: We consider the operation's effect on zeros and categorize them into three classes:
  1) SPARSIFYINGOP: Functions that introduce zeros, such as ReLU, step functions, thresholding operations (e.g., threshold$(x, t) = x$ if $|x| > t$ else 0), and sparsity-inducing regularizers.
  2) ZEROPRESERVINGOP: Functions that maintain zeros, such as $\sin$, $\tanh$, scalar multiplication, and similar element-wise operations where $f(0) = 0$.
  3) DENSIFYINGOP: Functions that potentially convert zeros to non-zeros, such as $\cos$, addition with a constant, sigmoid, or offset operations ($x + c$ where $c \neq 0$).
- *For tensor literals (T)*: We extract the format directly from the tensor's definition.

These inference rules are formalized in the recursive algorithm presented in Algorithm 3. The inference function INFER(E, i) determines the format for a tensor expression E and dimension i by traversing the expression tree, handling binary operations (addition, multiplication), unary operations, and tensor literals.

Algorithm 3 Tensor Format Inference

```
1:  procedure INFER($E$, $i$)
2:      match E
3:      | $A + B$ → match INFER($A$, $i$), INFER($B$, $i$) with
4:                  | DENSE, _ → return DENSE
5:                  | _, DENSE → return DENSE
6:                  | _, _ → return SPARSE
7:      | $A * B$ → match INFER($A$, $i$), INFER($B$, $i$) with
8:                  | SPARSE, _ → return SPARSE
9:                  | _, SPARSE → return SPARSE
10:                 | _, _ → return DENSE
11:     | SPARSIFYINGOP($A$) → return SPARSE
12:     | ZEROPRESERVINGOP($A$) → return INFER($A$, $i$)
13:     | DENSIFYINGOP($A$) → return DENSE
14:     | $T$ → return TENSORFORMAT($T$, $i$)
15:     end
```

VI. EVALUATION

We evaluate Scorch to demonstrate how our technical contributions—fast autoscheduling, mixed sparse-dense tiling, and automatic format inference—bridge the gap between sparse tensor compilers and practical machine learning frameworks. Our evaluation demonstrates the following claims:

1) Our autoscheduling and format inference algorithms match or exceed the performance of hand-tuned sparse kernels without requiring manual optimization (Section VI-B).
2) Our mixed sparse-dense tiling algorithm delivers meaningful performance improvements for real-world sparse workloads (Section VI-C).
3) Our contributions enable practical speedups over PyTorch Sparse and specialized graph frameworks on end-to-end machine learning tasks across diverse domains (Section VI-G).

To support these claims, we evaluate Scorch against existing libraries on both core sparse operations and end-to-end models across graph neural networks, sparse autoencoders, and sparse transformers. We select these domains specifically because they represent distinct sparsity patterns and computational requirements, allowing us to validate the generality of our approach. Throughout our evaluation, we use PyTorch Sparse as the primary baseline, with domain-specific libraries as additional comparison points where applicable.

A. Evaluation Environment

All experiments were conducted on two architectures: an Intel Core i9-14900K (3.2 GHz, 24 cores) with 196 GB DDR4 RAM and an Apple M1 Ultra (3.2 GHz, 20 cores) with 64 GB DDR5 RAM. For GPU benchmarks, we used an NVIDIA GeForce RTX 4090 GPU with an Intel i9-14900K CPU. Kernel benchmarks use matrices from the SuiteSparse Matrix Collection [14]. Using two different CPU architectures lets us validate that our autoscheduling approach generalizes beyond a single hardware platform. We used PyTorch's eager (immediate) execution mode for all experiments because graph mode in the current version (`torch.compile` [5]) treats sparse operations as opaque nodes and does not perform cross-operation fusion or optimization.

PyTorch uses a caching allocator to recycle memory and avoid repeated system calls. Scorch lacks a custom allocator: generated kernels invoke `malloc` and `memset` for each output tensor. In Section VI-B, this difference causes slowdowns with respect to PyTorch for small matrix sizes (where the 10–20μs per allocation is a large portion of the 50–100μs runtimes). Such tiny sparse matrices are uncommon in real-world ML workloads—our end-to-end benchmarks in Section VI-G show this allocator overhead has negligible impact on practical workloads. An output-tensor memory pool would amortize such small-matrix overheads.

B. Core Sparse Kernels

To demonstrate the effectiveness of automatic optimization algorithms—loop ordering, tiling, and format inference—we

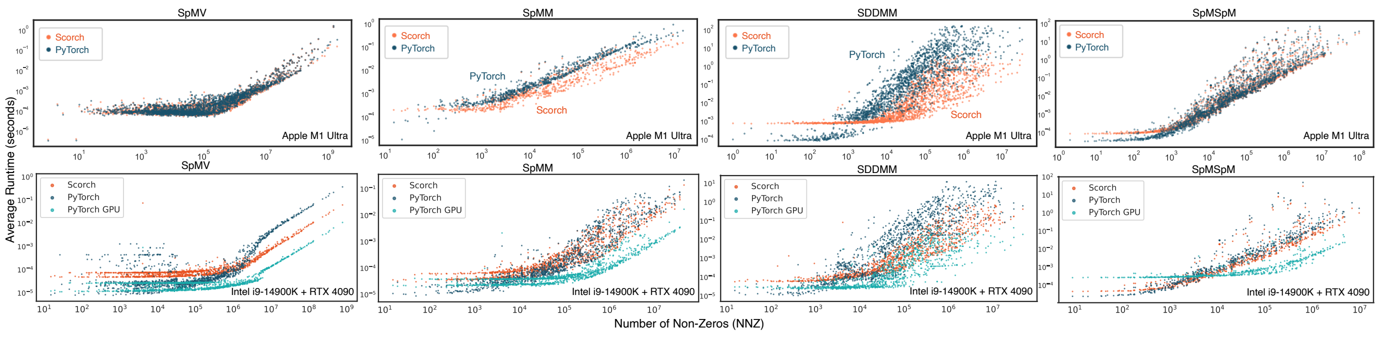


Fig. 6: Performance on core sparse kernels (ARM on top and x86 on bottom).

benchmark four core sparse kernels that form the building blocks of many sparse machine learning models: SpMV, SpMM, SDDMM, and SpMSpM. These kernels represent a range of computational patterns with different optimization challenges, allowing us to assess how our contributions address different sparse data access patterns and computational complexity characteristics. Each operation presents distinct challenges in terms of memory access patterns, fusion and tiling opportunities, and optimal output format selection.

Figure 6 shows the performance of Scorch versus PyTorch (with `torch.sparse`) for the four kernels across a range of problem sizes and sparsity patterns on the ARM architecture. The results demonstrate two key insights about our autoscheduling approach. First, for operations like SpMV and SpMSpM, where optimization opportunities are limited by the inherent computational patterns, Scorch achieves performance parity with PyTorch. This validates that our fast autoscheduling system correctly identifies when simple loop orders are optimal and avoids the complexity and overhead of any unnecessary optimizations to deliver competitive performance.

Second, and more importantly, for operations like SpMM, our heuristic-based autoscheduling efficiently identifies opportunities for tiling, resulting in performance gains that increase as the problem size increases. The results on SDDMM provide compelling evidence for our approach, where Scorch outperforms PyTorch by more than an order of magnitude on larger problems. This substantial improvement stems from fusion. While PyTorch must decompose SDDMM into separate sparse element-wise multiplication and dense matrix multiplication operations, thus materializing large dense intermediate results, Scorch can generate one single fused kernel that co-iterates over the sparse data structures and only computes the necessary dense products. This example highlights how our approach to automating key decisions in sparse computation can yield asymptotic performance improvements and not just constant-factor optimizations.

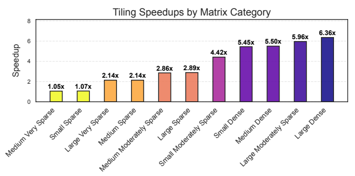
On the x86 architecture (Figure 6, bottom), Scorch achieves performance parity with PyTorch, which leverages Intel's Math Kernel Library (MKL) for sparse computations. This is significant in that it demonstrates that our general-purpose compilation approach with automatic decision-making can match specialized, hand-tuned implementations developed over many years. The ability to automatically match the performance of expert-engineered libraries is precisely the goal of our fast online scheduling system, which is to deliver reasonable performance on a diverse set of kernels without requiring users to have specialized knowledge of sparse computation.

The PyTorch GPU results in Figure 6 (bottom), which use cuSPARSE kernels on the NVIDIA RTX 4090 GPU, serve as a reference point for the potential performance benefits of GPU acceleration. These results illustrate the performance gap between CPU and GPU for sparse operations, underscoring the effectiveness of GPUs for some large-scale sparse computations. Although Scorch does not currently support GPU acceleration, understanding this performance gap informs future work.

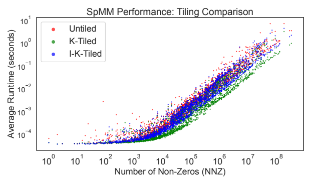
C. Mixed Sparse-Dense Tiling

To evaluate our second contribution, an algorithm for tiling mixed sparse-dense loops, we perform a series of comparisons of kernels with and without tiling. We provide a performance evaluation of the speedup gained by tiling sparse-dense kernels on the following benchmarks:

1) Sparse matrix-matrix multiplication (SpMM) across different matrix categories in Figure 7a. Large matrices benefit the most from tiling with up to 6.36× speedup over untiled implementations, while large moderately sparse matrices achieve 5.96× speedup.
2) Sparse matrix-matrix multiplication (SpMM) across varying numbers of non-zeros in Figure 7b. The tiled implementation consistently outperforms the untiled



(a) Tiling speedups (over untiled)



(b) Tiling ablation

Fig. 7: Comparison of tiling strategies across matrix categories and problem sizes. The left plot shows speedups from tiling the dimension k for SpMM ($C_{ik} = \sum_j A_{ij} B_{jk}$). The right plot compares the performance of untiled, k-Tiled (tiling the k dimension), and i-k-Tiled (tiling i and k dimensions).

implementation across most problem sizes, with the performance advantage becoming more pronounced as the number of non-zeros increases.

Figure 7a shows that SpMM achieves the most substantial performance gains from tiling the k dimension on larger, denser matrices (6.36× speedup) and large moderately sparse matrices (5.96× speedup). These matrices contain sufficient regular computation patterns to amortize the overhead and benefit from improved cache locality, while their untiled counterparts suffer from cache thrashing as problem size increases.

The dimension-wise comparison in Figure 7b demonstrates why selective tiling is crucial. Tiling only the k dimension consistently outperforms both the untiled baseline and more aggressive i-k-Tiled strategy. In the presence of a sparse dimension j, the additional control flow complexity and boundary checking overhead from tiling both dimensions can outweigh cache benefits. While multi-dimensional sparse tiling can be beneficial [28], the choice of not tiling sparse dimensions changes the optimization landscape. Without tiling the sparse dimension j, we cannot bound B's working set between i tiles, causing each i-tile to reload all k-tiles of B, destroying cache efficiency. To avoid the unpredictable overhead of sparse tiling, we provide a specialized solution for mixed sparse-dense loops, rather than adopting techniques from dense linear algebra. For highly sparse matrices, the algorithm achieves modest gains of 1.05–1.07×. These smaller improvements align with our theoretical understanding of these workloads, where irregular access patterns limit the benefit of cache optimizations. These results show that our mixed sparse-dense tiling algorithm can deliver meaningful performance improvements on real-world data while avoiding the pitfalls of over-aggressive tiling.

D. Autoscheduling Times

Table I compares Scorch's autoscheduling performance against Pigeon [3], the only other sparse autoscheduler of which we are aware, on the two benchmarks made available in their artifact. Pigeon is an exhaustive enumerative system: it explores all viable loop orderings and fusion choices, pruning this large space to produce an asymptotically optimal Pareto frontier of schedules. In contrast, Scorch's greedy autoscheduler is several orders of magnitude faster, yet its schedules lie within Pigeon's optimal frontier while additionally performing tiling and format inference decisions beyond Pigeon's capabilities. Table I also shows that Scorch's autoscheduling now has similar cost to the other lowering stages (e.g., CIN lowering), accounting for roughly 33% of end-to-end lowering time across the core sparse kernels evaluated in Section VI-B.

TABLE I: Autoscheduling and Compilation Times (M1 Ultra)

Kernel	Total	LLIR Lower	CIN Lower	Scorch Autosch.	Pigeon [3] Autosch.	Scorch Speedup
SpMM	2.17 ms	0.31 ms	1.17 ms	0.69 ms	N/A	N/A
SpMV	1.39 ms	0.17 ms	0.72 ms	0.51 ms	6.09 s	11941×
SpMSpM	1.74 ms	0.22 ms	0.96 ms	0.56 ms	2.44 s	4357×
SDDMM	2.22 ms	0.26 ms	1.20 ms	0.75 ms	N/A	N/A

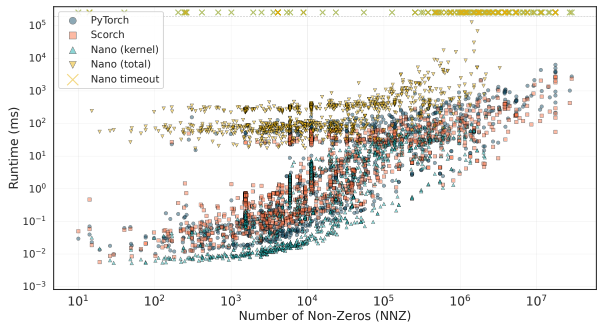


Fig. 8: Performance on SpMM across SuiteSparse matrices, with $k = 512$, on the first 1000 of 2903 matrices in default `ssgetpy` order.

E. Comparison to Offline Tiling Schemes

While our heuristic-based approach prioritizes fast compilation with predictable performance for direct-mode use, offline optimization can explore larger optimization spaces at the cost of runtime preprocessing overhead. To understand this trade-off, we compare Scorch with Nano [52], a register-tiling framework for SpMM that uses offline ILP optimization to pre-compute mappings between sparsity patterns and register-optimized micro-kernels. At runtime, Nano converts matrices from standard sparse formats (COO/CSR) into a specialized format where values are reorganized by sparsity pattern: columns with the same pattern are grouped together, values are extracted and potentially padded to match micro-kernel register layouts, enabling efficient branch-free vectorized execution at the cost of preprocessing each matrix.

For Nano, we report kernel-only execution time as Nano (kernel) and total time including the preprocessing overhead as Nano (total). Figure 8 shows that while Nano (kernel) achieves superior performance through pattern-specialized micro-kernels, the preprocessing overhead makes Nano (total) exceed Scorch's total time. This overhead is incurred for initial compilation (e.g., in direct-mode use) and whenever the matrix changes. Scorch achieves competitive performance while maintaining millisecond-scale compilation (Table I) and working directly with standard sparse formats. Inspector-based approaches, however, offer advantages when matrices are extensively reused, such as during inference, but our approach is appealing in dynamic settings where tensors change frequently.

F. Fusion Ablation

To isolate fusion benefits, we measured performance with fusion disabled (forcing materialization of all intermediates) and found that on average, fusion provides a 47.56× speedup over the unfused implementation on ARM and 45.63× on x86 for the SDDMM benchmark and a 1.95×, 1.18×, and 1.10× speedup over the unfused implementation on the sparse transformer benchmark on ARM for the AG News, IMDB, and Yahoo Answers datasets respectively.

G. End-to-End Benchmarks

While the benchmarks on core sparse kernels and ablation study on tiling validate our individual technical contributions,

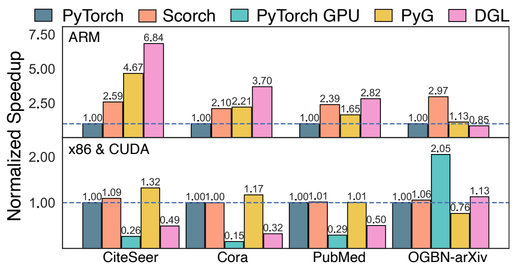


Fig. 9: Speedups of GCN inference implemented with Scorch, DGL, and PyG normalized to PyTorch.

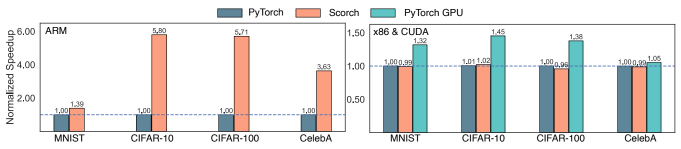


Fig. 10: Speedups on sparse autoencoder inference.

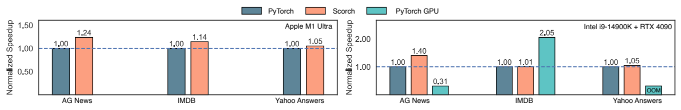


Fig. 11: Speedups on sparse transformer inference.

end-to-end application benchmarks are crucial to demonstrating that these components and improvements translate to practical benefits for machine learning practitioners. We evaluate three diverse workloads that represent different computational patterns and types of sparsity: graph neural networks (data sparsity), sparse autoencoders (weight sparsity), and sparse transformers (activation sparsity). This diversity allows us to validate that our automatic optimization approach generalizes across the spectrum of sparse deep learning applications.

1) Graph Neural Networks: Graph neural networks represent a key application area for sparse computation, as real-world graphs naturally exhibit sparsity in their input data, such as adjacency matrices. We evaluate a standard 2-layer Graph Convolutional Network (GCN) [24] on four benchmark datasets of increasing size: Cora (2,708 nodes, 5,429 edges) [36], CiteSeer (3,327 nodes, 4,732 edges) [19], PubMed (19,717 nodes, 44,338 edges) [46], and OGBN-ArXiv (169,343 nodes, 1,166,243 edges) [22]. As a critical comparison point, we include specialized graph libraries PyG [16] and DGL [51], which have been specifically written for graph computations.

The results in Figure 9 yield several insights about our approach. First, Scorch achieves a 2.1× mean speedup over PyTorch on the ARM CPU. This shows that our automatic loop ordering and tiling optimizations deliver meaningful benefits for real applications, not just isolated kernels. Second, our general-purpose sparse tensor framework achieves competitive performance with specialized graph libraries on most datasets. This is noteworthy because PyG and DGL are specialized for graph workloads. The fact that our domain-agnostic approach can match these hand-optimized implementations provides strong evidence that principled compilation techniques, when properly automated, can deliver the benefits of sparse computation without domain-specific engineering.

Performance analysis across varying graph sizes reveals algorithmic trade-offs. For small graphs, such as CiteSeer and Cora, the gather-scatter implementations of PyG and DGL have better performance than direct sparse matrix multiplication. As the graph dimensions increase, however, this performance advantage diminishes, primarily due to worse cache locality. The speedups of DGL fall below 1× for the largest problems.

Notably, the GPU results reinforce the importance of CPU execution for sparse workloads. As shown in Figure 9, the counter-intuitive finding that GPU implementations are slower than CPU for smaller graphs (0.15× to 0.29× relative to PyTorch CPU) highlights a crucial insight: sparse computations often lack the computational intensity to amortize GPU kernel launch overheads, especially at smaller scales. This suggests that optimizing CPU execution remains fundamentally important for sparse machine learning workloads, where the irregular memory access patterns and lower arithmetic intensity can limit the effectiveness of GPU acceleration.

2) Sparse Autoencoders: Sparse autoencoders represent a fundamentally different application of sparsity than graph neural networks. While GNNs exhibit sparsity in the input data, sparse autoencoders induce weight sparsity as a form of regularization [41]. We evaluate our system on a sparse autoencoder architecture consisting of an encoder with a sparse linear layer followed by ReLU activation and a decoder with a dense linear layer followed by sigmoid activation, on four benchmark datasets: MNIST [30], CIFAR-10 [27], CIFAR-100 [27], and CelebA [34] datasets.

The results in Figure 10 show speedups ranging from 1.39× to 5.80× over PyTorch on the ARM CPU, with the largest gains on more complex datasets like CIFAR-100 and CelebA. This pattern of increasing speedups with dataset complexity provides further evidence that our optimization approach scales effectively with problem size, which is crucial for usage in the real world where datasets continue to grow larger. Our performance parity with PyTorch on x86 again demonstrates that automated compilation approach can match highly optimized vendor libraries (Intel MKL), while the modest GPU speedups (1.05× to 1.45×) show that CPU execution remains competitive for many practical sparse workloads.

3) Sparse Transformers: To evaluate our system on highly complex sparse computation patterns, we implement and analyze sparse transformer models, which exhibit activation sparsity in attention patterns with the complex computational graphs typical of modern language models [50, 56, 12]. These models leverage sparse attention mechanisms to process longer sequences than would be possible with dense attention, making them an effective benchmark for assessing our system's ability to handle sparsity in complex real-world applications.

We evaluate the BigBird architecture [56] on three text classification datasets: IMDB [35], AG News [59], and Yahoo Answers [59]. The results in Figure 11 reveal several implications of our work. First, Scorch achieves consistent speedups over PyTorch CPU (1.01× to 1.40×) across all datasets and both CPU architectures. While speedups are more modest than for other workloads, they are significant given the complexity of transformer architectures. The consistent improvements demonstrate that our optimization approach generalizes to complex modern architectures with uncommon sparse operations and not just classical sparse workloads.

Second, the out-of-memory (OOM) result for PyTorch GPU on the Yahoo Answers dataset highlights an inherent limitation of GPU execution: VRAM capacity. CPU execution provides access to substantially larger system memory, enabling processing of models and datasets that exceed GPU memory constraints. As language models and sequence lengths continue to grow, this memory advantage becomes increasingly relevant for production deployment, even as GPU hardware evolves.

The variable GPU performance across datasets (from $2.05\times$ speedup to $0.31\times$ slowdown) reinforces the importance of efficient CPU execution for sparse workloads. This variability stems from the tension between the computational benefits of GPU parallelism and the overheads of kernel launches and memory transfers, which are precisely the trade-offs that our CPU-focused optimizations aim to address.

By automatically generating efficient sparse implementations for complex operations without existing specialized, hand-written kernels, we enable practitioners to explore novel sparse architectures with minimal engineering effort. This addresses the core thesis of our work—bridging the gap between sparse tensor compilers and practical machine learning.

VII. Related Work

Deep Learning Frameworks. Mainstream deep learning frameworks like PyTorch [42], JAX [9], and TensorFlow [1] provide limited support for sparse computation, focusing on a narrow set of sparse formats (COO, CSR) and operations (SpMM, SpMV). Low-level libraries like MKL-Sparse and cuSPARSE offer optimized primitives but lack support for high-order tensor operations, diverse formats, and operator fusion. Domain-specific libraries like PyG [16] and DGL [51] target sparse computation for specific domains but lack generality. We bridge these gaps by providing a unified sparse tensor abstraction for efficient computation across domains.

Tensor Algebra and ML Compilers. Compiler frameworks like Halide [43], TVM [11], XLA [20], Glow [45], and Tensor Comprehensions [49] optimize dense tensor operations but lack first-class sparse support. Sparse tensor algebra compilers, including Taco [25], Tiramisu [6], MLIR Sparse [8], and SparseTIR [55] have advanced sparse code generation, and new compilation models for shape operators and convolutions have also been proposed [44, 32]. However, these compilers require manual tuning, format selection and scheduling decisions. They also lack native integration with machine learning frameworks. We address these limitations by proposing fast automatic optimization and seamless PyTorch integration.

Autoscheduling. Prior autoscheduling techniques for sparse tensors [3, 23] rely on cost modeling and search-based methods, which are too slow for dynamic ML workloads. Pigeon [3] uses exhaustive enumeration while SpTTN-Cyclops [23] tunes contraction paths and index orders for specific SpTTN kernels (contractions of a single large sparse tensor with several dense tensors). Scorch contributes a lightweight heuristic-based autoscheduler that efficiently explores the schedule space for general sparse kernels without requiring offline search.

Sparse Deep Learning. Sparsity naturally occurs in various domains, including recommendation systems [40], drug discovery [58, 15], graph analytics [21], and via model sparsification techniques like pruning [37, 38, 17]. However, exploiting this sparsity typically requires custom kernels, which creates significant engineering barriers [18]. Scorch enables researchers to explore novel sparse architectures without implementing specialized kernels, reducing engineering burden and focusing research on modeling innovations.

Sparse Tensor Formats and Kernels. Various compressed sparse tensor formats have been developed to store only non-zero values and their indices, such as compressed sparse row/column storage (CSR/CSC), coordinate format (COO), CSB [10], CSR5 [33], HiCOO [31], and block sparse formats. Specialized sparse kernels implementing operations like SpMM and SpMV leverage techniques such as loop reordering and tiling [18]. While libraries like NVIDIA cuSPARSE and Intel MKL provide optimized implementations, they support limited operations and formats and lack fusion support.

Tiling Sparse Kernels. Recent work explores tiling for sparse kernels: Kurt et al. [28] investigate model-based tile size optimization for sparse kernels; Wilkinson et al. [52] propose register-level tiling using offline ILP optimization to pre-compute pattern-specific micro-kernels, which has strong performance but requires runtime inspection and data reorganization. These approaches prioritize performance through extensive optimization using runtime pattern analysis. Scorch instead uses fast heuristics that compile in milliseconds without runtime inspection, trading potentially higher performance for *predictable* performance and practical ML integration. Our evaluation shows our approach achieves competitive performance when inspection overhead cannot be amortized. The Nano comparison (Figure 8) demonstrates this trade-off: specialized micro-kernels achieve better per-kernel times, but dynamic workloads favor heuristic approaches that avoid inspection overhead.

VIII. Conclusion

We present three algorithms that enable fast, automatic optimization of sparse tensor computations: a heuristic-based loop ordering algorithm that avoids asymptotic performance cliffs while compiling in milliseconds, a tiling algorithm for mixed sparse-dense computations that matches hand-tuned kernels in performance, and a format inference algorithm that automatically determines appropriate sparse representations. The algorithms demonstrate that principled heuristic approaches grounded in sparse tensor algebra properties can bridge the gap between sparse compilers and practical ML frameworks.

Acknowledgments

We thank our anonymous reviewers for their feedback. We also thank Manya Bansal, James Dong, Christophe Gyurgyik, Olivia Hsu, Daniel Kim, Scott Kovach, Rubens Lacouture, Devanshu Ladsaria, Shiv Sundram, and Rohan Yadav, for valuable discussion and comments on early drafts of this paper. This work was in part supported by the National Science Foundation under grant CCF-2143061, the DARPA MOCHA program, the PRISM center, one of seven centers in JUMP 2.0, a Semiconductor Research Corporation (SRC) program sponsored by DARPA, the Swedish Research Council (Grant No. 2018-04329), Digital Futures, and the Stanford Portal Center. Alexander was supported by the NSF Graduate Research Fellowship and the Qualcomm Innovation Fellowship.

References

[1] Martin Abadi, Paul Barham, Jianmin Chen, Zhifeng Chen, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Geoffrey Irving, Michael Isard, Manjunath Kudlur, Josh Levenberg, Rajat Monga, Sherry Moore, Derek G. Murray, Benoit Steiner, Paul Tucker, Vijay Vasudevan, Pete Warden, Martin Wicke, Yuan Yu, and Xiaoqiang Zheng. TensorFlow: A system for large-scale machine learning. In *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)*, pages 265–283, 2016. URL https://www.usenix.org/system/files/conference/osdi16/osdi16-abadi.pdf.

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

[3] Willow Ahrens, Fredrik Kjolstad, and Saman Amarasinghe. 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*, PLDI 2022, page 269–285, New York, NY, USA, 2022. Association for Computing Machinery. ISBN 9781450392655. doi: 10.1145/3519939.3523442. URL https://doi.org/10.1145/3519939.3523442.

[4] Willow Ahrens, Daniel Donenfeld, Fredrik Kjolstad, and Saman P. Amarasinghe. Looplets: A language for structured coiteration. In Christophe Dubach, Derek Bruening, and Ben Hardekopf, editors, *Proceedings of the 21st ACM/IEEE International Symposium on Code Generation and Optimization, CGO 2023, Montréal, QC, Canada, 25 February 2023- 1 March 2023*, pages 41–54. ACM, 2023. doi: 10.1145/3579990.3580020. URL https://doi.org/10.1145/3579990.3580020.

[5] 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. 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*, ASPLOS '24, page 929–947, New York, NY, USA, 2024. Association for Computing Machinery. ISBN 9798400703850. doi: 10.1145/3620665.3640366. URL https://doi.org/10.1145/3620665.3640366.

[6] Riyadh Baghdadi, Jessica Ray, Malek Ben Romdhane, Emanuele Del Sozzo, Abdurrahman Akkas, Yunming Zhang, Patricia Suriana, Shoaib Kamil, and Saman P. Amarasinghe. Tiramisu: A polyhedral compiler for expressing fast and portable code. In Mahmut Taylan Kandemir, Alexandra Jimborean, and Tipp Moseley, editors, *IEEE/ACM International Symposium on Code Generation and Optimization, CGO 2019, Washington, DC, USA, February 16-20, 2019*, pages 193–205. IEEE, 2019. doi: 10.1109/CGO.2019.8661197. URL https://doi.org/10.1109/CGO.2019.8661197.

[7] Manya Bansal, Olivia Hsu, Kunle Olukotun, and Fredrik Kjolstad. Mosaic: An interoperable compiler for tensor algebra. *Proc. ACM Program. Lang.*, 7(PLDI), June 2023. doi: 10.1145/3591236. URL https://doi.org/10.1145/3591236.

[8] Aart J. C. Bik, Penporn Koanantakool, Tatiana Shpeisman, Nicolas Vasilache, Bixia Zheng, and Fredrik Kjolstad. Compiler support for sparse tensor computations in MLIR. *ACM Trans. Archit. Code Optim.*, 19(4):50:1–50:25, 2022. doi: 10.1145/3544559. URL https://doi.org/10.1145/3544559.

[9] 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. JAX: composable transformations of Python+NumPy programs, 2018. URL http://github.com/google/jax.

[10] Aydin Buluç, Jeremy T. Fineman, Matteo Frigo, John R. Gilbert, and Charles E. Leiserson. Parallel sparse matrix-vector and matrix-transpose-vector multiplication using compressed sparse blocks. In *Proceedings of the Twenty-First Annual Symposium on Parallelism in Algorithms and Architectures*, SPAA '09, page 233–244, New York, NY, USA, 2009. Association for Computing Machinery. ISBN 9781605586069. doi: 10.1145/1583991.1584053. URL https://doi.org/10.1145/1583991.1584053.

[11] Tianqi Chen, Thierry Moreau, Ziheng Jiang, Lianmin Zheng, Eddie Q. Yan, Haichen Shen, Meghan Cowan, Leyuan Wang, Yuwei Hu, Luis Ceze, Carlos Guestrin, and Arvind Krishnamurthy. TVM: an automated end-to-

end optimizing compiler for deep learning. In Andrea C. Arpaci-Dusseau and Geoff Voelker, editors, *13th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2018, Carlsbad, CA, USA, October 8-10, 2018*, pages 578–594. USENIX Association, 2018. URL https://www.usenix.org/conference/osdi18/presentation/chen.

[12] Rewon Child, Scott Gray, Alec Radford, and Ilya Sutskever. Generating long sequences with sparse transformers. *CoRR*, abs/1904.10509, 2019. URL http://arxiv.org/abs/1904.10509.

[13] Stephen Chou, Fredrik Kjolstad, and Saman P. Amarasinghe. Format abstraction for sparse tensor algebra compilers. *Proc. ACM Program. Lang.*, 2(OOPSLA): 123:1–123:30, 2018. doi: 10.1145/3276493. URL https://doi.org/10.1145/3276493.

[14] Timothy A. Davis and Yifan Hu. The university of florida sparse matrix collection. *ACM Trans. Math. Softw.*, 38 (1):1:1–1:25, 2011. doi: 10.1145/2049662.2049663. URL https://doi.org/10.1145/2049662.2049663.

[15] Moe Elbadawi, Simon Gaisford, and Abdul W. Basit. Advanced machine-learning techniques in drug discovery. *Drug Discovery Today*, 26(3):769–777, 2021. ISSN 1359-6446. doi: https://doi.org/10.1016/j.drudis.2020.12.003. URL https://www.sciencedirect.com/science/article/pii/S1359644620305213.

[16] Matthias Fey and Jan Eric Lenssen. Fast Graph Representation Learning with PyTorch Geometric. *CoRR*, abs/1903.02428, 2019. URL http://arxiv.org/abs/1903.02428.

[17] Jonathan Frankle and Michael Carbin. The lottery ticket hypothesis: Finding sparse, trainable neural networks. In *7th International Conference on Learning Representations, ICLR 2019, New Orleans, LA, USA, May 6-9, 2019*. OpenReview.net, 2019. URL https://openreview.net/forum?id=rJl-b3RcF7.

[18] Trevor Gale, Matei Zaharia, Cliff Young, and Erich Elsen. Sparse GPU kernels for deep learning. In Christine Cuicchi, Irene Qualters, and William T. Kramer, editors, *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis, SC 2020, Virtual Event / Atlanta, Georgia, USA, November 9-19, 2020*, page 17. IEEE/ACM, 2020. doi: 10.1109/SC41405.2020.00021. URL https://doi.org/10.1109/SC41405.2020.00021.

[19] C. Lee Giles, Kurt D. Bollacker, and Steve Lawrence. Citeseer: an automatic citation indexing system. In *Proceedings of the Third ACM Conference on Digital Libraries*, DL '98, page 89–98, New York, NY, USA, 1998. Association for Computing Machinery. ISBN 0897919653. doi: 10.1145/276675.276685. URL https://doi.org/10.1145/276675.276685.

[20] Google. Open xla, 2024. URL https://openxla.org/xla.

[21] William L. Hamilton, Zhitao Ying, and Jure Leskovec. Inductive representation learning on large graphs. In Isabelle Guyon, Ulrike von Luxburg, Samy Bengio, Hanna M. Wallach, Rob Fergus, S. V. N. Vishwanathan, and Roman Garnett, editors, *Advances in Neural Information Processing Systems 30: Annual Conference on Neural Information Processing Systems 2017, December 4-9, 2017, Long Beach, CA, USA*, pages 1024–1034, 2017. URL https://proceedings.neurips.cc/paper/2017/hash/5dd9db5e033da9c6fb5ba83c7a7ebea9-Abstract.html.

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

[23] Raghavendra Kanakagiri and Edgar Solomonik. Minimum cost loop nests for contraction of a sparse tensor with a tensor network. *arXiv preprint arXiv:2307.05740*, 2023. URL https://doi.org/10.48550/arXiv.2307.05740.

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

[25] Fredrik Kjolstad, Shoaib Kamil, Stephen Chou, David Lugato, and Saman P. Amarasinghe. The tensor algebra compiler. *Proc. ACM Program. Lang.*, 1(OOPSLA):77:1–77:29, 2017. doi: 10.1145/3133901. URL https://doi.org/10.1145/3133901.

[26] Fredrik Kjolstad, Willow Ahrens, Shoaib Kamil, and Saman P. Amarasinghe. Tensor algebra compilation with workspaces. In Mahmut Taylan Kandemir, Alexandra Jimborean, and Tipp Moseley, editors, *IEEE/ACM International Symposium on Code Generation and Optimization, CGO 2019, Washington, DC, USA, February 16-20, 2019*, pages 180–192. IEEE, 2019. doi: 10.1109/CGO.2019.8661185. URL https://doi.org/10.1109/CGO.2019.8661185.

[27] Alex Krizhevsky, Geoffrey Hinton, et al. Learning multiple layers of features from tiny images. Technical report, University of Toronto, 2009. URL https://www.cs.toronto.edu/~kriz/learning-features-2009-TR.pdf.

[28] Süreyya Emre Kurt, Aravind Sukumaran-Rajam, Fabrice Rastello, and P. Sadayappan. Efficient tiled sparse matrix multiplication through matrix signatures. In Christine Cuicchi, Irene Qualters, and William T. Kramer, editors, *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis, SC 2020, Virtual Event / Atlanta, Georgia, USA, November 9-19, 2020*, page 87. IEEE/ACM, 2020. doi: 10.1109/SC41405.2020.00091. URL https://doi.org/

10.1109/SC41405.2020.00091.

[29] Eldar Kurtic, Denis Kuznedelev, Elias Frantar, Michael Goin, and Dan Alistarh. Sparse fine-tuning for inference acceleration of large language models, 2023. URL https://arxiv.org/abs/2310.06927.

[30] Yann LeCun, Léon Bottou, Yoshua Bengio, and Patrick Haffner. Gradient-based learning applied to document recognition. *Proc. IEEE*, 86(11):2278–2324, 1998. doi: 10.1109/5.726791. URL https://doi.org/10.1109/5.726791.

[31] Jiajia Li, Jimeng Sun, and Richard Vuduc. Hicoo: Hierarchical storage of sparse tensors. In *SC18: International Conference for High Performance Computing, Networking, Storage and Analysis*, pages 238–252, 2018. doi: 10.1109/SC.2018.00022.

[32] Peiming Liu, Alexander J Root, Anlun Xu, Yinying Li, Fredrik Kjolstad, and Aart J.C. Bik. Compiler support for sparse tensor convolutions. *Proc. ACM Program. Lang.*, 8 (OOPSLA2), October 2024. doi: 10.1145/3689721. URL https://doi.org/10.1145/3689721.

[33] Weifeng Liu and Brian Vinter. Csr5: An efficient storage format for cross-platform sparse matrix-vector multiplication. In *Proceedings of the 29th ACM on International Conference on Supercomputing*, ICS '15, page 339–350, New York, NY, USA, 2015. Association for Computing Machinery. ISBN 9781450335591. doi: 10.1145/2751205.2751209. URL https://doi.org/10.1145/2751205.2751209.

[34] Ziwei Liu, Ping Luo, Xiaogang Wang, and Xiaoou Tang. Deep Learning Face Attributes in the Wild. In *2015 IEEE International Conference on Computer Vision, ICCV 2015, Santiago, Chile, December 7-13, 2015*, pages 3730–3738. IEEE Computer Society, 2015. doi: 10.1109/ICCV.2015.425. URL https://doi.org/10.1109/ICCV.2015.425.

[35] Andrew L. Maas, Raymond E. Daly, Peter T. Pham, Dan Huang, Andrew Y. Ng, and Christopher Potts. Learning word vectors for sentiment analysis. In *Proceedings of the 49th Annual Meeting of the Association for Computational Linguistics: Human Language Technologies*, pages 142–150, Portland, Oregon, USA, June 2011. Association for Computational Linguistics. URL http://www.aclweb.org/anthology/P11-1015.

[36] Andrew McCallum, Kamal Nigam, Jason Rennie, and Kristie Seymore. Automating the construction of internet portals with machine learning. *Inf. Retr.*, 3(2):127–163, 2000. doi: 10.1023/A:1009953814988. URL https://doi.org/10.1023/A:1009953814988.

[37] Decebal Constantin Mocanu, Elena Mocanu, Peter Stone, Phuong H. Nguyen, Madeleine Gibescu, and Antonio Liotta. Scalable training of artificial neural networks with adaptive sparse connectivity inspired by network science. *Nature Communications*, 9(1):2383, 2018. ISSN 2041-1723. doi: 10.1038/s41467-018-04316-3. URL https://www.nature.com/articles/s41467-018-04316-3. Publisher: Nature Publishing Group.

[38] Hesham Mostafa and Xin Wang. Parameter efficient training of deep convolutional neural networks by dynamic sparse reparameterization. In Kamalika Chaudhuri and Ruslan Salakhutdinov, editors, *Proceedings of the 36th International Conference on Machine Learning, ICML 2019, 9-15 June 2019, Long Beach, California, USA*, volume 97 of *Proceedings of Machine Learning Research*, pages 4646–4655. PMLR, 2019. URL http://proceedings.mlr.press/v97/mostafa19a.html.

[39] Suzanne Mueller, Peter Ahrens, Stephen Chou, Fredrik Kjolstad, and Saman Amarasinghe. Brief announcement: Sparse tensor transpositions. 2020.

[40] Maxim Naumov, Dheevatsa Mudigere, Hao-Jun Michael Shi, Jianyu Huang, Narayanan Sundaraman, Jongsoo Park, Xiaodong Wang, Udit Gupta, Carole-Jean Wu, Alisson G. Azzolini, Dmytro Dzhulgakov, Andrey Mallevich, Ilia Cherniavskii, Yinghai Lu, Raghuraman Krishnamoorthi, Ansha Yu, Volodymyr Kondratenko, Stephanie Pereira, Xianjie Chen, Wenlin Chen, Vijay Rao, Bill Jia, Liang Xiong, and Misha Smelyanskiy. Deep learning recommendation model for personalization and recommendation systems. *CoRR*, abs/1906.00091, 2019. URL https://arxiv.org/abs/1906.00091.

[41] Andrew Ng et al. Sparse autoencoder. *CS294A Lecture notes*, 72(2011):1–19, 2011. URL https://web.stanford.edu/class/cs294a/sparseAutoencoder_2011new.pdf.

[42] Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, Alban Desmaison, Andreas Köpf, Edward Z. Yang, Zachary DeVito, Martin Raison, Alykhan Tejani, Sasank Chilamkurthy, Benoit Steiner, Lu Fang, Junjie Bai, and Soumith Chintala. PyTorch: An Imperative Style, High-Performance Deep Learning Library. In *Advances in Neural Information Processing Systems 32*, pages 8024–8035, 2019. URL https://proceedings.neurips.cc/paper/2019/hash/bdbca288fee7f92f2bfa9f7012727740-Abstract.html.

[43] Jonathan Ragan-Kelley, Connelly Barnes, Andrew Adams, Sylvain Paris, Frédo Durand, and Saman Amarasinghe. 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, page 519–530, New York, NY, USA, 2013. Association for Computing Machinery. ISBN 9781450320146. doi: 10.1145/2491956.2462176. URL https://doi.org/10.1145/2491956.2462176.

[44] Alexander J Root, Bobby Yan, Peiming Liu, Christophe Gyurgyik, Aart J.C. Bik, and Fredrik Kjolstad. Compilation of shape operators on sparse arrays. *Proc. ACM Program. Lang.*, 8(OOPSLA2), October 2024. doi: 10.1145/3689752. URL https://doi.org/10.1145/3689752.

[45] Nadav Rotem, Jordan Fix, Saleem Abdulrasool, Summer Deng, Roman Dzhabarov, James Hegeman, Roman Levenstein, Bert Maher, Nadathur Satish, Jakob Olesen,

Jongsoo Park, Artem Rakhov, and Misha Smelyanskiy. Glow: Graph lowering compiler techniques for neural networks. *CoRR*, abs/1805.00907, 2018. URL http://arxiv.org/abs/1805.00907.

[46] Prithviraj Sen, Galileo Namata, Mustafa Bilgic, Lise Getoor, Brian Gallagher, and Tina Eliassi-Rad. Collective classification in network data. *AI Mag.*, 29(3):93–106, 2008. doi: 10.1609/AIMAG.V29I3.2157. URL https://doi.org/10.1609/aimag.v29i3.2157.

[47] Ryan Senanayake, Changwan Hong, Ziheng Wang, Amalee Wilson, Stephen Chou, Shoaib Kamil, Saman Amarasinghe, and Fredrik Kjolstad. A sparse iteration space transformation framework for sparse tensor algebra. *Proc. ACM Program. Lang.*, 4(OOPSLA), nov 2020. doi: 10.1145/3428226. URL https://doi.org/10.1145/3428226.

[48] John Thorpe, Yifan Qiao, Jonathan Eyolfson, Shen Teng, Guanzhou Hu, Zhihao Jia, Jinliang Wei, Keval Vora, Ravi Netravali, Miryung Kim, and Guoqing Harry Xu. Dorylus: Affordable, scalable, and accurate GNN training with distributed CPU servers and serverless threads. In *15th USENIX Symposium on Operating Systems Design and Implementation (OSDI 21)*, pages 495–514. USENIX Association, July 2021. ISBN 978-1-939133-22-9. URL https://www.usenix.org/conference/osdi21/presentation/thorpe.

[49] Nicolas Vasilache, Oleksandr Zinenko, Theodoros Theodoridis, Priya Goyal, Zachary DeVito, William S. Moses, Sven Verdoolaege, Andrew Adams, and Albert Cohen. Tensor comprehensions: Framework-agnostic high-performance machine learning abstractions. *CoRR*, abs/1802.04730, 2018. URL http://arxiv.org/abs/1802.04730.

[50] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N. Gomez, Lukasz Kaiser, and Illia Polosukhin. Attention is all you need. In Isabelle Guyon, Ulrike von Luxburg, Samy Bengio, Hanna M. Wallach, Rob Fergus, S. V. N. Vishwanathan, and Roman Garnett, editors, *Advances in Neural Information Processing Systems 30: Annual Conference on Neural Information Processing Systems 2017, December 4-9, 2017, Long Beach, CA, USA*, pages 5998–6008, 2017. URL https://proceedings.neurips.cc/paper/2017/hash/3f5ee243547dee91fbd053c1c4a845aa-Abstract.html.

[51] Minjie Wang, Da Zheng, Zihao Ye, Quan Gan, Mufei Li, Xiang Song, Jinjing Zhou, Chao Ma, Lingfan Yu, Yu Gai, et al. Deep graph library: A graph-centric, highly-performant package for graph neural networks. *arXiv preprint arXiv:1909.01315*, 2019. URL http://arxiv.org/abs/1909.01315.

[52] Lucas Wilkinson, Kazem Cheshmi, and Maryam Mehri Dehnavi. Register tiling for unstructured sparsity in neural network inference. *Proc. ACM Program. Lang.*, 7(PLDI): 1995–2020, 2023. doi: 10.1145/3591302. URL https://doi.org/10.1145/3591302.

[53] Michael E. Wolf and Monica S. Lam. A data locality optimizing algorithm. In *Proceedings of the ACM SIGPLAN 1991 Conference on Programming Language Design and Implementation*, PLDI '91, page 30–44, New York, NY, USA, 1991. Association for Computing Machinery. ISBN 0897914287. doi: 10.1145/113445.113449. URL https://doi-org.stanford.idm.oclc.org/10.1145/113445.113449.

[54] Rohan Yadav, Alex Aiken, and Fredrik Kjolstad. Distal: the distributed tensor algebra compiler. In *Proceedings of the 43rd ACM SIGPLAN International Conference on Programming Language Design and Implementation*, PLDI 2022, page 286–300, New York, NY, USA, 2022. Association for Computing Machinery. ISBN 9781450392655. doi: 10.1145/3519939.3523437. URL https://doi.org/10.1145/3519939.3523437.

[55] Zihao Ye, Ruihang Lai, Junru Shao, Tianqi Chen, and Luis Ceze. Sparsetir: Composable abstractions for sparse compilation in deep learning. In *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 3*, ASPLOS 2023, page 660–678, New York, NY, USA, 2023. Association for Computing Machinery. ISBN 9781450399180. doi: 10.1145/3582016.3582047. URL https://doi.org/10.1145/3582016.3582047.

[56] Manzil Zaheer, Guru Guruganesh, Kumar Avinava Dubey, Joshua Ainslie, Chris Alberti, Santiago Ontañón, Philip Pham, Anirudh Ravula, Qifan Wang, Li Yang, and Amr Ahmed. Big bird: Transformers for longer sequences. In Hugo Larochelle, Marc'Aurelio Ranzato, Raia Hadsell, Maria-Florina Balcan, and Hsuan-Tien Lin, editors, *Advances in Neural Information Processing Systems 33: Annual Conference on Neural Information Processing Systems 2020, NeurIPS 2020, December 6-12, 2020, virtual*, 2020. URL https://proceedings.neurips.cc/paper/2020/hash/c8512d142a2d849725f31a9a7a361ab9-Abstract.html.

[57] Genghan Zhang, Olivia Hsu, and Fredrik Kjolstad. Compilation of modular and general sparse workspaces. *arXiv preprint arXiv:2404.04541*, 2024. URL https://arxiv.org/abs/2404.04541.

[58] Wen Zhang, Kanghong Jing, Feng Huang, Yanlin Chen, Bo-Sheng Li, Jinghao Li, and Jing Gong. Sflln: A sparse feature learning ensemble method with linear neighborhood regularization for predicting drug-drug interactions. *Inf. Sci.*, 497:189–201, 2019. URL https://api.semanticscholar.org/CorpusID:182751868.

[59] Xiang Zhang, Junbo Jake Zhao, and Yann LeCun. Character-level convolutional networks for text classification. In Corinna Cortes, Neil D. Lawrence, Daniel D. Lee, Masashi Sugiyama, and Roman Garnett, editors, *Advances in Neural Information Processing Systems 28: Annual Conference on Neural Information Processing Systems 2015, December 7-12, 2015, Montreal, Quebec, Canada*, pages 649–657, 2015. URL https://proceedings.neurips.cc/paper/2015/hash/250cf8b51c773f3f8dc8b4be867a9a02-Abstract.html.

[60] Lianmin Zheng, Chengfan Jia, Minmin Sun, Zhao Wu, Cody Hao Yu, Ameer Haj-Ali, Yida Wang, Jun Yang,

Danyang Zhuo, Koushik Sen, Joseph E. Gonzalez, and Ion Stoica. Ansor: Generating High-Performance Tensor Programs for Deep Learning. In *14th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2020, Virtual Event, November 4-6, 2020*, pages 863–879. USENIX Association, 2020. URL https://www.usenix.org/conference/osdi20/presentation/zheng.

[61] Hongyu Zhu, Ruofan Wu, Yijia Diao, Shanbin Ke, Haoyu Li, Chen Zhang, Jilong Xue, Lingxiao Ma, Yuqing Xia, Wei Cui, Fan Yang, Mao Yang, Lidong Zhou, Asaf Cidon, and Gennady Pekhimenko. ROLLER: Fast and efficient tensor compilation for deep learning. In *16th USENIX Symposium on Operating Systems Design and Implementation (OSDI 22)*, pages 233–248, Carlsbad, CA, July 2022. USENIX Association. ISBN 978-1-939133-28-1. URL https://www.usenix.org/conference/osdi22/presentation/zhu.