

Partitioning Unstructured Sparse Tensor Algebra for Load-Balanced Parallel Execution

Atharva Chougule*
Stanford University
Stanford, CA, USA
atharvac@cs.stanford.edu

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

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

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

Rohan Yadav
Stanford University
Stanford, CA, USA
rohany@cs.stanford.edu

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

Abstract

Sparse tensor algebra is challenging to efficiently parallelize due to the irregular, data-dependent, and potentially skewed structure of sparse computation. We propose the first partitioning algorithm that provably load balances the computation of any sparse tensor algebra expression across parallel execution units. Our algorithm generalizes parallel merging algorithms to any number of operands, and to multi-dimensional, hierarchical sparse data structures. We implement our algorithm within an existing sparse tensor algebra compilation framework to automatically generate parallel sparse tensor algebra kernels that target multi-core CPUs and GPUs. We show that our generated code is competitive with hand-implemented parallelization strategies used by vendor libraries like Intel MKL and NVIDIA cuSPARSE (geo-means of 0.93–5.5×) and Taco (geo-means of 1.2–2.4×), and significantly outperforms PyTorch Sparse's general-purpose strategies on sparse tensor expressions where specialized algorithms have not been developed (geo-means of 2.0–9.1×).

CCS Concepts: • **Software and its engineering → Source code generation**; • **Computing methodologies → Parallel algorithms**.

Keywords: sparse tensor algebra, parallelism, load balance, GPUs, multicore CPUs, partitioning, compilers

ACM Reference Format:
Atharva Chougule, Alexander J Root, Rubens Lacouture, Bobby Yan, Rohan Yadav, and Fredrik Kjolstad. 2027. Partitioning Unstructured Sparse Tensor Algebra for Load-Balanced Parallel Execution. In *Proceedings of the 32nd ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 1 (ASPLOS '27), April 11–15, 2027, Heraklion, Greece.* ACM, New York, NY, USA, 17 pages. https://doi.org/10.1145/3845814.3850062

*Equal contribution.



This work is licensed under a Creative Commons Attribution 4.0 International License.

ASPLOS '27, Heraklion, Greece

© 2027 Copyright held by the owner/author(s).

ACM ISBN 979-8-4007-3012-2/2027/04
https://doi.org/10.1145/3845814.3850062

1 Introduction

Sparse tensor algebra is used across many computational domains [20, 34, 38], and high-performance implementations of sparse tensor algebra operations enable these domains to solve important problems quickly. Parallelization is a critical component of efficient execution on modern machines, and the main challenge in parallelizing sparse tensor algebra lies in load balancing the irregular work created from *coiteration*. Coiteration is the process of simultaneously iterating over multiple sparse tensors and performing different operations based on how the coordinates in each sparse data structure match. The total amount of work performed by coiteration is data-dependent and a function of the interaction between sparse data structures, making extracting equally-sized pieces of work a search problem. This difficulty is exacerbated by the increased irregularity that comes from the hierarchical structure in both storing and coiterating over multi-dimensional sparse tensors.

The community has developed a variety of compilers that take arbitrary sparse tensor expressions and descriptions of the underlying data structures and produce efficient implementations [1, 2, 36, 54]. Extensions of these compilers support various types of parallel hardware, e.g., multicore CPUs [21, 63, 67, 74], GPUs [4, 53, 64, 70, 73], supercomputers [66], or custom accelerators [28, 39, 55]. However, these parallelization strategies either do not guarantee the creation of equally-sized pieces of work for each parallel worker [4, 21, 28, 39, 53, 66, 67, 74] or can only do so for expressions with a single sparse tensor [55, 63, 64, 70, 73]. Additionally, these compilers often require user input in the form of scheduling directives to choose a parallelization strategy [4, 28, 39, 53, 55, 66, 70].

For specific kernels and specific sparse data structures, the community has developed algorithms that guarantee an equal distribution of work among parallel processors [16, 32, 47, 65, 68, 69]. Many of these algorithms [16, 65, 68, 69] are inspired by parallel merging algorithms from the sorting

community [22, 46], and extract load-balanced parallelism from the merge-like operations that implement coiteration in sparse tensor algebra kernels. These strategies are specialized for specific binary operations and only handle one and two-dimensional tensors. Other strategies developed for custom accelerators [32, 47] collect statistics across the operands to adapt partitions, but do so in a sequential manner.

We propose a load-balanced parallel partitioning technique that guarantees equally-sized pieces of work (within a small constant) for any number of tensors of any number of dimensions, and generalizes the ideas proposed in prior algorithms for specific expressions and data structures. Our partitioning technique efficiently searches the n-dimensional iteration space of the tensor algebra expression to find a load-balanced partitioning with respect to an abstract cost model. The key insight behind our partitioning strategy is that load-balanced *irregular* partitions of the iteration space can be extracted from a combination of the loop structure and efficient queries on sparse data structures. Critically, these queries can be performed independently, allowing for the partitioning strategy itself to be computed *in parallel* rather than as a separate pre-processing step. We implement a concrete cost model that corresponds to the number of non-zero values within a partition, which can be queried efficiently for many sparse tensor formats. By specializing our partitioning technique with this format-specific cost model and a concrete loop order, we generate custom partitioning algorithms for sparse kernels within a sparse tensor algebra compiler.

We implement our partitioning approach in a prototype compiler called Nacho,[1] which extends the Taco [14, 36] compiler to generate load-balanced parallel kernels for a general class of expressions and sparse data structures. To the best of our knowledge, Nacho is the first compiler to support load-balanced GPU code generation of sparse tensor algebra kernels with multiple sparse operands. Unlike prior approaches, our approach is fully automated and does not require user input (in the form of scheduling directives) to generate parallel code. The contributions of our work are:

- A parallel partitioning strategy for sparse tensor algebra that guarantees load-balanced parallel execution.
- A code generation approach that integrates this partitioning strategy into a sparse compilation framework.

To evaluate our contributions, we compare the performance of Nacho on a variety of sparse tensor algebra expressions on multi-core CPUs and GPUs. We show that Nacho achieves high performance when compared to tuned vendor libraries like Intel MKL [31], NVIDIA cuSPARSE [45], and the Taco compiler [36, 53]. We then show that the generality of our approach enables high performance for expressions not present in vendor libraries, heavily outperforming generic strategies in existing systems like PyTorch Sparse [50].

[1]The ideal plate of nachos load-balances cheese across all chips.

Listing 1. Sequential code computing $z_i = a_i \cdot b_i \cdot c_i$ via a three-finger merge on sparse vectors. nnz_x denotes the count of non-zeros in vector x and p_x is the iterator over vector x.

```
p_a = 0, p_b = 0, p_c = 0, p_z = 0
while p_a < nnz(a) && p_b < nnz(b) && p_c < nnz(c):
  i_a = a.i[p_a], i_b = b.i[p_b], i_c = c.i[p_c]
  i = min(i_a, i_b, i_c)
  if i == i_a && i == i_b && i == i_c:
    z.v[p_z] = a.v[p_a] * b.v[p_b] * c.v[p_c]
    z.i[p_z++] = i
  p_a += (i==i_a); p_b += (i==i_b); p_c += (i==i_c)
```

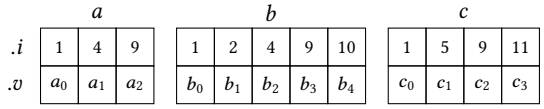


Figure 1. Sparse vectors a, b, and c, as index/value pairs.

2 Motivating Example and Challenges

We now use several examples to illustrate the concrete challenges of parallelizing multi-sparse tensor algebra operations. We start with coiteration over a single dimension before moving into additional challenges caused by multi-dimensional tensors as well as nested intersection iteration patterns.

2.1 Background: Coiteration

Sparse tensor compilers [1, 2, 7, 36, 54, 59] take advantage of the arithmetic properties $a \times 0 = 0$ and $a + 0 = a$ to iterate over only the *intersection* of non-zero coordinates to compute the result of a sparse multiplication, and the *union* of non-zero coordinates to compute the result of a sparse addition. There are multiple ways to perform such iteration, including the iterate-locate pattern for intersections [36, 51, 74], which iterates one operand and searches for a matching coordinate in the other operand. While this is asymptotically optimal if the searched operand is stored densely (and therefore stores values for all coordinates), it can be asymptotically sub-optimal if both operands are sparse. Iterate-locate also does not naively apply to unions of sparse operands. Therefore, Nacho follows Taco's approach for kernels with multiple sparse operands: generating *coiterative* while loops, similar to the merge operation of mergesort, to step through each sparse operand.

An example of a multi-way merge is provided in Listing 1, which computes the element-wise multiplication of three sparse vectors via a three-way merge. This multi-way merge maintains a pointer into each sparse vector and increments the pointers iteratively to compute the intersection, and performs the element-wise multiplication when all pointers match on a non-zero coordinate. This coiterative `while` loop is not amenable to standard data parallelization techniques; but can still be parallelized via a technique from the parallel sorting community.

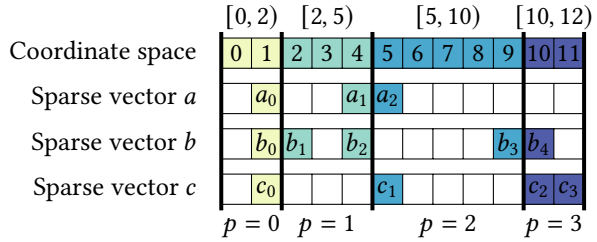


Figure 2. An irregular partitioning of the coordinate space into 4 parts such that 3 non-zeros are processed in each partition. Partition p's boundaries $[x_l, x_h)$ are obtained by independently searching for x_l and x_h s.t. $[0, x_l)$ contains $3p$ non-zeros and $[0, x_h)$ contains $3(p+1)$ non-zeros.

2.2 Parallelizing Coiteration

The parallel sorting community developed a technique to parallelize a coiterative *merge* of two vectors in a load-balanced manner [22, 46]. This technique searches the two sorted arrays to find non-overlapping index ranges that can be safely merged independently.

Our approach in Nacho adapts and generalizes this strategy to parallelize the coiterative intersections and unions performed by sparse tensor algebra kernels. Nacho partitions the dense space of all possible coordinates in each vector (referred to as the *coordinate space*) into *irregular* pieces, such that the number of non-zero elements processed in each partition is equal, rather than the total number of coordinates. Nacho finds these partition boundaries by binary searching over the coordinate space for ranges that enclose an equal number of non-zero coordinates across each sparse vector. When considering a candidate range $[x_l, x_h)$, the number of non-zero coordinates of each sparse vector contained within $[x_l, x_h)$ can be found by another binary search over the physical data structures storing the sparse vectors.

Figure 2 illustrates a partitioning found by using this search for three-way merge. The partition boundaries are found by searching for cutoff points in the coordinate space that enclose a specific number of non-zeros. For example, the first partition is found by searching for cutoff coordinate x_1 such that the range of coordinates from $[0, x_1)$ contains k non-zeros. Then, the second partition $[x_1, x_2)$ can be computed by also searching for x_1 and then searching for x_2 where the range of coordinates $[0, x_2)$ contains $2k$ non-zeros.

Critically, Nacho's strategy allows for the bounds of each range in the partition to be computed independently, and thus *in parallel*. As a result, the strategy can be applied to Listing 1 by having parallel workers independently compute the partition bounds shown in Figure 2, and then begin load-balanced coiteration within the corresponding ranges.

2.3 Higher-Order Coiteration

While the three-way sparse vector multiplication example shown previously builds intuition for parallelizing coiteration of multiple sparse operands, generalizing these ideas also requires extending to multiple tensor dimensions. Consider the coiteration required for an addition between two Doubly Compressed Sparse Row (DCSR) [10] matrices in Listing 2. This hierarchical coiteration requires a partitioning strategy that accounts for work across multiple dimensions. Figure 3a depicts one such strategy, with a two-dimensional irregular partitioning to load balance the number of non-zeros processed in each partition.

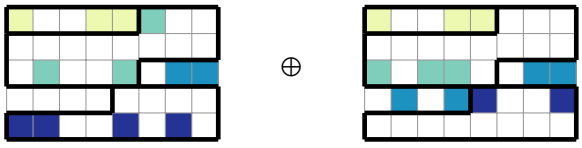
Listing 2. $Z_{ij} = A_{ij} + B_{ij}$ with DCSR inputs.

```
while i ← rows(A) ∪ rows(B):
  while j ← cols(A[i]) ∪ cols(B[i]):
    Z[i, j] = A[i, j] + B[i, j]
```

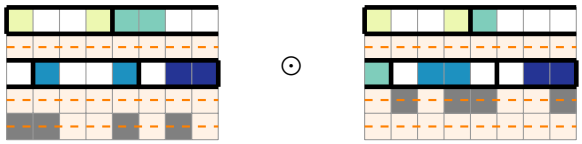
Listing 3. $Z_{ij} = A_{ij} \cdot B_{ij}$ with DCSR inputs.

```
while i ← rows(A) ∩ rows(B):
  while j ← cols(A[i]) ∩ cols(B[i]):
    Z[i, j] = A[i, j] * B[i, j]
```

The partitioning problem is further complicated by *hierarchical skipping*, as observed in the Hadamard product on DCSR matrices in Listing 3. Here, coiteration over the j dimension is restricted to rows that are nonempty in both operands. The sparse intersection in the outer loop causes entire rows to be skipped, and consequently the inner iteration over columns is skipped for those rows. This makes computing load-balanced partitions difficult, as large regions of the coordinate space may correspond to no non-zeros. Achieving load-balance in this case requires determining which rows contribute to the work and then constructing partitions only over these rows, as visualized in Figure 3b.



(a) Partitioning for DCSR addition: 6 non-zeros per chunk.



(b) Partitioning for DCSR (element-wise) multiplication. Entire rows are skipped (highlighted in orange) due to hierarchical skipping.

Figure 3. Two partitioning strategies that load-balance the number of non-zeros iterated across 4 parallel threads. Colors denote partition membership, where gray non-zeros are not part of any partition. Due to the hierarchical skipping in **(b)**, it is impossible to load balance iterated non-zeros without knowing which rows will be skipped versus iterated.

3 Partitioning Algorithm

We now discuss a generalized framework and search procedure for computing partitions that scales to sparse tensor algebra expressions that contain any number of tensors, any number of dimensions, and any combination of unions and intersections. This search procedure is defined with respect to abstract cost functions that model the amount of work performed within each partition; the definitions of these functions are specialized to the concrete tensor algebra expression and sparse data structure, and are described in Section 4. The produced partitions are non-overlapping ranges of coordinates that can be processed in parallel: this requires following a lexicographic ordering, the ordering of loop nest variables. We first describe the class of cost models supported by our search procedure in Section 3.1. Then, in Section 3.2, we describe a hierarchical search strategy that can compute partitions for tensor algebra expressions with any number of dimensions and union iteration patterns between operands. Finally, in Section 3.3, we introduce a recursive variant of the search that progressively computes partitions for nested intersection iteration patterns, which require a different strategy due to hierarchical skipping.

3.1 Cost Functions

A candidate sparse tensor algebra expression is defined over a d-dimensional iteration space $\mathcal{I}$. We assume these dimensions are ordered from 0 to $d-1$ by some loop ordering $\mathcal{L}$. We define a class of cost functions $\mathbf{C}_m$ for each dimension m in $\mathcal{I}$. Let $\mathbf{N}_m$ refer to the extent of dimension m in $\mathcal{I}$. Then, $\mathbf{C}_m(x \mid x_0, \ldots, x_{m-1})$ models the cost of coiterating lexicographically with respect to $\mathcal{L}$ between the d-dimensional coordinates $(x_0, \ldots, x_{m-1}, 0, 0, \ldots, 0)$ and $(x_0, \ldots, x_{m-1}, x-1, \mathbf{N}_{m+1}, \ldots, \mathbf{N}_{d-1})$. Intuitively, this subspace of $\mathcal{I}$ fixes coordinates for the 0 to $m-1$ dimensions of $\mathcal{I}$, limits the space of coordinates for dimension m to be within $[0, x)$ and spans the full extent of all remaining dimensions $m+1$ to $d-1$.

Various subspaces defined by $\mathbf{C}_m$ are shown in Figure 4. $\mathbf{C}_i(i_1)$ refers to all iteration space points with the first coordinate in $[i_0, i_1)$, which is just coordinates starting with i_0. Then, $\mathbf{C}_j(j_2 \mid i_1)$ refers to all iteration space points with i_1 as the i coordinate and j coordinates within $[0, j_2)$. Note that points with i coordinate as i_0 are excluded. Likewise, $\mathbf{C}_k(k_1 \mid i_1, j_2)$ is just the singular coordinate (i_1, j_2, k_0), as $[k_0, k_1) = k_0$.

We require two properties of cost functions for use in our search algorithm: *monotonicity* and *hierarchical consistency*:

Monotonicity: Cost functions must be monotone:

$$\mathbf{C}_m(x \mid x_0, \ldots, x_{m-1}) \leq \mathbf{C}_m(x+1 \mid x_0, \ldots, x_{m-1})$$

Hierarchical Consistency: Cost functions must hierarchically compose, meaning the cost at a coordinate must equal the cost to fully coiterate the dimension after it. Specifically, $\forall\, x_m \in [0, \mathbf{N}_m)$:

$$\mathbf{C}_m(x_m+1 \mid x_0, \ldots, x_{m-1}) = \mathbf{C}_m(x_m \mid x_0, \ldots, x_{m-1}) + \mathbf{C}_{m+1}(N_{m+1} \mid x_0, \ldots, x_{m-1}, x_m)$$

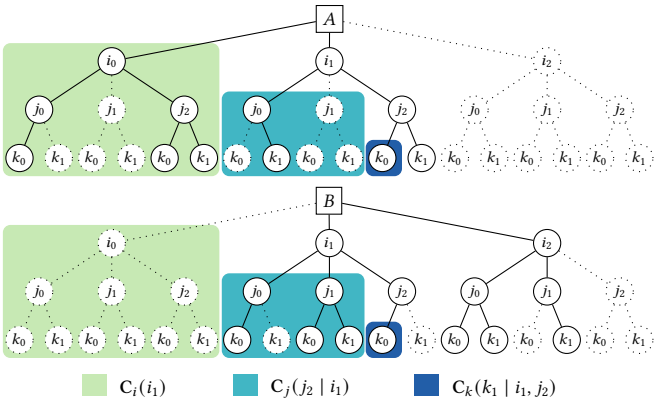


Figure 4. Iteration sub-spaces enclosed by different cost functions over two $3 \times 3 \times 2$ sparse tensors A and B. Dotted coordinates are 0-valued and are not explicitly stored.

3.2 Hierarchical Search Partitioning

We now describe a partitioning algorithm that leverages these cost functions to find load-balanced partitions of sparse tensor algebra operations over iteration spaces with arbitrary dimensionality. Our algorithm generalizes a one-dimensional binary search into a *hierarchical* binary search with iterative (per-loop) partitioning. The partitioning algorithm is shown in Algorithm 1, which searches for the multi-dimensional coordinate that incurs a cost of $\mathcal{Q}$. The hierarchical consistency property of cost functions enables the hierarchical search to be broken into a search at each iteration space dimension. Likewise, monotonicity enables binary search within each dimension. Algorithm 1 is used to construct a partition for each parallel processor by invoking it with a different value of $\mathcal{Q}$ for each processor. Concretely, given total cost of $\mathcal{Q}^*$, processor $p \in [0, \mathcal{P})$ can discover partition boundaries using Algorithm 1 by searching for $p \cdot \frac{\mathcal{Q}^*}{\mathcal{P}}$ as the partition lower-bound and $(p+1) \cdot \frac{\mathcal{Q}^*}{\mathcal{P}}$ as the partition upper bound.

Algorithm 1 Hierarchical Search Partitioning Algorithm

```
1:  Input: Loop order L, Cost functions C_m for m ∈ L, Queried
    Cost Q, Coordinate dimensions N
2:  Output: Coordinate x̄, tuple of per-dimension coordinates
3:  function FindPartition(L, C, Q, N)
4:      x̄ = (), R = Q   ▷ Init current coordinate and residual cost.
5:      for m in L do   ▷ Loop over each iteration space dimension.
6:          ▷ Binary search x_m ∈ [0, N_m) with C_m(x_m | x̄, ) ≤ R.
7:          x_m = HighestCoordinateLeqQuery(N_m, R, C_m, x̄)
8:          R ← R − C_m(x_m | x̄)
9:          x̄ ← x̄ :: x_m
10:     return x̄
```

Algorithm 1 restricts the search for partition boundaries from the full d-dimensional iteration space to increasingly smaller sub-spaces by fixing coordinates along each dimension. This is done by iterating over the desired loop order and computing the highest coordinate that has a cost less than or equal to the running query cost, $\mathcal{R}$. The coordinate is included as a component of the result ($\bar{x}$), and its dimension's cost is subtracted from the running query. This pattern maintains the invariant that $\mathcal{R}$ corresponds to the remaining query cost within the restricted subspace defined by the partially formed coordinate $\bar{x}$. At the final level, our algorithm yields a coordinate tuple that precisely identifies a partition boundary enclosing the queried cost $\mathcal{Q}$.

3.2.1 Asymptotics. Analysis of Algorithm 1 shows that it is $O\left(\sum_{l\in\mathcal{L}} \log(|l|)P_l\right)$, where $O(P_l)$ is the asymptotic complexity of evaluating the cost function at level l. With $n = \max_{l\in\mathcal{L}}(|l|)$ and $O(\mathcal{W}) = \max_{l\in\mathcal{L}}(P_l)$ for some $\mathcal{W}$, this runtime simplifies to $O(|\mathcal{L}|\mathcal{W}\log(n))$. Our implementations of cost functions (Section 4.2.3), are logarithmic in the number of non-zero elements and $|\mathcal{L}|$ is a constant for a given kernel, so the runtime is bounded by $O(\log(nnz)\log(n))$.

3.2.2 Tightness of Partitioning.

Theorem 3.1. *Let Δ_m denote the maximum cost difference between adjacent coordinates at loop level m:*

$$\Delta_m = \max_{\forall \bar{x}_m, 0\le x_m < N_m} (\mathbf{C}_m(x_m + 1 \mid \bar{x}_m) - \mathbf{C}_m(x_m \mid \bar{x}_m))$$

Then for any cost query, Algorithm 1 returns $\bar{x} = (x_0, \ldots, x_{d-1})$ such that $0 \le \mathcal{Q} - \mathbf{C}(\bar{x}) < \Delta_{d-1}$ where $\mathbf{C}(\bar{x})$ is the cost of coiteration from coordinate $(0, \ldots, 0)$ to $(x_0, \ldots, x_{d-1})$:

$$\mathbf{C}(\bar{x}) = \sum_{m=0}^{d-1} \mathbf{C}_m(x_m \mid x_0, \ldots, x_{m-1}).$$

Proof. By induction on m. □

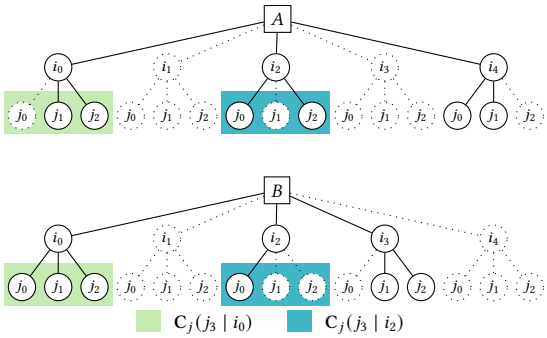
Theorem 3.1 implies that for a partition defined by lower and upper bound cost query values $\mathcal{Q}_l$ and $\mathcal{Q}_u$, Algorithm 1 returns coordinates $\bar{x}_l$ and $\bar{x}_u$ such that the cost of coiteration from $\bar{x}_l$ to $\bar{x}_u$ lies in the range $\mathcal{Q}_u - \mathcal{Q}_l \pm \Delta_{d-1}$. Consequently, partitions constructed from equally spaced query values incur a maximum load imbalance of at most $2 \cdot \Delta_{d-1}$. Furthermore, our implementation of $\mathbf{C}$ (Section 4.2.3) ensures that Δ_{d-1} is small (bounded by the number of tensors being coiterated), thereby guaranteeing that parallel processors execute approximately equal amounts of work.

3.3 Recursive Partitioning

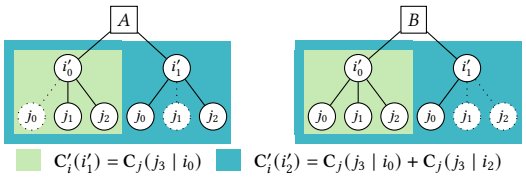
Section 3.2 describes how to compute partitions that balance the number of non-zeros processed in each partition equally across parallel workers. However, the hierarchical algorithm is insufficient for partitioning nested intersections on sparse tensors. This is because the hierarchical search algorithm and cost functions assume that all non-zeros within a candidate coordinate range are traversed, which is true only when unions are being computed or if a sparse intersection is performed only in the innermost loop. When iteration over nested sparse iterations is required, such as in the Hadamard product of DCSR matrices (Listing 3), entire segments of the iteration space are skipped when higher-order coordinates in an intersected dimension do not match, as illustrated in Figure 3b. Nested sparse intersections require changing how costs are computed to exclude the cost of hierarchically skipped portions of the iteration space.

Because the exact coordinates that are skipped during iteration with intersections are only known dynamically, intersections in outer loops must be computed first to discover the corresponding costs of intersections within inner loops. Therefore, we propose an iterative approach that progressively computes new cost functions on the remaining portions of the coordinate space after discovering which coordinates can be skipped. Furthermore, these functions can be computed while coiterating through each sparse intersection in a load-balanced manner.

3.3.1 Remapped Cost Functions. Nacho reasons about hierarchical skipping by *remapping* cost functions into cost functions that are aware of intersection's effects on coiteration. A remapped cost function considers the cost of coiteration over a given subset of the coordinate space. For example, a remapped cost function for Listing 3 can be computed over



(a) The loop nest i, j contains a sparse intersection at outer level i. We apply recursive partitioning to map $i, \mathbf{C}_i$ to $i', \mathbf{C'}_i$.



(b) Modified loop nest i', j does not have hierarchical skipping. The remapped cost function $\mathbf{C'}_i$ is calculated from $\mathbf{C}_j$.

Figure 5. Recursive partitioning for Hadamard product.

Listing 4. Computing $Z_{ij} = A_{ij} \cdot B_{ij}$ on DCSR matrices via multiple load-balanced kernels.

```
// Load balance on just loop i.
while i ← rows(A) ∩ rows(B):
  T[i] = C_j(N_j | i)
T' = exclusive_prefix_sum(T)
let C'_i(x_i) = T'[x_i]
// Load balance using C'_i and C_j.
for i ← i_T:
  while j ← cols(A[i]) ∩ cols(B[i]):
    Z[i,j] = A[i,j] * B[i,j]
```

the valid i's that are in $\texttt{rows}(A) \cap \texttt{rows}(B)$, where $\mathbb{1}_p$ denotes the indicator function over the predicate p:

$$\mathrm{C}'_i(x_i) = \sum_{r=0}^{x_i} \mathrm{C}_j(N_j \mid r) \cdot \mathbb{1}_{r \in \texttt{rows}(A) \cap \texttt{rows}(B)}$$

C'_i can be used to load-balance Listing 3 despite the hierarchical skipping: it only considers the cost of coordinate sub-spaces that must be coiterated over after the outermost intersection, precisely describing the costs of iteration in the inner loop, as illustrated in Figure 5. More generally, we define a remapped cost function that considers iteration over a subset of the dimension m given by the set I:

$$\mathrm{C}'_m(x_m \mid \bar{x}_m) = \sum_{r=0}^{x_m} \mathrm{C}_{m+1}(N_{m+1} \mid \bar{x}_m, r) \cdot \mathbb{1}_{r \in I}$$

3.3.2 Efficiently Computing Remapped Cost Functions.

The sum in a remapped cost function over the subset I can be memoized by iterating I and independently computing the cost function C_{m+1} for each valid coordinate, then taking the prefix sum of the stored values. For example, Listing 3 is rewritten into Listing 4, where each loop is load-balanced. The prefix sum can be computed efficiently in parallel [8, 42].

3.3.3 Applying Recursive Partitioning.

The structure of recursive partitioning is given in Algorithm 2, where the base case (Algorithm 1) is applied if a loop nest contains no sparse intersections or only an innermost sparse intersection. If an outer sparse intersection is found, the body of the sparse intersection is replaced with the computation of the remapped cost function. This modified loop nest is partitioned and computed in parallel, giving a precise cost model for the partitioning of the body of the loop. This recursive structure results in a series of load-balanced parallel loops.

4 Code Generation

Nacho is implemented as an extension of the Taco compiler that uses the partitioning algorithms in Section 3 to efficiently parallelize coiterative loops. Our points of extension are illustrated in Figure 6. Nacho generates three parallel kernels per load-balanced loop nest: a *partitioning* kernel, an *assembly* kernel that allocates the data structures for sparse result tensors, and a *compute* kernel that fills in the sparse output allocated by the assembly kernel. Both the assembly and compute kernels use the partitions computed by the partitioning kernel. We primarily build off of Taco [14, 15, 36] and provide the relevant background to understand our contributions. We refer the reader to these works or to Kjølstad [37] for more thorough descriptions of Taco.

We first briefly describe the requirements of a parallel backend, and provide our technique for generating parallel partitioning kernels in Section 4.2 with our implementation of data-structure-specific cost functions in Section 4.2.3. Lastly, we show how the computed partitions are used for both the assembly and compute phases in Sections 4.3 and 4.4.

Algorithm 2 Recursive Partitioning Algorithm

```
Input: Loop order L, Cost Functions C_m for m ∈ L, Cost Q
Output: Coordinate x̄, tuple of per-dimension coordinates
function FindRecursivePartition(L, C, Q)
    m = FindOutermostSparseIntersection(L)
    if m is not null and m is not innermost loop then
        ▷ Hierarchical skipping, remap cost function.
        C'_m, m' = MapIntersectionToNewSpace(C_m, m)
        C[m] = C'_m, L[m] = m'
        return FindRecursivePartition(L, C, Q)
    else
        ▷ No hierarchical skipping, do standard partitioning.
        return FindPartition(L, C, Q)
```

4.1 Backend-Specific Code Generation

In the following sections we describe a parallel backend assuming a shared-memory architecture and five capabilities: parallel loops, bulk memory allocation, prefix sums, and for scatter kernels (see Section 4.3), sorting and segmented summation. These are standard capabilities provided by vendor libraries [11, 30]. For segmented summation, we use a custom load-balanced implementation based on moderngpu [5]. We

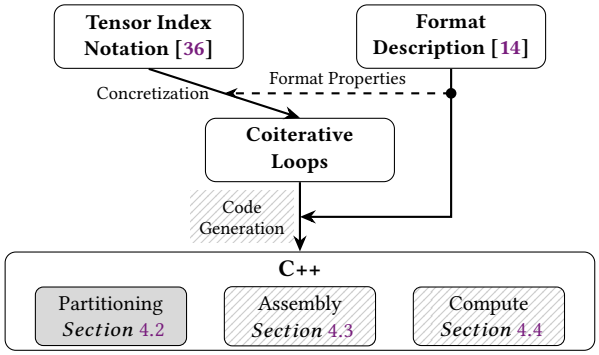


Figure 6. Nacho code generation. Gray boxes denote modifications to prior work in Taco. We intercept the lowering of coiterative loops to generate partition functions and modify the iteration bounds used in compute and assembly stages.

believe Nacho can be extended to other parallel backends that support these capabilities.

4.2 Parallel Partitioning

To produce a partitioning kernel, Nacho specializes Algorithm 1 with a concrete loop order (selected with the algorithm of Yan et al. [67]) and cost functions generated with respect to the data structure formats of the input tensors. Specifically, we implement cost functions as the sum of the number of non-zero values contained by each tensor operand up to the query coordinate. This choice of cost function satisfies both the monotonicity and consistency properties described in Section 3.1 as requirements for a cost function, and is a good proxy for work, as it closely corresponds to the number of memory reads per thread. The number of non-zeros up to a coordinate for a specific operand is computed via data-structure-specific queries, which we describe in Section 4.2.3. As part of specialization, we perform an optimization to search over sparse tensors and store their partitions by using the *position* space (iterators into sparse data structures), even though the searches are logically over the *coordinate* space. We first provide a brief background on Taco formats, as they relate to our code generation.

4.2.1 Background: Taco Formats. Taco decouples compute specifications from a *format* language that describes the data structure used to store each tensor dimension [14]. For example, the Compressed Sparse Row (CSR) [61] format is expressed as a Dense-Compressed row-major matrix. Chou et al. [14, Section 2] surveys tensor storage formats.

4.2.2 Nacho Formats. Nacho supports the Compressed, Dense, and Singleton formats. These are *sorted* formats, which enable Nacho to perform binary searches over the data structures. We believe that our choice of cost function is applicable to a wider variety of formats, such as those described by Ahrens et al. [1], as long as those formats support the efficient calculation of the number of non-zero values within a range of coordinates. Note that unsorted formats (such as hash-based or unsorted COO) that do not enable coiteration as described in Section 2.1 can still be partitioned via Nacho's technique, but the partitioning step is not efficient due to the required linear scans per cost model query instead of binary searches. Additionally, iterating data structures in *discordant* order, for example, iterating a CSR by columns before rows (CSR is a row-major format), is not supported by Taco, and therefore also not supported by Nacho.

4.2.3 Format-Specific Cost Functions. Our cost function implementations use the sum of the number of non-zeros contained in each sparse tensor within the queried coordinate subspace, which satisfies the properties defined in Section 3.1. This approximates the runtime of each partition, but future work could consider cardinality estimators [24, 52, 62] or a finer-grained estimation of memory traffic.

Listing 5. Generated cost functions for CSR matrix A_{ij} over loop nest $i \rightarrow j$.

```
int C_i(int x_i, CSR A) {
 return A.pos[x_i] - A.pos[0];
}
int C_j(int x_i, int x_jp, CSR A) {
 return x_jp - A.pos[x_i];
}
```

Listing 6. Generated cost functions for CSR matrix A_{ij} over loop nest $i \rightarrow k \rightarrow j$. A is a broadcasted over level k.

```
int C_i(int x_i, CSR A) {
 return (A.pos[x_i] - A.pos[0]) * N_k;
}
int C_k(int x_i, int x_k, CSR A) {
 return (A.pos[x_i+1] - A.pos[x_i]) * x_k
}
int C_j(int x_i, int x_k, int x_jp, CSR A) {
 return x_jp - A.pos[x_i]; // Same as Listing 5
}
```

Nacho generates a cost function for each level of each sparse tensor in the compiled tensor expression. Our code generation uses the query scheme described by Chou et al. [15] to generate format-specific implementations to count the number of non-zeros included within the cost function query. Instead of accepting coordinates, the cost functions instead accept *iterators* into the dimensions of sparse tensors that reference the physical locations within the sparse data structures referring to a logical coordinate. Listing 5 shows examples for the CSR data structure: the cost function for the outer dense level returns the number of non-zeros in the level below it, while the cost function for the compressed inner level subtracts the offset of the start of that row from the iterator that corresponds to the coordinate.

When an iteration space dimension does not appear in a tensor's index expression, such as the reference to x_j when iterating over the i dimension of the expression $y_i = A_{ij} \cdot x_j$, we refer to the tensor as *broadcasted* over that dimension. Broadcasted dimensions induce repeated coiteration over the tensor, which affects cost estimation. As illustrated in Listing 6, Nacho generates cost functions for broadcasted dimensions that scale the cost by the appropriate factor.

4.2.4 Specializing the Search for Partitions. Nacho generates a specialized implementation of Algorithm 1 for the target expression and sparse tensor data structures. Nacho inlines cost function invocations and unrolls the main loop over the ordering of iteration space dimensions. The result of specializing Algorithm 1 for the three-way sparse vector co-iteration example (Figure 2) is shown in Listing 7. The generated code contains an outer binary search (Line 5) for the i coordinate with cost less than the input query q. Each step of the outer binary search then invokes the cost function for each sparse vector, which results in inner binary searches over the non-zero elements in each sparse vector (lines 7—12). Nacho also tracks the searched position ranges

Listing 7. Algorithm 1 specialized to coiteration of three sparse vectors, which generates the partitions in Figure 2.

```
1  int partition(SpVec A, SpVec B, SpVec C, int q,
2                int N, Parts &p) {
3   int low = 0, high = N, lA = 0, lB = 0, lC = 0;
4   int hA = A.nnz-1, hB = B.nnz-1, hC = C.nnz-1;
5   while (low < high) {
6    int i = low + (high - low + 1) / 2;
7    // Find least ipA in [lA, hA] s.t. A[ipA] >= i.
8    int ipA = lb_search(A, i, lA, hA);
9    int ipB = lb_search(B, i, lB, hB);
10   int ipC = lb_search(C, i, lC, hC);
11   // Compute cost = C_i(i,A) + C_i(i,B) + C_i(i,C).
12   int cost = ipA + ipB + ipC;
13   if (cost <= q)
14    low = i, lA = ipA, lB = ipB, lC = ipC;
15   else
16    high = i - 1, hA = ipA, hB = ipB, hC = ipC;
17  }
18  p.ipA = lA, p.ipB = lB, p.ipC = lC;
19  return low;
20 }
```

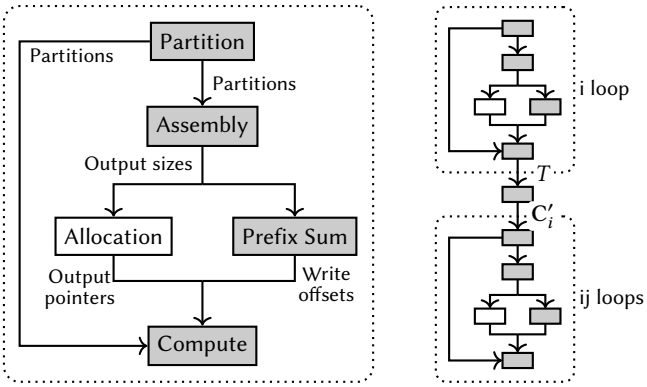
within each compressed data structure to limit the range of positions searched in each inner binary search as the outer binary search progresses (lines 13—16). Finally, Nacho stores the results of the partitioning search so that the results can be reused in the assembly and compute stages (line 18).

4.2.5 Partitioning Kernel per Sparse Intersection.

A partitioning kernel is generated once per loop nest isolated by Algorithm 2. For example, a CSR element-wise operation produces the kernels illustrated in Figure 7a, but a DCSR element-wise multiplication (nested sparse intersections, e.g., Listing 3), produces two partitioning kernels, one for load balancing each sparse intersection. We illustrate the flow of kernels for DCSR element-wise multiplication in Figure 7b. This load balances each stage of Listing 4: the first partitioning kernel load balances the outer intersection (Lines 2–3), and produces partitions for the assembly and compute kernels to iterate over the outer intersection and compute the remapped cost function. The second partitioning kernel uses the remapped cost function (Lines 4–5) to load balance the inner intersection (Lines 7–9), producing partitions for the final assembly and compute functions.

4.3 Parallel Assembly

We follow the approach of prior work [14, 36, 53, 66] for computing the size of sparse output dimensions: an *assembly* phase symbolically executes the sparse kernel to find allocation sizes and write offsets. The prefix sum of the thread-local counts computes write offsets for the *compute* stage, which computes and writes the result. This is a standard parallel programming practice for writing to a sparse output without additional synchronization [8]. Because the partitions computed by Nacho are irregular and not necessarily aligned along each dimension, a slight modification to this strategy is required for hierarchical writes (e.g., writes to the row offsets vector of a CSR matrix). To guard against duplicate



(a) Sparse union or innermost-loop sparse intersection. **(b)** DCSR multiplication (e.g., Listing 4).

Figure 7. Kernels for single and nested intersections. **(b)** connects two sets of kernels load balanced by the strategy in **(a)** with a prefix sum in between. Gray kernels are parallel.

counts and writes, only the parallel worker that finishes the iteration over a dimension performs the write or count.

Appends versus Scatters. This parallel-writing structure is sufficient for kernels that write to the output in the order of iteration, logically *appending* to the output [3]. *Scattering* kernels, which randomly insert into sparse outputs are incompatible with this approach, and efficiently supporting scatter patterns has been studied [25, 35, 65, 72]. However, the community has not developed a technique for *parallel* sparse scatters into arbitrary sparse output formats. Nacho instead takes the expand-sort-contract (ESC) [6, 16] approach, which does not fuse reductions when scattering into a sparse output. ESC instead materializes (expands) the full sparse output, sorts with respect to the reduction coordinate, and reduces (contracts) the sorted intermediate. The sort turns the reduction into a segmented reduction with an append-only output pattern, which can be parallelized via known techniques [5, 11]. Our load-balancing technique is only applied to the materialization of the temporary, as in Dalton et al. [16].

4.4 Parallel Compute

Taco *compute* kernels are modified similarly to assembly kernels: we adjust loop bounds to be based on stored partitions, perform writes via the offsets computed in the assembly phase, and guard higher-order writes to ensure only a single thread performs each write. Listing 8 illustrates these changes for a CSR Hadamard product: only loop bounds (Lines 3–9) and writes (Lines 1–2 and 21) were modified.

5 Evaluation

To show that Nacho effectively load balances across parallel processors, we provide evidence for the following claims:

1. Nacho load balances in the presence of skew.

Listing 8. Partitioned CSR Hadamard product, with the differences from Taco-generated code highlighted.

```
int jpZ = offsets[tid];
if (tid == 0) Z.pos[0] = 0;
for (int i = p.i[tid]; i <= p.i[tid+1]; i++) {
 bool first_i = (i == p.i[tid]);
 bool end_i = (i == p.i[tid+1]);
 int jpA = first_i ? p.jpA[tid] : A.pos[i];
 int jpB = first_i ? p.jpB[tid] : B.pos[i];
 int jpA_e = end_i ? p.jpA[tid+1] : A.pos[i+1];
 int jpB_e = end_i ? p.jpB[tid+1] : B.pos[i+1];
 while (jpA < jpA_e && jpB < jpB_e) {
  int jA = A.crd[jpA], jB = B.crd[jpB];
  int j = min(jA, jB);
  if (j == jA && j == jB) {
   Z.crd[jpZ] = j;
   Z.val[jpZ] = A.val[jpA] * B.val[jpB];
   jpZ++;
  }
  jpA += (j == jA); jpB += (j == jB);
 }
 // Write j offset when row is complete.
 if (jpA_e == A.pos[i+1] && jpB_e == B.pos[i+1])
  Z.pos[i+1] = jpZ;
}
```

2. Nacho achieves reasonable performance when compared to state-of-the-art handwritten kernels.
3. Generality across *data structures* improves performance.
4. Load-balancing fused expressions improves performance over load-balanced binary expressions.

Our results show that Nacho-generated kernels achieve parity with hand-parallelized kernels from Intel MKL [31], cuSPARSE [45], scheduled Taco [36, 53] kernels, and significantly outperform generic kernels and compound expressions from PyTorch Sparse [50]. Nacho takes, on average, 7ms to generate the C++/CUDA kernel for each benchmark.

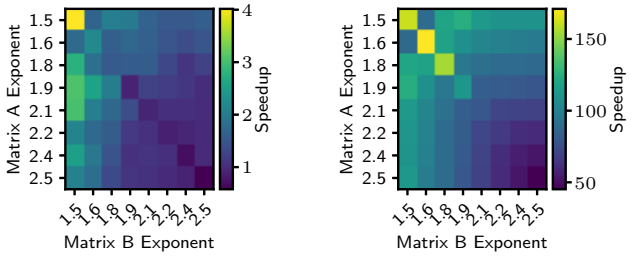
Methodology and Experimental Setup. We evaluate on a 24-core Intel i9-14900K, and a 24GB NVIDIA RTX 4090. Benchmarks are pinned to the 8 performance cores of the CPU using `numactl`. We used g++ 11.4.0 and CUDA 12.6, and compiled with `-O3`. Each result was collected by performing a warm-up run followed by the mean of 14 runs, where the 2 fastest and slowest runs were excluded. We evaluate on real-world tensors from SuiteSparse [18] and FROSTT [56], in addition to synthetic tensors. Benchmarks clarify which datasets they are evaluated on. Reported runtimes are end-to-end for all systems, including analysis phases (partitioning and precompute/symbolic phases), and output allocations, as well as the format conversions performed by PyTorch Sparse for some operations.

Overhead of Nacho Partitioning. Nacho's partitioning introduces low overhead. When evaluating CSR addition on the GPU across the SuiteSparse dataset, partitioning constitutes an average of 1% of the overall execution time (maximum 2.5%). In contrast, recursive partitioning for the SpGEMM kernel on the GPU constitutes an average of 7.9% of the total runtime (maximum 34.2%). Notably, this overhead includes the computation of the outer intersection required for the remapping step of our recursive partitioning algorithm, which is unavoidable even if recursive partitioning is not being used. Excluding this computation, the runtime overhead for just the two partitioning kernels in recursive partitioning for SpGEMM has an average of 2.7% (maximum 11.9%). All presented runtimes include partitioning time.

5.1 Nacho Load Balances in the Presence of Skew

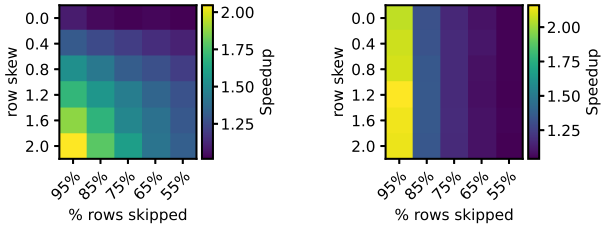
Figure 8a plots the speedup Nacho achieves over Taco's CPU-parallel CSR addition on skewed power-law matrices, which have heavy imbalance of nonzeros between rows. Taco parallelizes the outer for-loop of the addition, and uses work-stealing [9] to dynamically load-balance; with higher skew, Nacho achieves better load balance by creating equally-sized work items in the first place. Figure 8b performs the same evaluation on the GPU, where Nacho achieves significantly higher speedups over Taco, which assigns each row of the addition to a GPU thread. As the GPU has orders of magnitude more parallelism to saturate than the CPU, Nacho's load-balancing is critical for performance.

Next, we evaluate the impact of recursive partitioning on partition quality. Figure 9 shows the speedup of using recursive partitioning over using only hierarchical partitioning on the element-wise multiplication of two DCSR matrices. When the outer intersection results in many rows being



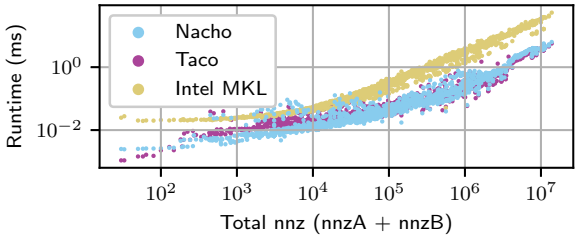
(a) CPU results. **(b)** GPU results.

Figure 8. Speedup over Taco of CSR addition ($A + B$) on synthetic power-law matrices of different exponents of size 10000×1000000. Lower exponents indicate higher skew.

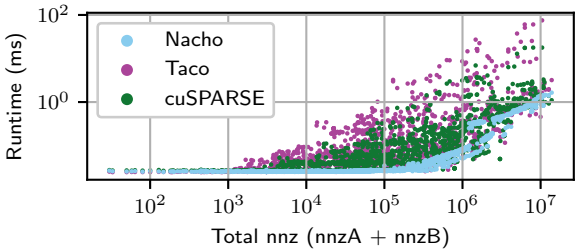


(a) CPU results. **(b)** GPU results.

Figure 9. Speed-up of DCSR Hadamard product on synthetic 100000×100000 matrices from using recursive partitioning (Algorithm 2) over hierarchical partitioning (Algorithm 1).

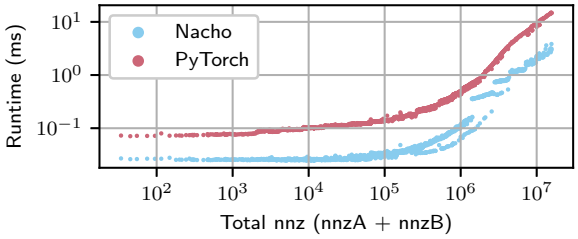


(a) CPU, geo-mean speedup: 1.2× (Taco) and 5.5× (Intel MKL).

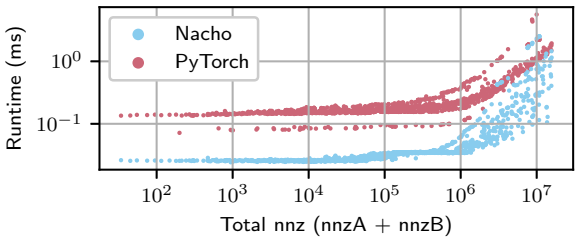


(b) GPU, geo-mean speedup: 2.4× (Taco) and 1.8× (cuSPARSE).

Figure 10. CSR addition on pairs of SuiteSparse matrices.



(a) COO addition, geo-mean speedup: 4.1×.



(b) COO Hadamard product, geo-mean speedup: 5.3×.

Figure 11. GPU kernels on pairs of SuiteSparse matrices.

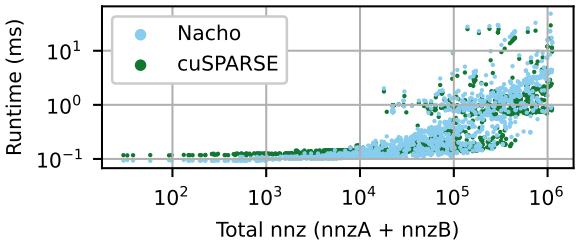


Figure 12. SpGEMM CSR × CSR, geo-mean speedup: 0.93×.

skipped, the recursive partitioning approach yields significant speedups by constructing partitions that better align with the true iteration costs. On our 8-core CPU, each hierarchical partition contains many (potentially skewed heavy) rows; when intersection iteration skips these rows, these partitions diverge from the true cost computed by our recursive partitioning algorithm and yield higher speedup. This effect is mitigated on the GPU, as Nacho computes fine-grained partitions for each thread. Heavy rows are already split across multiple threads, causing the observed speedup to vary only with the percent of skipped rows.

5.2 Performance Compared to Handwritten Kernels

We show that Nacho-generated code matches the performance of hand-written kernels by comparing it to hand-optimized parallel kernels in PyTorch Sparse [50], which uses Intel MKL [31] and cuSPARSE [45] when possible. For operations unsupported by vendor libraries, PyTorch converts to COO and dispatches to hand-written COO kernels.

We compare Nacho with these vendor libraries for CSR addition in Figures 10a and 10b. Nacho achieves a geomean speed-up of 5.5× over Intel MKL (between 0.38–56×), and a geomean speed-up of 1.8× over cuSPARSE (between 0.13–54×). Nacho is faster than Intel MKL on 98% of matrices and faster than cuSPARSE on 92% of matrices.

The cuSPARSE library does not support COO element-wise operations, so PyTorch's GPU backend uses custom implementations that do not perform coiteration and are thus asymptotically inefficient. Our evaluation in Figure 11 shows this: Nacho achieves geo-mean speedups of 4.1× and 5.3× for element-wise addition and multiplication, respectively.

Lastly, we compare against cuSPARSE on sparse matrix-matrix multiplication (using the default settings) on SuiteSparse matrices in Figure 12 and see a geo-mean slowdown of 1.08×. Based on symbols extracted from profiles, we believe this slowdown is due to cuSPARSE's use of hash-based accumulators [44] as opposed to our ESC-based reduction [6], which materializes the intermediate temporary before sorting and reducing. We believe these results warrant investigation into parallelized versions of Zhang et al. [72] to improve the performance of scatter-based kernels.

5.3 Generality Across Data Structures

To demonstrate the benefit of code specialized to particular data structures, we evaluate operations on sparse data structures that existing systems do not support. First, we evaluate third-order tensor addition on tensors from the FROSTT [56] dataset that fit in GPU memory in Table 1. Nacho's COO already outperforms PyTorch Sparse's COO on both CPU and GPU: PyTorch's CPU implementation is unparallelized, and its GPU implementation is asymptotically inefficient, as it does not perform coiteration. Nacho's parallel CPU kernel even surpasses PyTorch's GPU results. Supporting

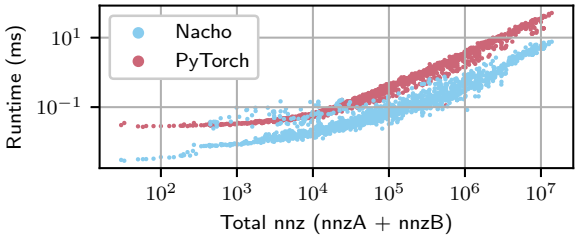
Table 1. Runtimes (ms) of third-order tensor addition on FROSTT [56] tensors. PyTorch (PT) uses COO and has integer overflow errors on nell-1.

	CPU			GPU		
Tensor	**CSF**	**COO**	**PT**	**CSF**	**COO**	**PT**
darpa	**17.78**	47.04	1013.75	**5.75**	10.32	69.34
nell-2	**50.02**	130.65	2687.37	**10.34**	25.15	331.29
fb-m	195.24	**181.55**	3674.43	**40.83**	84.02	OOM
nell-1	**114.49**	246.62	Err	**24.93**	63.52	Err

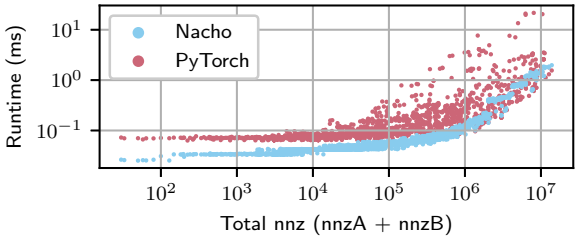
CSF yields further gains: Nacho's CSF kernel outperforms Nacho's COO kernel except one case where it is close to parity, demonstrating the concrete benefit of specializing to a more compressed format. This performance boost is likely due to reduced memory traffic in loading the CSF tensors as opposed to the COO tensors.

Next , we evaluate element-wise addition of a COO matrix with a CSR matrix in Figure 13. Nacho outperforms PyTorch, which converts the CSR operand to COO before performing the (non-coiterative) COO addition. Nacho's speedup comes from asymptotically efficient coiteration-based kernels and by avoiding the unnecessary format conversion.

Finally, we evaluate CSR element-wise multiplication in Figure 14. This operation is not supported by Intel MKL or cuSPARSE, so PyTorch implements it with a conversion to COO and an asymptotically inefficient, general-purpose kernel. Nacho again achieves significant speedup by avoiding conversions and generating coiterative code.

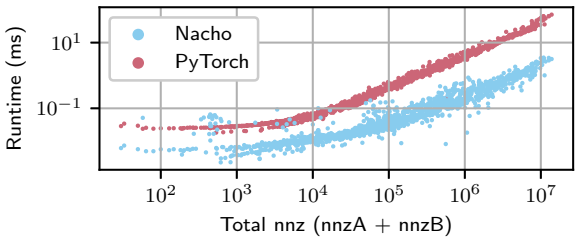


(a) CPU results, geo-mean speedup 4.0×.

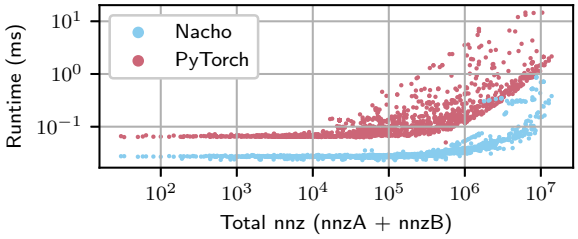


(b) GPU results, geo-mean speedup 2.0×.

Figure 13. COO matrix added to CSR matrix on SuiteSparse.

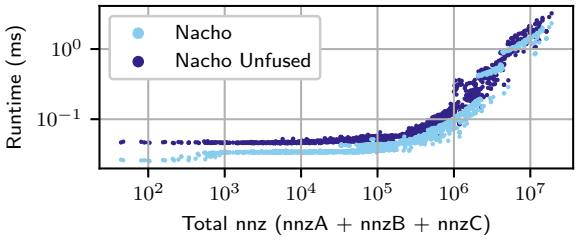


(a) CPU results, geo-mean speedup 9.1×.

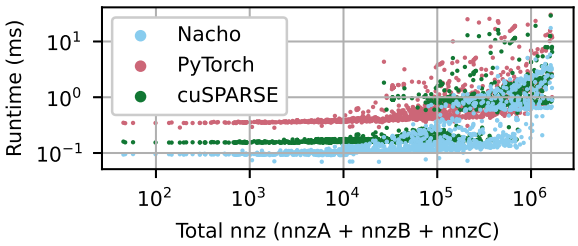


(b) GPU results, geo-mean speedup 3.8×.

Figure 14. CSR Hadamard product on SuiteSparse matrices.



(a) Three-way CSR ⊕, geo-mean speedup: 1.4× over unfused.



(b) Sampled SpGEMM (SSSMM), geo-mean speedup: 3.2× over PyTorch and 1.8× over cuSPARSE with Nacho's ⊙.

Figure 15. Fusion benefits across SuiteSparse matrices.

5.4 Load Balancing Fused Kernels

Because our partitioning algorithm is general across expressions, Nacho can extract both the constant-factor and asymptotic [36] improvements that come from fusing sparse tensor algebra operations into a single kernel. We show that Nacho can effectively load balance these fused kernels, allowing for additional speedup over parallelized separate kernels.

Figure 15a compares the performance of a fused three-way CSR addition with Nacho compared against an unfused, but parallelized, Nacho implementation (Figure 10b

demonstrates Nacho outperforms cuSPARSE as a baseline). The geo-mean speedup is 1.4× over the unfused kernels. Next, Figure 15b evaluates fused Sampled-SpGEMM (SSSMM) [43] over PyTorch, which uses both a cuSPARSE SpGEMM and element-wise multiplication, showing significant speedup. We additionally compare against an implementation that uses a cuSPARSE SpGEMM followed by a Nacho-generated element-wise multiplication, the best implementation of each binary operator. Despite Nacho's SpGEMM being slower than cuSPARSE (Figure 12), Nacho's fused SSSMM kernel outperforms the unfused cuSPARSE by 1.8× (geo-mean).

Lastly, we evaluate the third-order inner product [36] on FROSTT [56], on shifted versions of the same tensor in Table 2. PyTorch executes the element-wise multiplication and reduction as separate kernels, unlike Nacho. The relative performance of CSF and COO varies by tensor and backend. While CSF benefits from hierarchical skipping on some tensors, it incurs higher thread divergence on other tensors.

Table 2. Runtime (ms) of third-order inner product on FROSTT [56]. PyTorch (PT) uses COO, and has integer overflow errors on nell-1.

	CPU			GPU		
Tensor	**CSF**	**COO**	**PT**	**CSF**	**COO**	**PT**
darpa	**0.11**	11.16	116.949	**0.283**	0.861	5.525
nell-2	**0.30**	32.03	339.860	**0.177**	2.722	15.899
fb-m	74.99	**49.22**	681.892	**8.996**	10.749	20.411
nell-1	**13.16**	61.39	Err	50.363	**11.537**	Err

6 Related Work

Sparse Tensor Algebra Compilers. A foundational theory for sparse tensor algebra compilation was developed in the Taco compiler [36, 37], with extensions targeting different hardware [28, 29, 53, 66] and supporting a wider variety of tensor operations [27, 40, 51, 60]. These compilers expose parallelism through *scheduling languages* that describe execution strategies for nested loops [53, 66]. However, these scheduling languages only support parallel execution of dense loops or on tensor expressions that only contain a single sparse tensor. Automatic scheduling frameworks have been developed that automate other aspects of scheduling, such as loop ordering and temporary insertion [3, 19, 67]. Sparse tensor compilers for specific domains, such as sparse machine learning [70], support alternative parallelization strategies, but are also limited to tensor expressions that have a single sparse tensor. Compiling sparse tensor algebra to PyTorch [49] has been explored to improve performance of expressions that can utilize fixed-function units for matrix-multiplication on GPUs [64]. WingSpan [23], a concurrent work, extends Finch [1] with support for parallelization via explicit annotations on both the data structures and the loops over sparse data structures. Notably, WingSpan provides dependence analysis that enables sparse parallel workspaces to support scattering without intermediate materialization, but their parallelization scheme does not guarantee load balancing work. Finally, several approaches employ inspector-executor techniques to optimize sparse tensor operations [12, 13, 59, 74]. ParSy and Sympiler [12, 13] leverage compile-time analysis of tensor structures to generate parallelized or vectorized code. The Sparse Polyhedral Framework [59, 74] supports parallel iterate-locate kernels [74]. However, unlike Nacho, these frameworks do not support parallelizing general coiteration.

Merge Path Partitioning. A subtle but important difference between partitioning strategies for merge sort [22, 46] and partitioning strategies for sparse tensor algebra is that intersections and unions require matching coordinates across operands to be owned by the same partition [16, 47]. Other (non-coiterative) strategies for computing intersections and unions may be load balanced in other ways; we focus on a load-balancing strategy specifically for coiterative merge-based unions and intersections. This constraint motivates our algorithm searching *in coordinate space*, as opposed to position space (the space of the iterators into sparse operands).

Parallelizing Specific Sparse Kernels. Load-balanced parallel algorithms for sparse matrix operations such as sparse matrix-vector multiplication, sparse matrix addition, and sparse matrix-matrix multiplication were proposed by Dalton et al. [16]. These algorithms partition the merge path used to compute the union and intersection of non-zero sparse tensor coordinates, similar to parallel merging algorithms in the sorting community [22, 46]. We show a generalization of these algorithms to allow for load-balanced parallelization of any sparse tensor algebra expression. Significant work has gone into optimizing sparse matrix-matrix multiplication [17, 41], additionally focusing on parallelization of the construction of the sparse result.

Parallel Hardware For Sparsity. Various specialized hardware has been proposed to accelerate sparse tensor algebra [26, 48, 57, 58, 71]. Programmable dataflow hardware for arbitrary sparse tensor algebra expressions [29, 39] extract pipeline parallelism from coiteration and data parallelism from parallel outer loops. We believe the partitioning strategies presented in Nacho could be used by this hardware to achieve better load-balanced data-parallel execution.

7 Conclusion

We present a parallel partitioning algorithm that guarantees load balance for sparse tensor algebra with multiple sparse operands. Our implementation in Nacho specializes this partitioning algorithm to concrete sparse data structures and a loop order to generate efficient partitioning kernels for both CPUs and GPUs. We match hand-parallelized sparse tensor kernels while maintaining generality across data structures

and expressions. We believe our techniques can be extended to produce partitions for load-balanced execution on supercomputers and custom sparse accelerators, and to generate kernels for other irregular domains such as database queries.

Acknowledgments

We would like to thank (in no particular order): Aart Bik, Ben Driscoll, Chris Gyurgyik, Dalia Shaaban, Devanshu Ladsaria, Evan Williams, Gautham Ravipati, James Dong, Katherine Mohr, Michael Pellauer, Olivia Hsu, Rupanshu Soi, Shiv Sundram, and Shoaib Kamil, for their helpful feedback on this draft. This material is based upon work supported by the Defense Advanced Research Projects Agency (DARPA) under Contract HR0011-25-2-0039, as part of the Machine Learning and Optimization-guided Compilers for Heterogeneous Architectures (MOCHA) program. Alexander was supported by the Qualcomm Innovation Fellowship during this work.

References

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

[2] Willow Ahrens, Daniel Donenfeld, Fredrik Kjolstad, and Saman Amarasinghe. 2023. Looplets: A Language for Structured Coiteration. In *Proceedings of the 21st ACM/IEEE International Symposium on Code Generation and Optimization* (Montréal, QC, Canada) *(CGO '23)*. Association for Computing Machinery, New York, NY, USA, 41–54. doi:10.1145/3579990.3580020

[3] 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

[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] Sean Baxter. 2016. moderngpu. https://github.com/moderngpu/moderngpu. Version 2.0.0, released 2016-03-28.

[6] Nathan Bell, Steven Dalton, and Luke N. Olson. 2012. Exposing Fine-Grained Parallelism in Algebraic Multigrid Methods. *SIAM J. Sci. Comput.* 34, 4 (Jan. 2012), C123–C152. doi:10.1137/110838844

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

[8] Guy E. Blelloch. 1990. *Prefix Sums and Their Applications.* Technical Report CMU-CS-90-190. School of Computer Science, Carnegie Mellon University.

[9] Robert D. Blumofe and Charles E. Leiserson. 1999. Scheduling multithreaded computations by work stealing. *J. ACM* 46, 5 (Sept. 1999), 720–748. doi:10.1145/324133.324234

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

[11] CCCL Development Team. 2023. *CCCL: CUDA C++ Core Libraries.* https://github.com/NVIDIA/cccl

[12] Kazem Cheshmi, Shoaib Kamil, Michelle Mills Strout, and Maryam Mehri Dehnavi. 2017. Sympiler: transforming sparse matrix codes by decoupling symbolic analysis. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis* (Denver, Colorado) *(SC '17)*. Association for Computing Machinery, New York, NY, USA, Article 13, 13 pages. doi:10.1145/3126908.3126936

[13] Kazem Cheshmi, Shoaib Kamil, Michelle Mills Strout, and Maryam Mehri Dehnavi. 2018. ParSy: Inspection and Transformation of Sparse Matrix Computations for Parallelism. In *SC18: International Conference for High Performance Computing, Networking, Storage and Analysis*. 779–793. doi:10.1109/SC.2018.00065

[14] Stephen Chou, Fredrik Kjolstad, and Saman Amarasinghe. 2018. Format abstraction for sparse tensor algebra compilers. *Proc. ACM Program. Lang.* 2, OOPSLA, Article 123 (Oct. 2018), 30 pages. doi:10.1145/3276493

[15] Stephen Chou, Fredrik Kjolstad, and Saman Amarasinghe. 2020. Automatic generation of efficient sparse tensor format conversion routines. In *Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation* (London, UK) *(PLDI 2020)*. Association for Computing Machinery, New York, NY, USA, 823–838. doi:10.1145/3385412.3385963

[16] Steven Dalton, Sean Baxter, Duane Merrill, Luke N. Olson, and Michael Garland. 2015. Optimizing Sparse Matrix Operations on GPUs Using Merge Path. In *2015 IEEE International Parallel and Distributed Processing Symposium, IPDPS 2015, Hyderabad, India, May 25-29, 2015*. IEEE Computer Society, 407–416. doi:10.1109/IPDPS.2015.98

[17] Steven Dalton, Luke Olson, and Nathan Bell. 2015. Optimizing Sparse Matrix—Matrix Multiplication for the GPU. *ACM Trans. Math. Softw.* 41, 4, Article 25 (Oct. 2015), 20 pages. doi:10.1145/2699470

[18] Timothy A. Davis and Yifan Hu. 2011. The university of Florida sparse matrix collection. *ACM Trans. Math. Softw.* 38, 1, Article 1 (Dec. 2011), 25 pages. doi:10.1145/2049662.2049663

[19] Kyle Deeds, Willow Ahrens, Magdalena Balazinska, and Dan Suciu. 2025. Galley: Modern Query Optimization for Sparse Tensor Programs. *Proc. ACM Manag. Data* 3, 3, Article 164 (June 2025), 24 pages. doi:10.1145/3725301

[20] Jonathan Frankle and Michael Carbin. 2019. The Lottery Ticket Hypothesis: Finding Sparse, Trainable Neural Networks. arXiv:1803.03635 [cs.LG] https://arxiv.org/abs/1803.03635

[21] Mahdi Ghorbani, Emilien Bauer, Tobias Grosser, and Amir Shaikhha. 2025. Compressed and Parallelized Structured Tensor Algebra. *Proc. ACM Program. Lang.* 9, OOPSLA1, Article 141 (April 2025), 29 pages. doi:10.1145/3720506

[22] Oded Green, Robert McColl, and David A. Bader. 2012. GPU merge path: a GPU merging algorithm. In *Proceedings of the 26th ACM International Conference on Supercomputing* (San Servolo Island, Venice, Italy) *(ICS '12)*. Association for Computing Machinery, New York, NY, USA, 331–340. doi:10.1145/2304576.2304621

[23] Adrian Gushin, Sang Yoon Kim, and Willow Ahrens. 2026. WingSpan: Concurrency and Dependence for Sparse and Structured Tensor Compilers. arXiv:2606.20855 [cs.MS] https://arxiv.org/abs/2606.20855

[24] Peter J. Haas, Jeffrey F. Naughton, S. Seshadri, and Arun N. Swami. 1996. Selectivity and Cost Estimation for Joins Based on Random Sampling. *J. Comput. Syst. Sci.* 52, 3 (June 1996), 550–569. doi:10.1006/jcss.1996.0041

[25] Shideh Hashemian, Michael F. P. O'Boyle, and Amir Shaikhha. 2026. Optimizing Sparse Tensor Compilation for Sparse Output. In *Proceedings of the 35th ACM SIGPLAN International Conference on Compiler Construction* (Sydney, NSW, Australia) *(CC '26)*. Association for Computing Machinery, New York, NY, USA, 27–39. doi:10.1145/3771775.3786267

[26] 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* (Columbus, OH, USA) *(MICRO-52)*. Association for Computing Machinery, New York, NY, USA, 319–333. doi:10.1145/3352460.3358275

[27] Rawn Henry, Olivia Hsu, Rohan Yadav, Stephen Chou, Kunle Olukotun, Saman Amarasinghe, and Fredrik Kjolstad. 2021. Compilation of sparse array programming models. *Proc. ACM Program. Lang.* 5, OOPSLA, Article 128 (Oct. 2021), 29 pages. doi:10.1145/3485505

[28] 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

[29] 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* (Vancouver, BC, Canada) *(ASPLOS 2023)*. Association for Computing Machinery, New York, NY, USA, 710–726. doi:10.1145/3582016.3582051

[30] Intel. 2026. Intel® OneAPI Threading Building Blocks (oneTBB). https://www.intel.com/content/www/us/en/developer/tools/oneapi/onetbb.html Accessed: 2026-04-13.

[31] Intel Corporation. [n. d.]. Intel oneAPI Math Kernel Library. https://www.intel.com/content/www/us/en/developer/tools/oneapi/onemkl.html. Accessed: 2026-04-05.

[32] Anirudh Jain, Pulkit Gupta, and Thomas M. Conte. 2025. RASSM: Residue-based Acceleration of Single Sparse Matrix Computation via Adaptive Tiling. In *Proceedings of the 30th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 1* (Rotterdam, Netherlands) *(ASPLOS '25)*. Association for Computing Machinery, New York, NY, USA, 907–923. doi:10.1145/3669940.3707219

[33] Wenzel Jakob. 2022. nanobind: tiny and efficient C++/Python bindings. https://github.com/wjakob/nanobind.

[34] Daniel Kats and Frederick R. Manby. 2013. Sparse tensor framework for implementation of general local correlation methods. *The Journal of Chemical Physics* 138, 14 (04 2013), 144101. doi:10.1063/1.4798940

[35] Fredrik Kjolstad, Willow Ahrens, Shoaib Kamil, and Saman Amarasinghe. 2019. Tensor algebra compilation with workspaces. In *Proceedings of the 2019 IEEE/ACM International Symposium on Code Generation and Optimization* (Washington, DC, USA) *(CGO 2019)*. IEEE Press, 180–192.

[36] Fredrik Kjolstad, Shoaib Kamil, Stephen Chou, David Lugato, and Saman Amarasinghe. 2017. The tensor algebra compiler. *Proc. ACM Program. Lang.* 1, OOPSLA, Article 77 (Oct. 2017), 29 pages. doi:10.1145/3133901

[37] Fredrik Berg Kjølstad. 2020. *Sparse Tensor Algebra Compilation*. PhD thesis. Massachusetts Institute of Technology, Cambridge, CA. Available at https://fredrikbk.com/publications/kjolstad-thesis.pdf.

[38] Joseph C Kolecki. 2002. *An Introduction to Tensors for Students of Physics and Engineering*. Technical Report NASA/TM-2002-211716. NASA Glenn Research Center. https://ntrs.nasa.gov/citations/20020083040

[39] Kalhan Koul, Olivia Hsu, Yuchen Mei, Sai Gautham Ravipati, Maxwell Strange, Jackson Melchert, Alex Carsello, Taeyoung Kong, Po-Han Chen, Huifeng Ke, Keyi Zhang, Qiaoyi Liu, Gedeon Nyengele, Zhouhua Xie, Akhilesh Balasingam, Jayashree Adivarahan, Ritvik Sharma, Christopher Torng, Joel S. Emer, Fredrik Kjolstad, Mark Horowitz, and Priyanka Raina. 2025. Onyx: A 12-nm Programmable Accelerator for Dense and Sparse Applications. *IEEE Journal of Solid-State Circuits* (2025), 1–13. doi:10.1109/JSSC.2025.3604724

[40] Peiming Liu, Alexander J Root, Anlun Xu, Yinying Li, Fredrik Kjolstad, and Aart J.C. Bik. 2024. Compiler Support for Sparse Tensor Convolutions. *Proc. ACM Program. Lang.* 8, OOPSLA2, Article 281 (Oct. 2024), 29 pages. doi:10.1145/3689721

[41] Weifeng Liu and Brian Vinter. 2014. An Efficient GPU General Sparse Matrix-Matrix Multiplication for Irregular Data. In *2014 IEEE 28th International Parallel and Distributed Processing Symposium*. 370–381. doi:10.1109/IPDPS.2014.47

[42] Duane Merrill and Michael Garland. 2016. Single-pass parallel prefix scan with decoupled look-back. *NVIDIA, Tech. Rep. NVR-2016-002* (2016).

[43] Srđan Milaković, Oguz Selvitopi, Israt Nisa, Zoran Budimlić, and Aydin Buluç. 2022. Parallel algorithms for masked sparse matrix-matrix products. In *Proceedings of the 27th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming* (Seoul, Republic of Korea) *(PPoPP '22)*. Association for Computing Machinery, New York, NY, USA, 453–454. doi:10.1145/3503221.3508430

[44] Yusuke Nagasaka, Akira Nukada, and Satoshi Matsuoka. 2017. High-Performance and Memory-Saving Sparse General Matrix-Matrix Multiplication for NVIDIA Pascal GPU. In *2017 46th International Conference on Parallel Processing (ICPP)*. 101–110. doi:10.1109/ICPP.2017.19

[45] NVIDIA Corporation. [n. d.]. cuSPARSE Library. https://docs.nvidia.com/cuda/cusparse/. Accessed: 2026-04-05.

[46] Saher Odeh, Oded Green, Zahi Mwassi, Oz Shmueli, and Yitzhak Birk. 2012. Merge Path - Parallel Merging Made Simple. In *2012 IEEE 26th International Parallel and Distributed Processing Symposium Workshops & PhD Forum*. 1611–1618. doi:10.1109/IPDPSW.2012.202

[47] Toluwanimi O. Odemuyiwa, Hadi Asghari-Moghaddam, Michael Pellauer, Kartik Hegde, Po-An Tsai, Neal C. Crago, Aamer Jaleel, John D. Owens, Edgar Solomonik, Joel S. Emer, and Christopher W. Fletcher. 2023. Accelerating Sparse Data Orchestration via Dynamic Reflexive Tiling. In *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 3* (Vancouver, BC, Canada) *(ASPLOS 2023)*. Association for Computing Machinery, New York, NY, USA, 18–32. doi:10.1145/3582016.3582064

[48] Subhankar Pal, Jonathan Beaumont, Dong-Hyeon Park, Aporva Amarnath, Siying Feng, Chaitali Chakrabarti, Hun-Seok Kim, David Blaauw, Trevor Mudge, and Ronald Dreslinski. 2018. OuterSPACE: An Outer Product Based Sparse Matrix Multiplication Accelerator. In *2018 IEEE International Symposium on High Performance Computer Architecture (HPCA)*. 724–736. doi:10.1109/HPCA.2018.00067

[49] Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, Alban Desmaison, Andreas Köpf, Edward Yang, Zach DeVito, Martin Raison, Alykhan Tejani, Sasank Chilamkurthy, Benoit Steiner, Lu Fang, Junjie Bai, and Soumith Chintala. 2019. PyTorch: An Imperative Style, High-Performance Deep Learning Library. arXiv:1912.01703 [cs.LG] https://arxiv.org/abs/1912.01703

[50] PyTorch Contributors. [n. d.]. PyTorch Sparse Documentation. https://docs.pytorch.org/docs/stable/sparse.html. Accessed: 2026-04-05.

[51] 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

[52] Semih Salihoğlu, Jeremy Chen, Yuqing Huang, Mushi Wang, and Ken Salem. 2026. Cardinality Estimation Graphs. *Commun. ACM* 69, 2 (Jan. 2026), 99–109. doi:10.1145/3780104

[53] 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

[54] Amir Shaikhha, Mathieu Huot, Jaclyn Smith, and Dan Olteanu. 2022. Functional collection programming with semi-ring dictionaries. *Proc. ACM Program. Lang.* 6, OOPSLA1, Article 89 (April 2022), 33 pages. doi:10.1145/3527333

[55] Marco Siracusa, Olivia Hsu, Victor Soria-Pardos, Joshua Randall, Arnaud Grasset, Eric Biscondi, Doug Joseph, Randy Allen, Fredrik Kjolstad, Miquel Moretó Planas, and Adrià Armejach. 2026. Ember: A Compiler for Embedding Operations on Decoupled Access-Execute Architectures. In *2026 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*. 150–163. doi:10.1109/CGO68049.2026.11395192

[56] Shaden Smith, Jee W. Choi, Jiajia Li, Richard Vuduc, Jongsoo Park, Xing Liu, and George Karypis. 2017. *FROSTT: The Formidable Repository of Open Sparse Tensors and Tools.* http://frostt.io/

[57] Nitish Srivastava, Hanchen Jin, Jie Liu, David Albonesi, and Zhiru Zhang. 2020. MatRaptor: A Sparse-Sparse Matrix Multiplication Accelerator Based on Row-Wise Product. In *2020 53rd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. 766–780. doi:10.1109/MICRO50266.2020.00068

[58] Nitish Srivastava, Hanchen Jin, Shaden Smith, Hongbo Rong, David Albonesi, and Zhiru Zhang. 2020. Tensaurus: A Versatile Accelerator for Mixed Sparse-Dense Tensor Computations. In *2020 IEEE International Symposium on High Performance Computer Architecture (HPCA)*. 689–702. doi:10.1109/HPCA47549.2020.00062

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

[60] 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

[61] W.F. Tinney and J.W. Walker. 1967. Direct solutions of sparse network equations by optimally ordered triangular factorization. *Proc. IEEE* 55, 11 (1967), 1801–1809. doi:10.1109/PROC.1967.6011

[62] David Vengerov, Andre Cavalheiro Menck, Mohamed Zait, and Sunil P. Chakkappen. 2015. Join Size Estimation Subject to Filter Conditions. *Proc. VLDB Endow.* 8, 12 (Aug. 2015), 1530–1541. doi:10.14778/2824032.2824051

[63] Lucas Wilkinson, Kazem Cheshmi, and Maryam Mehri Dehnavi. 2023. Register Tiling for Unstructured Sparsity in Neural Network Inference. *Proc. ACM Program. Lang.* 7, PLDI, Article 188 (June 2023), 26 pages. doi:10.1145/3591302

[64] Jaeyeon Won, Willow Ahrens, Saman Amarasinghe, and Joel S. Emer. 2026. Insum: Sparse GPU Kernels Simplified and Optimized with Indirect Einsums. In *Proceedings of the 31st ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2* (USA) *(ASPLOS '26)*. Association for Computing Machinery, New York, NY, USA, 993–1006. doi:10.1145/3779212.3790176

[65] Guoqing Xiao, Chuanghui Yin, Tao Zhou, Xueqi Li, Yuedan Chen, and Kenli Li. 2023. A Survey of Accelerating Parallel Sparse Linear Algebra. *ACM Comput. Surv.* 56, 1, Article 21 (Aug. 2023), 38 pages. doi:10.1145/3604606

[66] Rohan Yadav, Alex Aiken, and Fredrik Kjolstad. 2022. SpDISTAL: compiling distributed sparse tensor computations. In *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis* (Dallas, Texas) *(SC '22)*. IEEE Press, Article 59, 15 pages.

[67] Bobby Yan, Alexander J Root, Trevor Gale, David Broman, and Fredrik Kjolstad. 2026. Fast Autoscheduling for Sparse ML Frameworks. In *2026 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*. 28–43. doi:10.1109/CGO68049.2026.11394842

[68] Carl Yang, Aydın Buluç, and John D. Owens. 2022. GraphBLAST: A High-Performance Linear Algebra-based Graph Framework on the GPU. *ACM Trans. Math. Softw.* 48, 1, Article 1 (Feb. 2022), 51 pages. doi:10.1145/3466795

[69] Carl Yang, Yangzihao Wang, and John D. Owens. 2015. Fast Sparse Matrix and Sparse Vector Multiplication Algorithm on the GPU. In *2015 IEEE International Parallel and Distributed Processing Symposium Workshop*. 841–847. doi:10.1109/IPDPSW.2015.77

[70] Zihao Ye, Ruihang Lai, Junru Shao, Tianqi Chen, and Luis Ceze. 2023. SparseTIR: Composable Abstractions for Sparse Compilation in Deep Learning. In *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 3* (Vancouver, BC, Canada) *(ASPLOS 2023)*. Association for Computing Machinery, New York, NY, USA, 660–678. doi:10.1145/3582016.3582047

[71] Guowei Zhang, Nithya Attaluri, Joel S. Emer, and Daniel Sanchez. 2021. Gamma: leveraging Gustavson's algorithm to accelerate sparse matrix multiplication. In *Proceedings of the 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems* (Virtual, USA) *(ASPLOS '21)*. Association for Computing Machinery, New York, NY, USA, 687–701. doi:10.1145/3445814.3446702

[72] Genghan Zhang, Olivia Hsu, and Fredrik Kjolstad. 2024. Compilation of Modular and General Sparse Workspaces. *Proc. ACM Program. Lang.* 8, PLDI, Article 196 (June 2024), 26 pages. doi:10.1145/3656426

[73] Huihui Zhang, Anand Venkat, and Mary Hall. 2016. Compiler transformation to generate hybrid sparse computations. In *Proceedings of the Sixth Workshop on Irregular Applications: Architectures and Algorithms* (Salt Lake City, Utah) *(IA3̂ '16)*. IEEE Press, 34–41.

[74] Tuowen Zhao, Tobi Popoola, Mary Hall, Catherine Olschanowsky, and Michelle Strout. 2022. Polyhedral Specification and Code Generation of Sparse Tensor Contraction with Co-iteration. *ACM Trans. Archit. Code Optim.* 20, 1, Article 16 (Dec. 2022), 26 pages. doi:10.1145/3566054

A Artifact Appendix

A.1 Abstract

The Nacho compiler is provided as a C++ library, which generates CUDA/C++ code, as well as nanobind [33] bindings that enable calling the generated code via Python 3. Specifying a Nacho program is done via embedded C++, and then calling the compiler. We additionally provide the benchmarking scripts used to make the figures in Section 5. The artifact can be evaluated on any x86-64 machine (with TBB [30] support) and with an NVIDIA GPU (with CCCL [11] support). The benchmarks require at least 24 GB RAM for both the CPU and GPU, and at least 200 GB disk space. It is possible to restructure the evaluation to use less disk space (e.g., download a few SuiteSparse [18]/FROSTT [56] tensors, evaluate, delete them, evaluate the next batch), but our artifact is not set up to do this.

A.2 Artifact check-list (meta-information)

- **Compilation:** We use g++ 11.4.0 and CUDA 12.6, and compile with `-O3` to compile the generated code at C++20. g++ 11.4.0 is also used to compile the Nacho compiler. The generated CUDA is compiled for `sm_89`; override `CMAKE_CUDA_ARCHITECTURES` for a different GPU.
- **Data set:** We evaluate on synthetic data for Figures 8 and 9, with the synthetic data generators in benchmarks/powerlaw_heatmap_csr_add.py and benchmarks/dcsr_lb_comparison.py, respectively; SuiteSparse [18] for Figures 10, 11, 12, 13, 14, and 15b; and FROSTT [56] for Tables 1 and 2.

- **Run-time environment:** Ubuntu 22.04 with the NVIDIA 560.35 driver, Python 3.12.12, and PyTorch 2.8.0+cu128. The remaining Python dependencies are pinned in requirements.txt: NumPy 2.3.3, SciPy 1.17.0, pandas 2.3.2, Matplotlib 3.10.6, networkx 3.5, nanobind 2.12.0, CMake ≥ 3.30, Ninja, and scikit-build-core ≥ 0.10. oneTBB 2022 (`libtbb.so.12`) provides the CPU runtime. Generated figures are typeset with LaTeX, which needs texlive-latex-base, texlive-latex-extra, cm-super, dvipng, and ghostscript.
- **Hardware:** We evaluate on a 24-core Intel i9-14900K (8 performance cores with SMT, and 16 efficiency cores) with 188 GB of RAM, and a 24 GB NVIDIA RTX 4090. Benchmarks are pinned to the 16 hardware threads of the 8 performance cores using `numactl -physcpubind 0-15`.
- **Execution:** `bash run_all_benchmarks.sh` runs every experiment in sequence. Each script is also a standalone entry point, so individual figures can be reproduced on their own.
- **Metrics:** We evaluate end-to-end runtime of sparse tensor algebra kernels, where the arguments are already in GPU memory for GPU benchmarks. Each result was collected by performing a warm-up run followed by the mean of 14 runs, where the 2 fastest and slowest runs were excluded. Reported runtimes are end-to-end for all systems, including analysis phases (partitioning and precompute/symbolic phases), and output allocations, as well as the format conversions performed by PyTorch Sparse for some operations.
- **Output:** Running the artifact should reproduce Figures 8, 9, 10, 11, 12, 13, 14, and 15b, and Tables 1 and 2. Figures are written to `benchmark_results/` as EPS and PDF, alongside an `.npz` of the raw series for each; the tables are printed to standard output.
- **How much disk space required (approximately)?:** 200 GB
- **How much time is needed to prepare workflow (approximately)?:** About 20 minutes for the setup. Build should take less than 5 minutes. Dataset download for running the benchmarks depends on the network bandwidth. Can take 1-2 hrs for the downloading the complete 145 GB datasets.
- **How much time is needed to complete experiments (approximately)?:** About 3 hrs.
- **Publicly available?:** At https://github.com/rootjalex/nacho
- **Code licenses:** MIT License.
- **Archived:** https://doi.org/10.5281/zenodo.21926844

Methodology and Experimental Setup.

A.3 Description

A.3.1 How to access. The artifact is a single Git repository, available at https://github.com/rootjalex/nacho and archived at https://doi.org/10.5281/zenodo.21926844. It contains the Nacho compiler (`src/`, `include/`), the kernel declarations evaluated in the paper (`compiler.cpp`), the runtime (`runtime/`, `python/`), the baselines we compare against (`baselines/`), and the benchmark scripts (`benchmarks/`).

A.3.2 Hardware dependencies. An x86-64 machine and a CUDA-capable NVIDIA GPU. The x86-64 machine should have at least 16 physical cores. The GPU experiments need a device with at least 24 GB of memory.

A.3.3 Software dependencies. A CUDA toolkit with `nvcc` either on `PATH` or named by `CUDACXX`, oneTBB, a C++20 compiler, Python 3.12 or newer, and LaTeX for the figures. Every Python dependency except PyTorch is installed by `requirements.txt`; PyTorch is installed separately because its wheel is tied to a CUDA version, and LaTeX because it is not a Python package. We tested with the versions listed under *Run-time environment* above.

A.3.4 Data sets. Two datasets are required. Locations for both are set by environment variables:

```
export NACHO_SUITESPARSE_DIR=/path/to/suitesparse
export NACHO_FROSTT_DIR=/path/to/frostt
```

SuiteSparse [18] supplies the real matrices. benchmarks/datasets/download_suitesparse_reals.sh fetches the 1712 real-valued matrices we use.

FROSTT [56] supplies the third-order tensors for Tables 1 and 2: `nell-2`, `nell-1`, `1998DARPA`, and `fb-m`. download_frostt.py fetches them and sort_frostt.py deduplicates and lexicographically sorts their coordinates, which the generated kernels require. `fb-m`'s coordinates are additionally permuted to `kij` order so that its short mode can lead to more compression for CSF evaluation.

A.4 Installation

Clone the repository:

```
git clone https://github.com/rootjalex/nacho nacho
cd nacho
```

Install the dependencies that are not Python packages. Ensure CUDA toolkit is installed; LaTeX and `numactl` from the system package manager, and oneTBB. We use the following conda environment:

```
conda create -n nacho_env python=3.12
conda activate nacho_env
conda install conda-forge::tbb-devel conda-forge::cmake \
   nvidia::cuda-toolkit=12.6 texlive-core ghostscript
```

If you already have CUDA toolkit 12.6 and don't wish to reinstall via conda, `nvcc` has to be findable before anything is configured:

```
export PATH=/usr/local/cuda-12.6/bin:$PATH
```

Then install the Python dependencies.

```
pip install -r requirements.txt
pip install torch \
  --index-url https://download.pytorch.org/whl/cu126
```

Build the Nacho compiler, generate the kernels declared in `compiler.cpp`, and compile them into the Python extension:

```
cmake -S . -B build
cmake --build build -j16
./build/compiler
pip install --no-build-isolation -ve .
```

Confirm the installation is succesful with

Claim	Script (benchmarks/)	Output
Fig. 8	heatmap_csr_add.py	powerlaw_csr_add_cpu_10000x1000000.pdf powerlaw_csr_add_cuda_10000x1000000.pdf
Fig. 9	heatmap_dcsr_mul.py	dcsr_lb_comparison_cpu_heatmap.pdf dcsr_lb_comparison_gpu_heatmap.pdf
Fig. 10	csr_add.py	csr_add_cpu_0-1600.pdf csr_add_gpu_0-1600.pdf
Fig. 14	csr_mul.py	csr_mul_cpu_0-1600.pdf csr_mul_gpu_0-1600.pdf
Fig. 11	coo_add.py coo_mul.py	coo_add_gpu_0-1600.pdf coo_mul_gpu_0-1600.pdf
Fig. 13	coo_csr_add.py	coo_csr_add_cpu_0-1600.pdf coo_csr_add_gpu_0-1600.pdf
Fig. 12	spgemm.py	spgemm_gpu_0-1300.pdf
Fig. 15a	csr_add_3.py	csr_add3_gpu_0-1600
Fig. 15b	sssmm.py	sssmm_gpu_0-130.pdf0
Tbl. 1	frostt_tensors_add.py	standard output
Tbl. 2	inner_prod.py	standard output

Table 3. Each evaluation claim, the script that reproduces it, and the basename it writes into `benchmark_results/`.

```
python tests/smoke.py
```

A.5 Experiment workflow

Fetch the datasets once:

```
cd "$NACHO_SUITESPARSE_DIR"
bash <nacho>/benchmarks/datasets/\
     download_suitesparse_reals.sh
find . -name '*.tar.gz' -print0 \
  | xargs -0 -P 8 -n 1 tar -xzf
cd <nacho>
python benchmarks/datasets/download_frostt.py --all
python benchmarks/datasets/sort_frostt.py --all
```

Then run every experiment:

```
bash run_all_benchmarks.sh
```

Each line of that script is an independent entry point, so a single figure can be reproduced without running the rest. Settings are shared across scripts live in benchmarks/config.py which include dataset paths, iteration counts, launch geometry, and output directory — each can be overridden by an environment variable of the same name prefixed with `NACHO_`.

A.6 Evaluation and expected results

Every script prints a per-input line as it runs, then a summary with the geometric-mean speedup over each baseline and the number of inputs on which the generated kernel disagreed with its reference. Correctness is checked on every input, so a run that reports zero mismatches has also validated the kernels it timed. Figures are written to `benchmark_results/`. Table 3 maps each claim in Section 5 to the script that reproduces it and the file it writes.