A Probabilistic Perspective on Tiling Sparse Tensor Algebra

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

Zi Yu Xue
MIT
Cambridge, Massachusetts, USA
fzyxue@mit.edu

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

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

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

Sara Achour
Stanford University
Stanford, California, USA
sachour@stanford.edu

Mark Horowitz
Stanford University
Stanford, California, USA
horowitz@ee.stanford.edu

Abstract

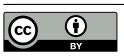
Sparse tensor algebra computations are often memory-bound due to irregular access patterns and low arithmetic intensity. We present D2T2 (Data-Driven Tensor Tiling), a framework that optimizes static coordinate-space tiling schemes to minimize memory traffic by identifying and leveraging relevant high-level statistics from input operands. For a given tensor algebra computation, D2T2 collects statistics from input tensors, builds a probability distribution-based model of the tensor computation, and uses it to predict traffic for various tiling configurations. It searches over tile shape and size configurations to minimize total traffic. We evaluate D2T2 against Tailors and DRT, two state of the art tiling schemes for sparse tensor algebra. We find that D2T2 achieves, on average, a 2.54× speedup over Tailors and a 1.13× lower memory bandwidth compared to DRT for sparse-sparse matrix multiplication (SpMSpM). We also achieve 1.22–48.94× lower bandwidth for SpMSpM and up to 34.31× lower bandwidth for tensor operations (TTM and MTTKRP) than conservative static tiling schemes. Unlike prior tiling techniques, D2T2 is deployable without specialized hardware support. On Opal, a 16nm sparse tensor algebra accelerator, D2T2 generated tiling configurations that achieve 1.23–3.34× speedups compared to their original hand-tuned configurations.

CCS Concepts

• **Computer systems organization → Architectures**; • **Software and its engineering → Software notations and tools**; • **Computing methodologies → Modeling and simulation**.

Keywords

Tensor Algebra, Sparse Computation, Hardware Acceleration, Tiling



MICRO '25, Seoul, Republic of Korea

ACM ISBN 979-8-4007-1573-0/25/10
https://doi.org/10.1145/3725843.3756095

ACM Reference Format:
Ritvik Sharma, Zi Yu Xue, Nathan Zhang, Rubens Lacouture, Fredrik Kjolstad, Sara Achour, and Mark Horowitz. 2025. A Probabilistic Perspective on Tiling Sparse Tensor Algebra. In *58th IEEE/ACM International Symposium on Microarchitecture (MICRO '25), October 18–22, 2025, Seoul, Republic of Korea.* ACM, New York, NY, USA, 14 pages. https://doi.org/10.1145/3725843.3756095

1 Introduction

Due to the importance of sparse tensor algebra [18, 19, 30, 32], a variety of acceleration techniques have been proposed including hardware accelerators [13, 27, 41], FPGA libraries [4], and two-in-four structured sparsity on GPUs [23]. These approaches tackle the utilization problem head-on, dedicating custom circuitry to handling data access — trading generality for improved performance for a select family of sparse tensor operations. Efficient sparse tensor algebra computation is fundamentally a *utilization* problem: the unpredictable and dynamic data access patterns makes keeping computational resources occupied a tall task.

Hardware accelerators use a memory hierarchy to improve performance deploying small and fast near-compute memory buffers [21, 28] to enable local data reuse. Typically, the size of the sparse tensors exceeds the capacity of such buffers, forcing the partitioning of data into smaller tiles. Each of these tiles needs to fit in the hardware accelerator buffers for correct execution. The memory movement is then explicitly handled by the software, which shuffles these tiles across the target memory many times to perform the final computation [7, 13, 16, 20, 25, 29, 33].

Most of these accelerators choose a target dataflow along with a static data partitioning that is uniform-sized in the coordinate-space (a tile of the logical tensor including both zeros and non-zeros) [13, 16, 20, 28]. This chooses a tile size that is guaranteed to fit in the on-chip buffers, assuming the worst-case occupancy (i.e. a dense tile). Hence, the scheme partitions tensors into tiles of identical logical size and shape based on the available buffer capacity without regard to the tensor sparsity. While this enables zero-overhead execution on each set of tiles, buffer utilization generally remains poor and produces excessive memory movement [13, 17, 22, 26, 40].

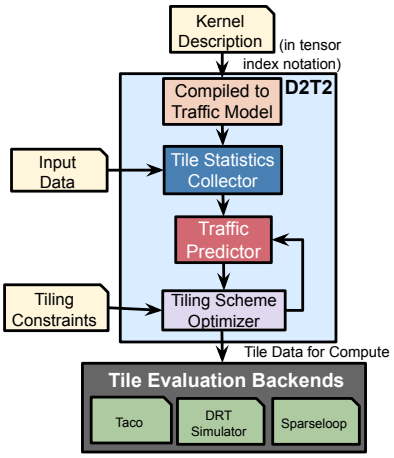


Figure 1: Tool Chain Flow for Data-Driven Tensor Tiling

While software-based optimization tools for auto-scheduling have been proposed [14, 38], they are mostly for CPUs, and sometimes restricted to sparse-dense computations [38]. Since CPUs operate with caches, these tools do not have to deal with the hard constraints of tiles fitting in an accelerator buffer.

For accelerators, recent work on optimizing tiling of sparse-sparse computations has addressed this issue using either hardware-supported dynamic tiling (DRT) [26] or aggressive tiling by supporting buffer overflows (Tailors [40]), achieving lower memory movement than conservative static methods. However, these approaches require specialized hardware and may not support arbitrary computation (dataflow) orders. Many sparse tensor accelerators [13, 16, 27, 41] use fixed dataflows with hardware-tuned optimizations, while reconfigurable designs [20, 24, 26] support multiple dataflows but lack optimizations for every case. This raises the question: can a static tiling scheme for a dataflow be made competitive to these recent methods without relying on specialized hardware?

In this work, we answer affirmatively. We present D2T2: Data-Driven Tensor Tiling, a tool (shown in Figure 1) that uses the sparse data distributions to find memory-movement-optimal non-uniform coordinate-space tiling schemes for the dataflow order choice made by the accelerator. D2T2 explores the search space of static tiling configurations using a distribution aware traffic modeling framework and builds static tiling schemes that are competitive with or outperform the prior efforts that require specialized hardware support. This is made possible by two key observations:

(1) Dense memory analysis can be extended to sparse tensor algebra by considering the *probability* and *expected size* of accesses.
(2) These probabilities and expectations can be estimated for a given computation using structural metadata contained within the compressed sparse fiber format.

Our contributions are:

(1) An algorithm that captures high-level statistics of a given tensor, that we find are convenient to calculate given a compressed data structure.
(2) A statistical model that estimates input/output traffic for sparse tensor programs, capturing data-dependent behavior.
(3) A tile scheme optimizer that minimizes total data traffic for a given tensor algebra kernel and buffer size. It first finds an optimal tile shape using the statistical model, then conservatively adjusts tile size while ensuring that the input tiles fit.

D2T2 supports various (possibly fused) tensor algebra kernels across dataflow orders, using real-world data from the Suitesparse and FROSTT datasets [8, 36] and achieves 1.22–48.94× lower bandwidth for SpMSpM and up to 27.34× for tensor operations like TTM and MTTKRP, compared to conservative sparse tiling. Against prior state-of-the-art techniques, Tailors and DRT, which use different dataflow orders for SpMSpM, we achieve a 2.54× improvement in runtime over Tailors and 1.13× improvement in memory bandwidth compared to DRT. Unlike prior work, it does not require specialized hardware and applies optimizations during initial tiling, adding only 9.3% and 7.9% overhead for Tile Statistics Collection and Tile Optimization over the initial tiling time on average.

2 Background

Sparse tensor algebra is a generalization of dense linear algebra to higher dimensions and sparse data structures. Tensor computations are often expressed in tensor index notation [31] (also referred to as Einsum notation). The Tensor index notation (TIN) was used by the TACO compiler [19] and included the underlying storage format and the computation iteration order. This order corresponds to the loop ordering of the generated loop nest of the computation and is referred to as the dataflow order. For any computation, various possible dataflow orders exist with different trade-offs [1].

For example, TIN for sparse matrix multiplication kernel using the standard inner-product dataflow order between compressed matrices A, B can be expressed as:

$$C[i,j] = \sum_k A[i,k] \times B[k,j] \qquad Order : i \rightarrow j \rightarrow k \tag{1}$$

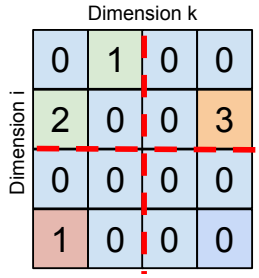
This dataflow corresponds to the pseudocode below, where *row_crds* are row coordinates and rows[i] returns column coordinates for row i (similarly, cols[i] for row coordinates).

```
for i in A.row_crds:
  for j in B.col_crds:
    for k in intersect(A.rows[i], B.cols[j]):
      C[i, j] += A[i][k] * B[j][k]
```

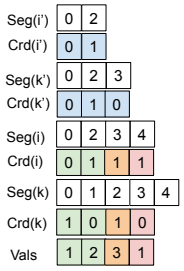
Where A and B are stored in i,k and j,k format so that they agree with the dataflow order. Alternative dataflow orders include Gustavson's algorithm and outer-product matrix multiplication expressed with $(i \rightarrow k \rightarrow j)$ and $(k \rightarrow i \rightarrow j)$ index variable dataflow orders. Not all dataflow orders are compatible with all tensor formats, the tensor storage format needs to match the dataflow order (as shown above). In this work, we assume the user has provided a valid dataflow order for the given computation and data formats.

2.1 Sparse Data Structures

Sparse tensors are stored using compressed representations (formats) that retain only the non-zero values in the tensor. Example formats include (coordinate format) COO [10] and (compressed sparse row) CSR [9]. The compressed sparse fiber (CSF) format is a trie-structured storage format used for arbitrary order sparse



(a) Example sparse matrix with tiling on red lines



(b) Tiled CSF with new indices

Figure 2: An example tiled tensor and its CSF storage format where segment arrays address into the coordinate arrays. i', k' correspond to `outer`, and i, k correspond to `inner` indexes.

tensors [5, 13, 17, 19, 26, 40]. In a CSF each level corresponds to an index variable present in the tensor. In this representation, a coordinate at one level is linked to its child coordinates at the next level. Only the children that have sub-trees containing non-zero values are stored. CSFs can be stored in memory using two arrays for each dimension (or index). The first is a segment (*Seg*) array that stores the start and stop positions of the coordinates of the same index, whereas the second is a coordinate (*Crd*) array with all the coordinates of each segment abutted together. Figure 2a shows a sparse matrix with the tiled fiber-tree implementation in Figure 2. With the CSF format, we can access an array of coordinate data and the pointers (segments) that access this data for any index "i" of the sparse tensor B with crd(B, i) and seg(B, i) respectively. We can also fetch the size of the coordinate array at any part of the CSF using the corresponding *seg* array.

2.2 Tiling Sparse Tensors

Tiling a tensor partitions the tensor dimensions and reorders them to generate a new tensor. A tiled sparse tensor will contain new additional index variables. All index variables of the input tensor correspond to an index variable in the dataflow order. Thus, reorganizing input data to generate new indices leads to new indices in the dataflow order as well. The new index variables are mapped to a higher level of the memory hierarchy. They allow iteration through tiles of the input data, as opposed to singular values. For example, for a two-level memory hierarchy, we can transform the inputs from matrices (addressed with 2 variables i, k) to tensors (addressed with 4 variables i', k', i, k) where i, k index variables are also transformed to span the partitioned iteration space inside a tile by $A[i,k] \rightarrow A[i',k',i,k]$ or $B[j,k] \rightarrow B[j',k',j,k]$. For the tiled inner-product computation, the dataflow order changes from $i \rightarrow j \rightarrow k$ to $i' \rightarrow j' \rightarrow k' \rightarrow i \rightarrow j \rightarrow k$. In this paper, we refer to variables that iterate across tiles (using a i', j' notation) and call them `outer` indices, and the variables that refer to iteration inside a tile (using a i, j notation) and call them `inner` indices.

2.3 Sparse Data Tiling Schemes

Tiling partitions input data such that each partition fits in the local memory of a hardware accelerator, allowing the accelerator to leverage its fast local memory for more efficient reuse. Tiling in dense tensor algebra guarantees full buffer utilization and consistent, predictable reuse for the input and output tensors, leading to a simple tiling strategy. Data gets tiled in the coordinate space, where each tile contains data within a fixed shape/size boundary. This enables scheduling without any overhead for orchestrating multiple operands as tile boundaries match. However, applying the same approach in tiling sparse tensor algebra suffers from low memory utilization alongside unpredictable and low data reuse. This inefficiency has motivated the exploration of different strategies for sparse tensor algebra, as shown in Table 1.

Static coordinate space tiling (Conservative) partitions data in the coordinate space where indexes shared among different tensors are tiled with the same tile dimension. Popularly, Conservative tiling is performed with a square tile shape [13, 17, 40, 41]. It generally uses a conservative tile size which ensures that each tile fits the target buffer in the worst case, i.e. a fully dense tile. For example, Extensor [13] uses a 128×128 size for its processing elements (PEs). We will refer to this scheme as Conservative.

Dynamic Reflexive Tiling (DRT [26]) builds tiles dynamically to maximize each tile's occupancy while shared tile dimension sizes are kept identical for all operands. Thus, DRT gets larger tile sizes that still fit in the input buffers. However, the tile shape and size built by DRT are dictated by the greedy dynamic-tile-aggregation algorithm implemented in the hardware. DRT operates on data that is tiled first by a static coordinate space scheme. DRT proposes specialized hardware that iterates on tiles on-chip and then aggregates multiple small tiles together into a larger tile before computing on the tile. Thus it effectively tiles data again in hardware.

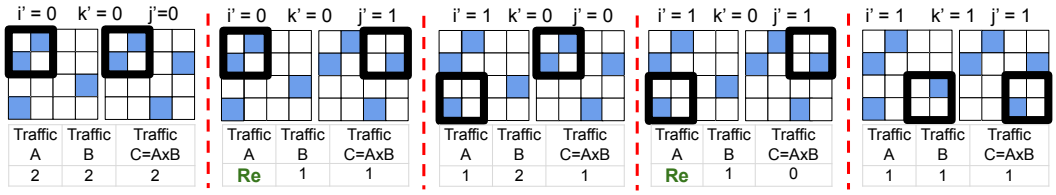
Tailors [40], on the other hand, builds large-size coordinate space tiles that are not guaranteed to fit in the accelerator buffer but does not specialize the tile shapes for the target expression and dataflow order. Tailors tiles data twice to build large-size coordinate space tiles. However, since the tiles generated from Tailors can overflow a buffer, specialized buffers are required which support streaming of the input data that overflows the target buffer.

3 Example: Gustavson's Matmul

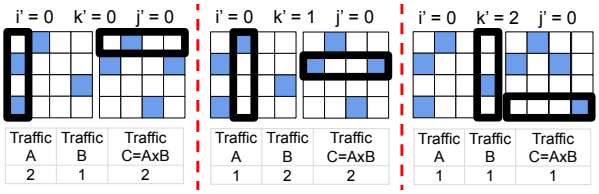
Consider a tiled Sparse Matrix-Sparse Matrix Multiply (SpMSpM) operation using Gustavson's algorithm [12] on input tensors stored

Table 1: Different tiling scheme features. To fit: tiles sized to fit buffer size, overflow: tiles may be larger than buffers, conserve: size that ensures even dense tiles fit. Tiling strategy includes if the tiling is static/dynamic (performed on hardware) and the number of times data needs to be tiled.

Tiling Scheme	Tile Size	Tile Shape	Tile Strategy	Hardware Overhead
Conservative	Conserve	Square	Static: tiled once	None
DRT	To fit	non-uniform rectangles	Dynamic: tiled twice	**Large**: specialized dynamic tiling hardware to generate tiles
Tailors	Overflow	Square	Static: tiled twice	**Low**: special buffers for overbooked input tiles
D2T2 (us)	To fit	non-uniform rectangles	Static: tiled twice	**Minimal**: ability to write out partial output results directly



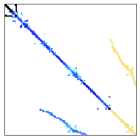
(a) Gustavson's multiplication ($C_{ij} = A_{ik} \times B_{kj}$) with Conservative Tiling. Matrices A, B are visualized with their constituent tiles being multiplied at any iterations highlighted with a black boundary. Tile traffic (in terms of number of non-zeros for simplicity) is shown for A, B tiles, and the output tile of matrix C. Matrix A is reused for multiple iterations (marked with Re).



(b) Multiplication with reshaped tiles with D2T2.

Config	Traffic A	Traffic B	Traffic C	Total Traffic	Total Tile Iterations
Conservative	4	7	5	16	5
D2T2 (Ours)	4	4	5	13	3

(c) Number of Elements accessed for different tiling schemes



(d) Matrix with output reuse: RMA10

Figure 3: Tile reshaping for reducing memory movement required to perform a computation

as a CSF format. Gustavson's algorithm (shown without tiling in Equation (1)) is an efficient SpMSpM computation order [17, 22, 41] that maintains stationary output rows, thereby balancing input and output reuse.

We tile this SpMSpM-ikj computation to allow computing on tiles that use the system memory hierarchy efficiently. Various valid tiling configurations exist, all of which have different tradeoffs. We represent a statically tiled version of the Gustavson SpMSpM using an annotated TIN program, which now includes the `outer` indexes and the corresponding tile dimension sizes for the scheme.

$$
\begin{aligned}
T1, T2 &= TileVar() \\
A &= Input[i' : N/T1, k' : M/T2, i : T1, k : T2] \\
B &= Input[k' : M/T2, j' : L/T1, k : T2, j : T1] \\
C &= Output[i' : N/T1, j' : L/T1, i : T1, j : T1] \\
C_{i',j',i,j} &= \sum_{k=1}^{T2} A(i', k', i, k) \times B(j', k', j, k) \\
order\ i' &\rightarrow k' \rightarrow j' \rightarrow i \rightarrow k \rightarrow j
\end{aligned}
$$

In this specification, the first line contains the tile variables, i.e., the tiling dimension $T1, T2$. The next three lines contain specifications that define the dimension size for the `outer` index variables for input matrices and the output matrix. Finally, we write the tensor kernel as a reduction over tensor values, accessed by the indices in the argument, while the traversal *order* specifies the order in which the indices of the computation are traversed.

Using Conservative scheme: In this scheme, $T1, T2$ are chosen to ensure all tiles fit in the input buffer under worst-case (fully dense) conditions, so $T1 \times T2 \leq$ Buffer Size. Square tiles are typically used since they would presumably balance input and output traffic [13, 17, 24]. Figure 3a shows an example computation with buffer size 4 and Conservative scheme with a 2×2 configuration.

Analyzing costs of a computation: For analyzing this example computation in Figure 3a, we approximate the traffic of a tensor as the number of nonzero elements fetched during the tiled execution. In this computation, the dataflow *order* determines how the computation is performed. Here, tensor A is indexed by variables i, k and is thus only read once; thus, as seen in Figure 3c, the traffic (number of nonzero elements transferred) for matrix A is equal to its number of nonzeros. Matrix B is indexed by k, j variables (and not i). Thus, during the computation, each tile of B will be re-fetched multiple times during different iterations of the `outer` index variable i'.

An alternative tiling configuration can reduce traffic: We can improve traffic by adjusting the tiling scheme (the tile dimensions $T1$ and $T2$) to better match the sparse data distribution in the input matrices. Figure 3b shows the matrices tiled with $T1 = 4, T2 = 1$, which is more amenable to the input matrix data distribution. Unlike before, where most tiles were accessed, the empty column at $k' = 2$ leads to skipping the corresponding tile iteration, reducing traffic.

This configuration also reduces total traffic by minimizing how often matrix B is read. A large tile dimension for the i index variable ($T1$) reduces the number of iterations of the corresponding `outer` index i', resulting in a traffic of 4 for both input operands. Thus, a non-uniform tiling scheme found using the data distribution can change the input/output traffic for a computation drastically.

Tradeoffs of changing tiling schemes: But, a single solution does not fit all cases. For example, using the outer-product-like tile shapes shown in Figure 3b does not satisfy any local output reuse. Thus, for matrices that have correlated data distributions that could result in a lot of output reuse, such schemes would hurt performance. For example, the matrix RMA10 shown in Figure 3d results in 87.49 MB traffic for square tiles, 72.04MB for tiles with an aspect ratio of 4 and 318.86MB for outer-product-tiles. Therefore, the data distribution of the input sparse tensors impacts how tiling can be optimized, and we collect statistics on input data to analytically model the relationship between the tile configuration and input/output traffic.

The next section describes our prediction model used to search for the optimal tiling schemes without executing the program.

4 Probabilistic Memory Modeling

At the heart of D2T2 is a predictive, probabilistic memory model that estimates input/output traffic from statistical properties of the input tensor data. D2T2 analyses the memory traffic of a single tensor algebra kernel computing a single result tensor. We model the traffic of a sparse tensor algebra computation by building upon ideas from dense tensor algebra memory movement modeling.

4.1 Dense Modeling

At a high level, memory analysis determines how many times an access is performed, and how large the access is. In dense analysis, we have static loop nests, where the number and size of accesses can be computed from the structure of the computation and data dimensions. Consider the case where we want to calculate the input traffic of an input tensor A embedded within multiple loop nests:

```
for i in range(I):
    for j in range(J):
        for k in range(K):
            access(A)
```

Where $range()$ denotes all the possible values the index variable takes. A traditional dense analysis would suggest that the total memory movement for A is:

$$|range(I)| \times |range(J)| \times |range(K)| \times size(access),$$

where $|range(Var)|$ refers to the number of elements in the variable Var i.e. it's dimension size. This traffic model for dense computations only depends on the number of iterations and access size.

4.2 Sparse Modeling

For sparse computations, compilers [2, 19, 34], accelerators [13, 22, 41], and abstractions [17, 24] often perform pruning optimizations. These filter out data from a tensor using intersections for multiplication operations to reduce the amount of unnecessary memory movement. Sparse tensor operations will only load nonzero tensor regions, and the amount of data loaded for a given computation is highly data-dependent. To generalize the dense model, we augment the static loop nest abstraction with the *probability* that an access occurs and the *expected* size of accesses.

Thus the memory traffic for a tensor A depends on the probability that a address in A (given by $addr$) is accessed ($\mathbb{P}(access(addr))$) and the expected number of non-zero (nnz) elements retrieved from A provided it is accessed ($\mathbb{E}(\text{nnz}(addr) \mid access(addr))$), both quantities that depend on the input dataset:

$$\sum_{addr} \mathbb{P}(access(addr))\, \mathbb{E}(\text{nnz}(addr) \mid access(addr)) \qquad (2)$$
$$addr \in dom = (range(I) \times range(J) \times range(K))$$

4.2.1 Modeling Sparse Loads. D2T2 targets arbitrary tensor algebra expressions (similar to previous tools that use TIN, like TACO [19]). It analyses the input computation to build a probabilistic model for computation traffic. We demonstrate the D2T2 modeling framework with the following example.

$$D[i,j] = (A[i] + B[i]) * C[i,j] \qquad (3)$$

Since any efficient sparse tensor algebra implementation will only load data $A[i]$ (which may be a value and its corresponding index information or even an entire tile containing multiple values and indexes) from tensor A if (1) the data is nonzero, and (2) the data is necessary for the computation [17, 19, 24]. $A[i]$ is necessary if it may be used to compute the output tensor i.e. used to generate the output directly ***or*** used for a necessary sub-computation.

We model the probability of load accesses for tensor A, $\mathbb{P}_A(load)$, by first deriving the constraint that must hold for an access to occur, and then estimating the probability that the constraint holds. Thus for Equation (3), the probability $A[i]$ is loaded is given by:

$$\underbrace{A[i] \neq 0 \wedge (A[i] + B[i]) \neq 0 \wedge \underbrace{((A[i] + B[i]) \times C[i,j] \neq 0)}_{(A[i]+B[i]) \text{ is used}}}_{A[i] \text{ is used in a necessary computation}} \qquad (4)$$

Here, the leftmost constraint ensures $A[i]$ is non-zero, satisfying condition (1). The middle and right constraints ensure $A[i]$ is used during the computation, satisfying condition (2). To see if $A[i]$ would be loaded, we can simply the above constraint to get:

$$\underbrace{A[i] \neq 0}_{\text{A is nonzero}} \wedge \underbrace{\exists j.C[i,j] \neq 0}_{\text{load request for A}} \qquad (5)$$

Here, the middle constraint is eliminated since $A[i] + B[i]$ will be computed whenever $A[i]$ is nonzero. However, the rightmost constraint is only satisfied if there exists some j where $C[i,j]$ is nonzero (given $A[i] + B[i] \neq 0$). In general, the probability of an access remains unchanged for an addition operation, but is decreased by a multiplication. This is because addition leads to a union between data, whereas multiplications lead to intersection-based filtering of data in sparse tensor algebra [15, 17, 19].

The filtering behavior of multiplication depends on the structure of the tensors involved. In Equation (5), the probability of access is:

$$\mathbb{P}_A(access(i,j,k)) = \mathbb{P}(A[i] \neq 0 \wedge \exists j.C[i,j] \neq 0) \qquad (6)$$

Note that calculating the above expression would require co-iterating over the input operands. We *estimate* this probability in terms of single tensor probabilities by assuming that the nonzero structures of A and C are uncorrelated. This assumption simplifies these probabilities significantly, reducing the previous calculation to $\mathbb{P}_A(access(i,j,k)) = \mathbb{P}(A[i] \neq 0)\, \mathbb{P}(\exists j.C[i,j] \neq 0)$.

Multiplying this probability of access of $A[i]$ with the expected size of $A[i]$ summed over the entire domain gives the estimated memory movement for A, giving the following equation.

$$Traffic_A = \sum_{dom} E[nnz|\vec{o}]\, \mathbb{P}_A(access(\vec{o})) \qquad (7)$$

Where the variable $\vec{o}$ gives a particular point in the `outer` index variables. If $A[i]$ is a scalar, the expected data size ($E[nnz|\vec{o}]$) is 1. But, D2T2 memory modeling directly deals with tiles. Thus, this expected size is the expected tile size of tensor A. This expected tile size is estimated directly from the input data and includes the values and metadata sizes to capture tiled traffic accurately.

4.2.2 Modeling Sparse Stores. The probability of a sparse store is the probability that the sparse result is nonzero. We recursively analyze the tensor sums and products in the computation to calculate the constraint that must hold for a nonzero output. If a tensor D is produced by summing tensors B_k (over index k), then $D[i,j]$ is nonzero if any of $B_k[i,j] \neq 0$, i.e. $D[i,j] \neq 0 \equiv \bigvee_k B_k[i,j] \neq 0$.

For an output tensor D calculated as sum of products $D[i,j] = \sum \prod B_i[\dots]$, then $D[i,j]$ would be nonzero if at least one of the summands $\prod B_i[\dots]$ is nonzero. In turn, this term will be nonzero only if all of the multiplied elements are also nonzero. Thus the

probability of $D[i, j]$ being nonzero is:

$$\mathbb{P}(D[i,j] \neq 0) \equiv \mathbb{P}(\bigvee \bigwedge (B_1[\cdots] \neq 0) \cdots (B_n[\cdots] \neq 0)) \quad (8)$$

Note that, if all B_k are assumed to have independent (any tensor B_k is independent from tensor $B_{k'}$), this probability can be calculated as a simple function of single-tensor statistics. We multiply this probability by the size of the output tile to estimate the expected number of nonzeros, and similarly, compute the expected metadata per index to estimate the total output tile size. Summing this over the entire domain yields the total expected output traffic.

Next in Sections 4.3, 4.4 we describe how we estimate these probabilities using statistics collected by D2T2 Tile Statistics Collector.

4.3 Extracting Single-Tensor Probabilities

Although estimating the probability of a tensor element at a specific position—especially when dependent on multiple indices—may appear costly, it can be efficiently calculated from the CSF format. The CSF is traversed from `outer` to `inner` index variables, thus the data at any index variable in the CSF is automatically conditioned on prior index values allowing us to readily estimate marginal and conditional probabilities at any index with a CSF traversal.

For example, the number of nonzeros in a fiber (sub-)tree can be computed as the difference in positions between its left-most and right-most leaves. This allows us to estimate the probability that a tensor element is nonzero, conditioned on `outer` indices and marginalized over the indices contained within the (sub-)tree. In essence, thanks to the compressed structure of the CSF format, a simple traversal is sufficient to extract the single-tensor probabilities we require for traffic modeling. While the entire CSF provides detailed conditional probabilities at any point, we only extract four key statistics: the expected and maximum size of a subtile, and two averages over the exact conditional probabilities.

To collect these statistics, we first tile the tensor data using the Conservative scheme. While creating tiles, we capture the maximum (**`MaxTile`**) and the average (**`SizeTile`**) tile size of each input tensor. These tile sizes are the sum of the number of nonzeros and the size of all the segment and coordinate arrays of the tile stored in the CSF format. This ensures that we estimate the traffic arising from both the non-zero values and the metadata of these tiles.

Next, Tile Statistics Collector collects statistics across different indexes in the CSF data structure. At any level of the CSF, we calculate the conditional probability of a valid element (i.e. one with a nonzero subtree) as the size of the compressed fiber for a given index variable divided by the size of that dimension. We treat the indexes that operate on different levels of a memory hierarchy differently. For `outer` indexes (that address tiles) we collect the statistic **`PrTileIdx`**, and for the `inner` indexes (that address values) we collect the statistic **`ProbIndex`**.

`PrTileIdx` is the average probability estimate for each `outer` index variable calculated using the average number of valid elements in the `outer` index divided by dimension size. This average estimate predicts traffic as well as using the entire distribution for `outer` index variables. We estimate the probability of a non-zero tile of a tensor A that contains `outer` iteration variables $[o_0, o_1, o_2, \dots]$ as:

$$P_{tile}(A[\vec{o}]) = \prod_{o_i} PrTileIdx(A, o_i) \quad (9)$$

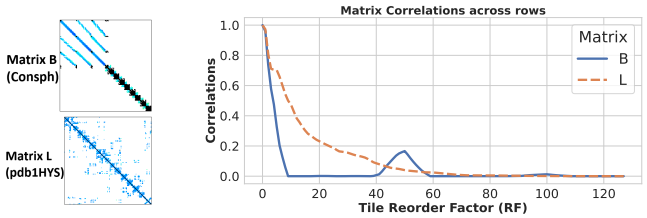


Figure 4: Two matrices and corresponding Correlations

`ProbIndex` collects the probability distribution for any `inner` index variable as a conditional probability conditioned over previous `inner` index variables (index variables that lie above the chosen `inner` index in the fibertree), while still being averaged over all `outer` index variables (thus averaged over all tiles). For a tensor A, we estimate the probability of a nonzero element at position $(i0, i1, i2...)$ given the tile exists as:

$$Pr(A[i_0, i_1, ...]) = \prod_j \text{ProbIndex}(i_j \mid i_0, i_1, ...i_{j-1}) \quad (10)$$

4.4 Estimating Correlated Probability Proxies

The above single tensor probabilities are estimated as density estimates at all index levels. However, this can fail to capture the structured data distribution of sparse tensors. For example, diagonally dominant and banded tensors have correlated data, where, assuming independent probabilities across different index variables of the same tensor, leads to inaccurate performance and traffic modeling. Thus, assuming independent probabilities can overestimate the amount of output traffic for correlated matrices, because it assumes very few of the output partial products will be reduced with each other. In turn, for structured matrices, these reductions are common. To capture the effect of correlated data, we calculate a correlation statistic inside a tile (**`Corrs`**) and the correlation between tile positions (**`TileCorrs`**).

`Corrs` calculates the amount of overlapping data (i.e. correlation) that is non-zero (and thus exists in the next fiber of the sub-trees) for two given positions of the index variable of the tensor that is contracted upon, as a function of the shift between them to capture output reductions. We show two example suitesparse matrices, and their **`Corrs`** plot in Figure 4.

Consider SpMSpM-ikj where the index k is contracted upon. In this computation, when we reduce the partial products over the k index, we add together rows of the $B[k, j]$ matrix for different k values. When two rows of $B[k, j]$, say $B[k = k_i, j]$ and $B[k = k_i+s, j]$, are summed, the overlapping coordinates inside these rows are added, resulting in output reuse. Thus, the estimated output reuse between two such rows can be estimated with the size of their intersection, i.e. $|B[j|k = k_i] \cap B[j|k = k_i + s]|$.

The total reductions between all such $k = k_i$ rows with $k = k_i + s$ rows, is $\sum_{k=k_i} |B[j|k = k_i] \cap B[j|k = k_i + s]|$. For $s = 0$, this function is the number of nonzeros in the tensor; therefore, we appropriately normalize this function such that it is 1 at $s = 0$. For SpMSpM-ikj, the estimated degree of output reduction between rows $k = k_i, k = k_i + s$ is:

$$Corrs(B, s)_k = \frac{\sum_{k_i \in T} |B[j|k = k_i] \cap B[j|k = k_i + s]|}{\sum_{k_i \in T} |B[j|k = k_i]|} \quad (11)$$

We calculate this statistic as an average across tiles for all values of s, to allow for estimating reductions with different tile choices.

The total reductions for the chosen tile size of k (T_k) is the sum of **Corrs** over T_k ($\sum_{s=0}^{T_k} Corr(B,s)_k$). We also calculate the one-dimensional correlation over k for $A[i,k]$, but find that it does not result in much deviation from the estimate from $B[k,j]$ alone.

Later in Section 6.7, we show that **Corrs** informs tiling choice fairly well. If the sum of **Corrs** over $[0,T]$ is low, corresponding to low output reuse, then outer product-like tiles (with a small tile dimension T_k) perform better. Whereas, if the total sum is high, then a symmetric tile choice (with a larger T_k) performs better.

We generalize the entire approach for calculating **Corrs** for higher-order tensors by calculating the estimated overlap for all underlying coordinates given a point in the contracted index ($crds|k = k_i$) and the coordinates given a shifted point ($crds|k = k_i + s$).

Correlations between `outer` indexes (that address a tile) are captured in **TileCorrs**. Since sparse tensors often have varying characteristics such as shifted data patterns, bands, etc., we calculate the correlation of whether different values of the `outer` indices exist in a tensor. For any outer index iteration variable "i'" we calculate the following average 1D correlation function:

$$TileCorrs(B,s)_{i'} = \sum_{i \in D_i/T} [B[i'=i] \neq 0 \cap B[i'=i+s] \neq 0] \tag{12}$$

Where the dimension size of the index variable i in the original matrix was D_i, with the initial tiling dimension T. Thus, D_i/T is the entire domain of the i' `outer` index variable.

5 Probabilistic Tiling

In this section, we describe how D2T2 uses the general traffic modeling scheme presented in Section 4 to predict traffic using data statistics. Section 5.1 demonstrates the traffic modeling process for SpMSpM-ikj (Gustavson's algorithm) example. We also discuss in Section 5.2 how D2T2 uses this model to search over different tiling configurations. Finally, we validate the model in Section 5.3.

5.1 Traffic Predictor

Input traffic in D2T2 is calculated as the sum of the traffic of individual tensors. For each input, the corresponding traffic is modeled as the sum over the entire iteration domain, the product of the expected number of nonzero elements in a tile and the probability a tile is accessed, as shown in Equation (7). It is then simplified by approximating the position-dependent $\mathbb{E}[nnz|\vec{o}]$ term as an averaged value given by **SizeTile**. This results in the expression

$$\text{Input Traffic} = SizeTile \sum_{dom} \mathbb{P}(access(\vec{o})) \tag{13}$$

While D2T2 builds the traffic formulas for different sparse tensor algebra expressions, we take a SpMSpM-ikj ($A[i,k] \times B[k,j]$) where i,j,k indexes are tiled to demonstrate how D2T2 estimates traffic. In this case, the entire domain *dom* of the tiling iteration space is given by the `outer` variables $\{i',k',j'\}$. Whereas, similar to the derivation in Equation (5), the probability of an access for matrices A and B is given by:

$$\mathbb{P}_A(access) = \mathbb{P}(A[i',k'] \neq 0 \wedge \exists j'.B[k',j'] \neq 0) \tag{14}$$

$$\mathbb{P}_B(access) = \mathbb{P}(B[k',j'] \neq 0 \wedge \exists i'.A[i',k'] \neq 0) \tag{15}$$

The first matrix tile $A[i',k']$ is only loaded on each iteration of the $\{i',k'\}$ variables. Thus the sum over j' index does not affect the $P_{tile}(A[i',k'])$ term, but it sums over $\mathbb{P}(\exists k'.B[k',j'] \neq 0)$ to result in $PrTileIdx(B,k')$. This results in the traffic for matrix A as:

$$\begin{aligned} Traffic_A &= \text{SizeTile}_A \sum_{i',k',j'} P_{tile}(A[i',k']) \times \mathbb{P}(\exists j'.B[k',j'] \neq 0) \\ &= \text{SizeTile}_A \frac{D_i}{T_i} \frac{D_k}{T_k} P_{tile}(A[i',k']) \times PrTileIdx(B,k') \end{aligned} \tag{16}$$

On the other hand, the other matrix $B[k,j]$ is loaded on each iteration of all $\{i',k',j'\}$ variables. Thus, the probability of a load is $\mathbb{P}(access(B[\vec{o}])) = P_{tile}(B) \times \mathbb{P}(\exists i'.A[i',k'] \neq 0)$. When this is summed over i' we get The $\sum_{i'} \mathbb{P}(\exists i'.A[i',k'] \neq 0) = E(|i'|) \times \mathbb{P}(A[i',k'] \neq 0)$. The latter term is $P_{tile}(A[i',k'])$. Thus, the overall traffic becomes:

$$\begin{aligned} Traffic_B &= SizeTile_B \sum_{k',j'} P_{tile}(B[k',j'])E[|i'|]P_{tile}(A[i',k']) \\ &= SizeTile_B \frac{D_k}{T_k} \frac{D_j}{T_j} P_{tile}(B[k',j'])E[|i'|]P_{tile}(A[i',k']) \end{aligned} \tag{17}$$

Where $E(|i'|)$ is the size of the `outer` index variable i'. If tile dimension T_i is smaller than the original tile size($T_i < T$), we model the `outer` index to change accordingly i.e. $E(|i'|) = D_i/T_i$ where D_i was the original dimension.

If, instead, the new tile size T_i is larger than T ($T < T_i$), it corresponds to incorporating multiple (T_j/T) tiles together. How this affects $E(|i'|)$ depends on whether adjacent tiles are correlated. If they are not correlated,i.e. they are empty, making the block bigger doesn't change the number of reloads. So we estimate this changing iteration space using **TileCorrs** as

$$E(|i'|) = \frac{D_i}{\sum_{i'=0}^{\frac{T_i}{T}} TileCorrs[i']} \tag{18}$$

To estimate input traffic at different tile shapes, we modulate the tile sizes (T_i, T_k, T_j) in the above formula. We assume that the tile probabilities $P_{tile}(...)$ do not change drastically with tile shape.

Output traffic is estimated as the sum over the entire domain of the probability of a store and the expected output tile size, resulting in the following equation:

$$Traffic = \prod_{dom} \mathbb{P}(store)E[size(tile)] \tag{19}$$

We calculate the probability of a store ($\mathbb{P}(store)$) by computing the output traffic expression in Equation (8). We replace the individual probabilities for each tensor B_i with the probability of its tile $P_{tile}(B_i[..])$, and for each multiplication term, we multiply tile probabilities, and for each addition term, we add probabilities.

For SpMSpM-ikj, the $\mathbb{P}(store)$ is $P_{tile}(A[i',k'])P_{tile}(B[k',j'])$ summed over the entire domain $\frac{D_i D_k D_j}{T_i T_k T_j}$.

We calculate the expected output tile ($E(size(tile))$) in two steps. We first estimate the size of the output tile without any correlations. This is the product of the size of the output tile with the probability of a nonzero element in the output tile. Similar to prior cases, to calculate the probability of a nonzero, for additions we add probabilities (i.e. for $A[i,k]+B[i,k]$ compute $min(1, Pr(A[i,k])+P(B[i,k]))$), for multiplications, we can multiply probabilities (i.e. for $A[i,k] \times B[k,j]$ compute $Pr(A[i,k]) \times Pr(B[k,j])$). So for $A[i,k] \times B[k,j]$ the probability of generating an output is $Pr(A[i,k]) \times Pr(B[k,j])$ when reduced over T_k. Finally, we estimate the size of the nonzero tile as the probability of a non-zero element multiplied by the dense size of the tile. For SpMSpM-ikj, the output tile ($size(tile)$) becomes

$T_i \times T_j \times T_k \times Pr(A) \times Pr(B)$. To get $E[size(tile)]$, we average this estimate over multiple samples.

We next consider the effect of correlated data, as we described in Section 4.4, and it is estimated with **Corrs**. Although each output might be improbable, if **Corrs** is large, any non-zero term is likely to have a number of partial products. We account for this by dividing each multiplication term by the sum of **Corrs** statistic from 0 to the target tile size T_k for each contracted index variable to estimate the output traffic. For SpMSpM-ikj, the expected output tile size is:

$$\mathbb{E}(size(tile)) = \frac{T_i T_j \times T_k \times Pr(A[i,k]) \times Pr(B[k,j])}{\sum_{s=0}^{T_k} Corrs(B,s)_k}$$

Given the above equation for the expected tile size, the total expected output traffic for SpMSpM-ikj, is given by:

$$\frac{D_i D_k D_j}{T_i T_k T_j} \times P_{tile}(A[i',k'])P_{tile}(B[k',j']) \times E(size(tile)) \quad (20)$$

5.2 Tiling Scheme Optimizer

We utilize the modeling scheme to perform tiling configuration optimizations. We optimize tiling configurations in two stages — first optimizing shape, and then optimizing size.

D2T2's shape optimization explores candidate tiling schemes of equal sizes but different tile shapes. It takes as input the tile configuration constraints that describe the target buffer size and the relationships between different tile dimensions. These dimensions are described as a function of the initial tile configuration and new variables that D2T2 can search over and uses the traffic prediction model to estimate total traffic. For example, for a matrix, a tile dimension relationship that maintains matrix size while modifying shape is the *aspect ratio* of the tile dimensions. For matrix $A[i,k]$, the relationship between i,k would be:

$$\text{Variables in } (A[i,k]) = \{i : T1, k : T2\} = \{i : T \times RF, k : \frac{T}{RF}\} \quad (21)$$

Here, T denotes the original Conservative tile size, and RF is a tunable variable controlling the tile's aspect ratio (RF^2). These constraints ensure that D2T2 searches only over different tile shapes, and the tiles still fit within the buffer. In practice, we evaluate a small set of geometrically sizes values for the tunable variables and use the traffic model to identify the tile shape with the optimum traffic. Geometric search is effective because traffic remains stable for small changes in the tile and only changes significantly when the tile captures new parts of the structured input data.

5.2.1 Tile Size Optimizations. Finally, D2T2 also allows incorporating tile size optimizations. We implement a simple tile resizing step where we conservatively calculate the total number of tiles that can fit in the input buffer based on the size of the tile with maximum occupancy (**MaxTiles**) and find the number of reshaped tiles that fit in the buffer, called *TileFactor*:

$$TileFactor = \frac{BufferSize}{MaxTiles} \quad (22)$$

D2T2 uses this estimate to conservatively increase the tile dimensions for the `inner` indices of all tensors. This effectively creates a larger tile that incorporates multiple smaller tiles inside.

5.3 Traffic Model Validation

We validate our predictive model's traffic estimates against actual traffic for SpMSpM-ikj dataflow described in Table 3 on the matrices described in Table 2. For each matrix, we capture **Corrs** from randomly sampling 1% of the tiles and all other statistics in Sections 4.3 and 4.4 from the full set of tiled data. Real traffic is measured by performing tiled matrix multiplication and recording input/output traffic. The output traffic is calculated by computing the tiled outputs with the TACO compiler [19]. Both predicted and measured traffic include the full CSF memory footprint, corresponding to the size of the values and metadata (coordinate and segment) arrays.

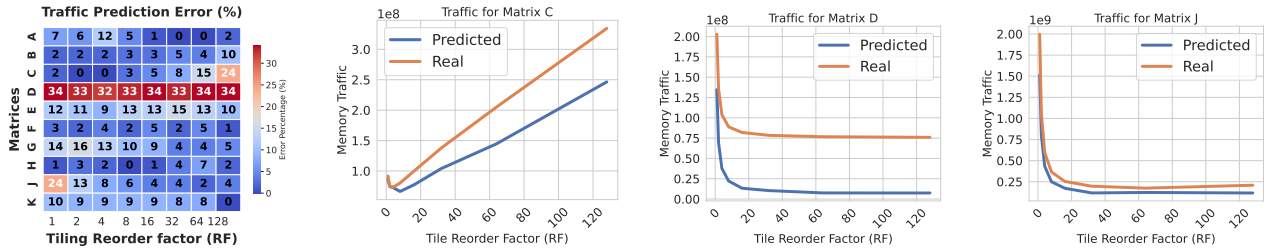
Analysis. Figure 5a shows the error between predicted and real traffic for $A \times A^T$ for different tile reorder factors (RF). Each reorder factor value maps to a different tile shape. We find that the D2T2 model matches actual traffic within 15% for most cases, with a few outliers showing higher error. Figures 5b-5d present the predicted and real memory traffic for these outliers.

For all three cases, D2T2 underestimates memory traffic, but captures the relationship between memory traffic and the reorder factor. Therefore, even with high absolute error, we can reliably use predicted memory traffic to choose between two tile shapes (i.e., RFs), or perform other kinds of relative comparisons. Since D2T2 uses relative comparisons, it can find performant tile configurations even with this form of absolute model error.

We believe the observed prediction errors arise from the output traffic modeling, which assumes independent input tensors. In the $A \times A^T$ kernel, the tiles may be correlated, leading to underestimation of valid intersections (i.e. filtering) in the computation. However, since the intersection rate is not tied to tile configuration, the model still accurately captures how traffic changes across tiling schemes. We also evaluate our model for uncorrelated ($A \times R$, with R random) and partially correlated ($A \times A'^T$, where A' is a shifted version of A) cases. For such cases, the predicted traffic closely matches actual traffic, with average errors of 2.9–9.7% and

Table 2: Tensors used in our evaluation from the Suiteparse matrices (SS), FROSTT tensors (FF), Facebook activities graph (FB). We modify FROSTT higher-dimensional tensors to 3-tensors by dropping one or more dimensions (marked FF*)

Label	**Tensor**	Dimension	Nonzeros	Dataset
A	mc2depi	525,825×525,825	2,100,225	SS
B	consph	83,334×83,334	6,010,480	SS
C	rma10	46,835×46,835	2, 329, 092	SS
D	sx-mathoverflow	24,818×24,818	239,978	SS
E	scircuit	170,998×170,998	958,936	SS
F	mac-econ-5000	206,500×206,500	1,273,389	SS
G	shipsec1	140,874×140,874	3,568,176	SS
H	pwtk	217,918×217,918	11,524,432	SS
I	soc-sign-epinions	131,828×131,828	841,372	SS
J	cop20k_A	121,192×121,192	2,624,331	SS
K	geom	7,343×7,343	23,796	SS
L	pdb1HYS	36,417×36,417	4,344,765	SS
M	cant	62,451×62,451	4,007,383	SS
N	bcsstk17	10,974×10,974	428,650	SS
O	email-EuAll	265,214×265,214	420,045	SS
P	amazon0302	262,111×262,111	1,234,877	SS
Q	p2p-Gnutella1	62,586×62,586	147,892	SS
R	soc-Epinions1	75,888×75,888	508,837	SS
S	sx-askubuntu	159,316×159,316	596,933	SS
T	Chicago-crime3	$6,187 \times 78 \times 33$	2,597,198	FF*
U	Uber3	183×1, 140×1,717	1,117,629	FF*
T	Facebook	$1,504 \times 42,390 \times 39,986$	737,934	FB
W	Nips3	$2,483 \times 2,863 \times 14,307$	3,101,609	FF*



(a) Real vs Predicted traffic (b) rma10 (C) (c) sx-Mathoverflow (D) (d) cop20k_A (J)

Figure 5: Model Validation with relative errors and traffic plots for $A \times B$ computation for different RF values. Each RF corresponds to the following tile configurations: matrix A $\{i : 128 \times \mathbf{RF}, k : \frac{128}{\mathbf{RF}}\}$ and matrix B $\{k : \frac{128}{\mathbf{RF}}, j : 128 \times \mathbf{RF}\}$.

Table 3: Benchmark kernels. SS refers to Suitesparse Matrix [8], SS^T refers to its transpose, FF refers to Frostt+Facebook Tensors [36, 37], FLAT means the tensor has one index dropped, RAND means a random tensor with 0.1% nonzeros.

Benchmark	TIN Expression	Dataflow	Datasets Used	Description
SpMSpM-ijk	$C_{ij} = A_{ik} \times B_{jk}$	i, j, k	**A**:SS, **B**:SS^T	Suitesparse Matrix times its transpose
SpMSpM-ikj	$C_{ij} = A_{ik} \times B_{jk}$	i, k, j	**A**:SS, **B**:SS^T	Suitesparse Matrix times its transpose
TTM	$X_{ijk} = C_{ijl} \times B_{kl}$	i, j, l, k	**A**:FF, **B**:RAND	Frostt/Facebook times the random Matrix
MTTKRP-3	$D_{ij} = A_{ikl} \times B_{jk} \times C_{jl}$	i, k, l, j	**A**:FF, **B**:RAND, **C**:RAND	Frostt/Facebook Tensor times two random matrices

worst-case errors below 18%. Unlike the fully correlated case, we do not observe any major outliers or systemic errors.

6 Evaluation

Benchmarks. We evaluate D2T2 for real-world datasets including matrices from the Suitesparse [8] and tensors from the Frostt [36] and Facebook tensor [37] datasets, described in Table 2. We use the labels presented in the table to refer to the matrices. These matrices have been previously used to evaluate prior tiling work [26, 40].

Tiling Scheme Optimizers. We compare D2T2-generated tiling schemes (**D2T2**) against the DRT and Tailors tiling scheme optimizers, two state-of-the art optimizers for sparse matrices:

- **Tailors** [40], a statistical tiling scheme that automatically optimizes tile sizes, but does not guarantee these tiles fit in the input buffer. Tailors was previously evaluated with the Sparseloop [39] execution backend [40] on the SpMSpM-ijk kernel with a target 10% rate of overbooking (%age tiles that overflow input buffers).
- **DRT** [26], a dynamic tiling scheme that produces non-square tiles at runtime by assembling micro-tiles in hardware. DRT is evaluated with a DRT simulation backend. DRT was previously evaluated on the SpMSpM-ikj kernel.
- **D2T2**, our non-square tiling scheme that optimizes tiles using a statistical model. D2T2 produces tiles that are guaranteed to fit in the input buffer, but may not fit in the output buffer. D2T2 supports the Sparseloop and DRT execution backends, and has a TACO backend which supports a wider range of kernels. TACO is used to measure input/output traffic as described in Section 5.3.

Square Tiling Schemes. In this evaluation, we use two square tiling schemes. **Conservative** selects square tiles that fit in the buffer in the worst case (assuming dense tiles) without considering data sparsity. It was used as a baseline in DRT paper. **Prescient** dynamically finds the largest square tile size that fits in the buffer with a binary search, therefore leveraging data sparsity. It was used as a baseline in Tailors paper.

Experimental Setup. We evaluate DRT and Tailors tiling scheme with the execution backend and computational kernel from the original publication. We find their optimized tiling configurations and report runtime or memory traffic improvements for each execution. When possible, we evaluate D2T2 on the same backend and kernel; any exceptions are described directly before each analysis.

In both Tailors and DRT, their original execution backends were parametrized to model the Extensor architecture, optimizing tiling between two memory levels. We similarly model the Extensor architecture and use the architectural parameters reported in the original architecture [13]. We target Extensor's PE buffer that supports a 128×128 size dense tiles, resulting in a target buffer of size $128KB$ with Extensor's PE memory bandwidth.

The accelerator is modeled as a push-memory where tiles are pushed to compute if both have nonzero elements and can not be skipped by tile filtering, and assume accumulations across tiles are performed at the main memory level. This optimization exists in the original accelerator design, and is used in the other tiling works[13, 17, 24, 26, 40]. We also include two non-standard modifications to the accelerator model. First, we support input buffer overflows as described in [40], to support tiles generated by Tailors. Second, we add support for handling output buffer overflows, so that over-sized D2T2 output tiles can be processed. Specifically, whenever a partial result overflows, it is streamed out, freeing up the output buffer. D2T2 does not require any specialized input buffer implementation that refetches that is required in Tailors [28].

Metrics. Given a target (*Trg*) tiling scheme and a reference (*Ref*) tiling scheme, we report the relative speedup ($Time_{Ref}/Time_{Trg}$), where $Time_{Ref}, Time_{Trg}$ are the execution times. When the execution times cannot be calculated, we report the relative traffic improvement $(Inp_{Ref} + Out_{Ref})/(Inp_{Targ} + Out_{Trg})$, where Inp_{Ref}, Inp_{Trg} are input traffic in megabytes, and Out_{Ref}, Out_{Targ}

are output traffic in megabytes. The DRT and Sparseloop backends both produce input/output traffic and execution time measurements, while the TACO backend produces only input/output traffic measurements. The speedup metric is favored and the traffic improvement metric is only used in comparisons involving TACO.

To demonstrate that the traffic improvement metric is a reasonable proxy measurement for speedup, we compute both metrics for all benchmark matrices for the SpMSpM-ijk kernel with Sparseloop – the results are shown in Figure 6a. The relationship between speedup and traffic improvement is linear. This suggests that sparse tensor algebra computation is memory-bound and thus, memory traffic improvements directly translate to runtime speedups.

6.1 Comparison against Tailors

Experimental Setup. We evaluate D2T2 against Tailors [40] on SpMSpM-ijk of $A \times A^T$ with the Sparseloop execution backend, and report the speedup relative to the Prescient square tiles. Both executions use the same architectural parameters and optimizations.
Analysis. Figure 6b presents the speedup results of D2T2 compared to Tailors. D2T2 tiling schemes achieve an average speedup of 2.54× over Tailors and 4.85× over Prescient tiling. In particular, D2T2 outperforms Tailors for matrices with very structured diagonal data which is heavily centered on the matrix diagonal (like matrices A (mc2depi), B (consph), C (rma10)), etc. In these cases, Tailors has trouble finding a tile shape where only 10% of tiles overflow, as all tiles are quite similar, increasing traffic due to refetching of overflowing input tiles. For matrices with irregular, drastically varying sparse data distributions, like L (*pdb1HYS*), R (*sx-askubuntu*), D2T2 performs similarly or slightly worse than Tailors.

6.2 Comparison against DRT

Experimental Setup. We next evaluate D2T2 against the DRT-generated and **`Conservative`** square tiles for SpMSpM-ikj computing $A \times A^T$. We report traffic improvement relative to **`Prescient`** square tile baseline. As outlined in the experimental setup, traffic improvement is a reasonable proxy for runtime. The DRT execution backend is employed for evaluating the DRT-generated tiles, while the TACO backend is used for D2T2-generated and the Prescient and Conservative square tiles. The TACO backend is chosen for D2T2 because the DRT simulator was unable to map D2T2-generated configurations to micro-tiles for most matrices, resulting in execution failures. For the DRT-generated tiling schemes, the DRT simulator could only run 14 out of 19 matrices when given a time-out period of 72 hours per computation. We report the results for the 14 matrices that were successfully run with DRT.

To ensure the traffic input/output measurements produced by the DRT backends and TACO backends are comparable, we evaluated input/output traffic measurements using both backends for all matrices with the Conservative tiling scheme and find that the average difference between the two simulators to be <5%.
Analysis. Figure 6c reports the memory traffic improvements of D2T2 against DRT. For the 14 matrices run by the DRT simulator, D2T2 achieved an average of 1.45× improvement over Prescient while DRT only achieved a 1.29× improvement. Thus, D2T2 provides a 1.13× reduction in memory traffic over DRT. D2T2 performs similar to DRT for uniform and structured matrices (A (*mc2depi*), B (*rma10*)) and outperforms DRT for matrices with variable but still structured data distributions (E (*scircuit*), J (*cop20k_A*)) We believe D2T2 performs well because DRT's tile aggregation hardware only has a local view of the matrix data, whereas D2T2 has a statistical view of the entire matrix, allowing it to outperform for more structured matrices. We also achieve on average 4.17× lower traffic than Conservative and 1.83× lower traffic than Prescient tiling.

Table 4: Memory traffic improvement for other kernels. For both the TTM and MTTKRP-3, we generate random matrices with 1% sparsity for all tensors except one (the tensor W, was too large to compute these kernels with 1% sparse matrices; therefore, for this tensor, we choose 0.1% sparse matrices).

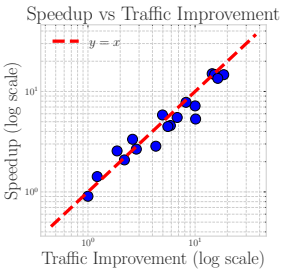
Label	Tensor	TTM	MTTKRP-3
T	Chicago-crime3	1.43×	1.72×
U	Uber3	1.22×	1.05×
V	Facebook	24.34×	15.49×
W	Nips3	12.68×	34.31×

6.3 Higher-order Tensor Computations

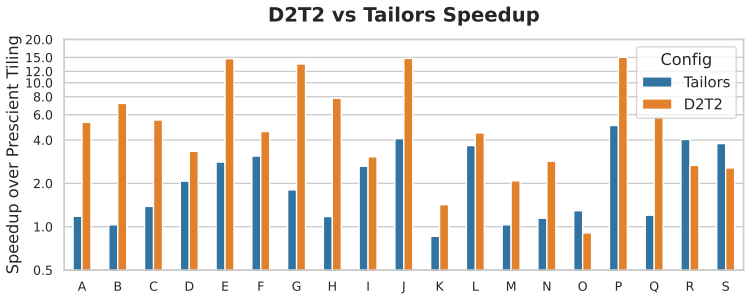
Experimental Setup: We also evaluate D2T2 for tiling **`TTM`** and **`MTTKRP-3`** (see Table 3). These kernels are not supported by the DRT/Tailor tile optimizers and the DRT/Sparseloop backends. Therefore, we simulate the D2T2 generated tiling configurations with our TACO-based traffic simulation to measure the traffic improvement achieved by D2T2 relative to a Conservative square tile baseline. Table 4 presents the relative traffic improvement results. Given a tensor with dimensions given by $T1 \times T2 \times T3$, for TTM, we use a random matrix with dimensions $T3 \times \max(T1, T2)$ and for MTTRP-3 we use other matrices of dimensions $T3 \times T1$ and dimensions $T1 \times T2$ and $T1 \times T3$. We generate these random sparse matrices with 1% sparsity for all tensors except W (*nips*). For this tensor, we instead use random matrices with 0.1% sparsity. This is because Tensor W is quite large, and thus computing **`TTM`** and **`MTTKRP-3`** for this tensor with 1% sparse other matrices becomes prohibitively slow to simulate.
Analysis: We find that for *TTM* and *MTTKRP*, we get average memory traffic improvement of 4.09× and 5.56× respectively. We find the largest imporvements for tensors T (*Facebook*) and W (*Nips3*) which have large dimensions with a lot of traffic from the other matrix operands as well. We find the smallest improvement for the tensor *Uber3*. This tensor has dimensions $183 \times 1140 \times 1717$. Given our dataflow choices for the TTM, MTTKRP-3, D2T2 generally increases the tile size corresponding to the first dimension (of size 183). This tile dimension is quite small, allowing very little opportunity for reshaping, this in turn, leads to only a small performance improvement.

6.4 Applying D2T2 on Opal Sparse Tensor Algebra Accelerator

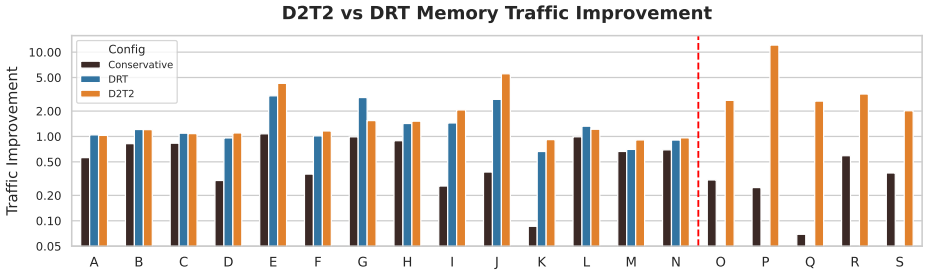
Since, unlike previous sparse tensor algebra tiling tools, D2T2 does not require major hardware changes, it can be readily deployed to sparse tensor algebra accelerators. We demonstrate this by using D2T2 to optimize tiling for a 16nm taped-out coarse-grained reconfigurable array (CGRA) for full sparse ML applications, Opal [3]. Opal has a memory hierarchy consisting of a 1.75 MB global



(a) Speedup vs memory traffic improvement for SpMSpM-ijk.



(b) D2T2 and Tailors Speedup over Prescient (Mean Improvements: D2T2 4.85×, Tailors 1.90×).



(c) DRT, D2T2 and Conservative, normalized to Prescient tiling. DRT simulator failed to simulate matrices after the red line. Mean improvements over Prescient and Conservative tiling by D2T2: (1.83×, 4.17×) and by DRT: (1.29×, 2.59×) respectively

Figure 6: D2T2 against DRT and Tailors normalized to Prescient Tiling

Table 5: Performance improvement with using D2T2 generated tiling schemes for SpMSpM-ikj on 8 Suitesparse Matrices on Opal [3] compared to the Prescient Tiling scheme.

Matrix	Dimension	Nonzeros	Speedup
bcsstm26	1,922 × 1,922	1,922	1.56×
bwm2000	2,000 × 2,000	7,996	3.08×
G33	2,000 × 2,000	8,000	1.70×
N_biocarta	1,922 × 1,996	4,335	3.34×
progas	1,650 × 1,900	8,897	2.51×
qiulp	1,192 × 1,900	4,492	1.45×
tols2000	2,000 × 2,000	5,184	1.60×
west2021	2,021 × 2,021	7,310	1.34×

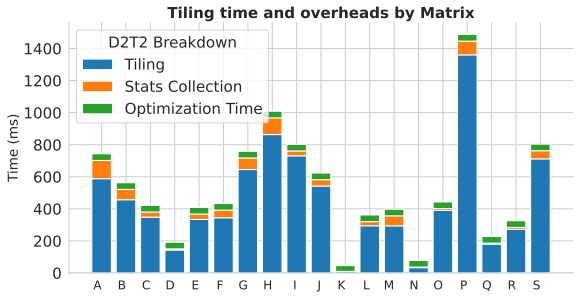


Figure 7: D2T2's overheads on the initial tiling for SpMSpM (Average is: Statistics Collection 9.3%, Optimization 7.9%).

buffer and a number of memory tiles, each of size 2 KB. Opal's 2 kB memory buffers can support a conservative tile size of 32 × 32 for matrices. This limit arises because, during execution, Opal stores index metadata and values separately, distributing them across the memory tiles for each sparse operand.

Table 5 reports the SuiteSparse matrices used in Opal's evaluation [3], along with the speedups achieved by D2T2-generated tiling configurations over Prescient tiling for SpMSpM-ikj. The Prescient baseline was the previously optimal tiling scheme used for Opal's evaluation [3]. We observe that D2T2 yields speedups ranging from 1.33× to 3.34×, with a geometric mean improvement of around 2×.

6.5 Overhead Analysis

D2T2 enables efficient tiling scheme selection for accelerators without requiring specialized hardware support. D2T2 optimizations are performed during tiling on the CPU, and consists of two steps: statistics collection and Tile scheme Optimization. Figure 7 shows the runtime overhead of these steps, compared to the time taken to tile a sparse CSR matrix into CSF format using the Eigen library [11] in C++ across all matrices. The data statistics collection step, implemented in C++, adds an average overhead of 9.3% relative to the initial tiling. In contrast, the tiling scheme optimization — currently implemented in Python — adds a near constant overhead, amounting to approximately 7.9% of the average time required to tile two matrices for the SpMSpM computation. Importantly, tiling is only a small fraction of the total runtime for the tensor computation.

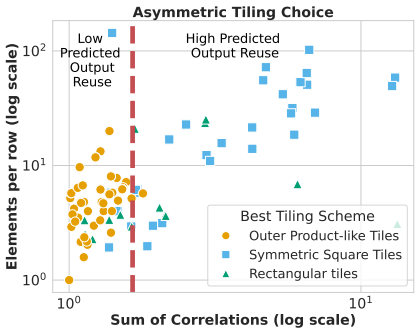


Figure 8: Desired Tile Shapes as a function of Sum of Correlations. Left region matrices have low predicted output reuse and right region matrices have high predicted output reuse

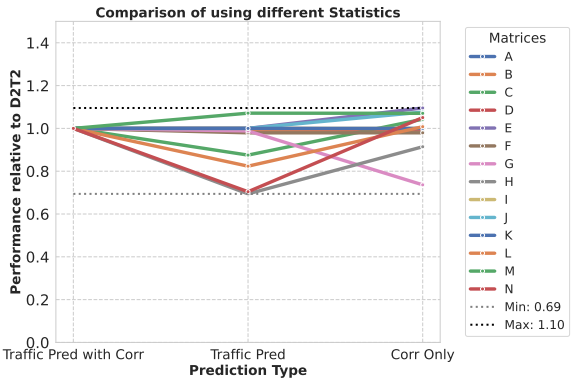


Figure 9: Memory traffic improvement with different prediction models relative to Traffic Prediction with Correlations.

6.6 Optimality Analysis of D2T2 Configurations

We next study the optimality of D2T2-generated configurations to better understand how much room for performance improvement remains. We compare D2T2 to an *exhaustive-search* static tiling scheme that selects low-traffic shapes from D2T2 and resizes them presciently using binary search. We execute all *exhaustive-search* candidate configurations using the **TACO** backend and choose the one with the best total traffic.

D2T2 has two sources of inefficiencies in finding tiling configurations. First, it can mispredict the optimal tile shape, and second, it performs a relatively conservative tile size optimization (compared to presciently resizing tiles). After these two steps, however, the primary difference between D2T2 and *exhaustive-search* schemes was in tile size, whereas the tile shape difference is not large. This is because various matrices prefer outer-product-like tiles (tall-skinny times short-fat). When such tiles are resized, their larger tile dimension can get clipped when the matrix dimension (D) is smaller than the new tile dimension ($TF * RF$). In fact, we find that D2T2 configurations have 52% (23%-100%) of the buffer utilization of the configurations found by the *exhaustive-search* scheme, showing tile size to be the larger concern.

However, even though our buffer utilization is lower, when we compare the memory movement between these schemes, we find that D2T2 gets 92.4% (83-100%) of the memory traffic improvement reported by the *exhaustive-search* scheme. In this dataflow, traffic for one of the input matrices that remains stationary (A) and the output matrix (X) remains similar in both configurations. Only the traffic for matrix B, which is repeatedly refetched, is reduced slightly in the *exhaustive-search* scheme, leading to only small improvements. Thus, the buffer utilization improvements with the *exhaustive-search* tiling scheme do not necessarily lead to large memory traffic improvements.

6.7 Ablation Studies of D2T2 Design Decisions

To assess the impact of different statistics, we modify D2T2 tiling by selectively using subsets of Tile Statistics Collector outputs. The resulting optimizations, ordered by decreasing complexity, are:

- Traffic Prediction with Correlations: We estimate output traffic via the proxy statistic **`Corrs`** to capture reductions in the computation. This scheme is used by D2T2.
- Traffic Prediction w/o Correlations: Same as above, but without using **`Corrs`**, which assumes minimal reductions.
- Using Correlations only: Tile shapes are chosen using only **`Corrs`**. Figure 8 shows the desired tile shapes against the sum of the **`Corrs`** array. We note that matrices with sum < 1.6 favor outer-product tiling, while others prefer square tiles for output reuse.

We use the selected statistics guide tile shape estimation for all these schemes, then apply tile size optimization. Figure 9 shows the relative performance of these schemes compared to the full traffic prediction with correlations. In the best case, simpler schemes reduce memory traffic by up to 10% compared to D2T2, while in the worst case, they perform at only 69% of D2T2's efficiency.

Finally, we evaluate the *performance benefits without retiling*. Similar to prior work in Tailors [40], D2T2 retiles data to generate tiles in the final optimized configurations. We examine the performance benefits achieved by D2T2 if we only tile data once. To accomplish this, we first collect statistics on the tiled input data for a tiling scheme and then find the optimal configurations. We then force each dimension of the generated configurations to be a multiple of the original tile size, normalizing it to the original tile dimension. Now we pack these original tiles into sets of tiles that together would have the tiling configuration recommended by D2T2 post normalization. These *packed tiles* are each indexed through a sparse data structure. Finally, we perform the target computation on the packed tiles' CSFs, transforming the original tensor algebra computation to a new tensor algebra expression with new indexes that access these small tiles which have been packed together.

We find that using such a scheme for a 128×128 title size does not cause any increase in traffic for 13/19 matrices. However, it can cause up to 5× higher traffic than D2T2 for the rest 6/19 more graph-like matrices. Overall, using this *packed tiles* scheme leads to a 31% drop in memory traffic improvement. This scheme still retains a 3.4× advantage over Prescient tiling. We can further mitigate these performance drops by using smaller initial tile sizes. Using 32×32 tiles leads to only an 11% drop in advantage over D2T2, resulting in

4.36× improvement over Prescient. Thus, the cost of dropping the second tiling step is often not large, and is sometimes beneficial.

7 Related Work

Some software-only scheduling optimisation tools have been previously proposed for sparse computations, like WACO [38] or BACO [14]. ML techniques like WACO often target sparse-dense computation, while D2T2 targets sparse-sparse computations. They also target CPUs which contain caches. Thus, their generated schedules do not need to lead to tiling schemes that are guaranteed to fit inside the cache. Whereas D2T2 is meant for dataflow accelerators, where tiles need to fit in available buffers.

Autotuning frameworks like BACO [14] are designed for use cases where the optimized computation will be performed multiple times, amortizing the high cost of autotuning. In comparison, D2T2 is a lightweight framework that can be used for individual computations while adding a very small overhead on the tiling step. The statistics-based insights provided from a probabilitic modeling approach used by D2T2 might prove useful in pruning large parts of the search spaces of such frameworks.

Prior work in sparse-sparse tiling optimizations for accelerators either utilize dynamic tiling on runtime [26] or require dynamic handling of tiled data [40]. While, DRT uses dynamic tiling to optimize sparse data, D2T2 only generates static tile configurations, and thus can be used for an accelerator by replacing the data preprocessing step from a single pass of Conservative tiling to tiling with D2T2 (which applies: Conservative tiling, input statistic collection, finding optimized tiling scheme and retiling).

Like D2T2, Tailors [40] also adopts a statistical view of tensor data. It utilizes the probability distribution of data occupancy across tiles to find tile size configurations that fit in input buffers for a target percentage of cases but, also allows for tiles to overflow the input buffer. To support these tiles, it proposes specialized buffers [28] that support refetching of input tile data when needed. In contrast, D2T2 allows us to perform computation while ensuring input data does not overflow. Thus it does not require such specialized buffers. While Tailors aggressively optimizes tile sizes, it does not present a way to explore the tile shapes systematically. In contrast, D2T2's modeling approach directly targets shape optimizations.

Some accelerators like Gamma [41], Spada [22] also have special hardware for fetching tensor data efficiently. In contrast, D2T2 optimizes traffic without detailed hardware information and optimizes tiling schemes while requiring little to no hardware overhead.

Prior work in [6] also optimizes sparse matrix multiplication order by predicting the size of sparse outputs. However, it is not general enough to target compiler optimizations like tiling. Furthermore, it only works for matrices .

8 Conclusion

The memory traffic of sparse tensor computations can be accurately estimated using statistics readily available from the compressed-sparse fiber (CSF) formats. These statistics can be efficiently computed during the initial data tiling, allowing specialization of tiling schemes to the sparse data distributions. Using these insights, we introduce D2T2, a tiling scheme optimizer which produces conservative yet performant tile configurations for real-world sparse tensor data. D2T2 tiling optimizations help build tiling heurestics to balance the tradeoff between input refetches and output reuse for sparse computations. We find that in general, optimal tiling typically resembles an outer-product patterns (tall-skinny times short-fat tiles) that maintains the thickness of tall tiles, i.e. the dimension that is reduced over, to be large enough such that it preserves most of the output reuse in the computation.

D2T2 optimized tiling schemes are more efficient than prior methods and translate to a 1.22–48.94× improvement for matrix multiplication, 1.22–24.34× improvement for TTM and 1.05–34.31× improvement in MTTKRP. D2T2 also beats prior state-of-the-art tiling schemes, achieving on average a 2.55× runtime speedup over Tailors and 1.13× memory traffic improvement over DRT without requiring any specialized hardware. D2T2 only adds a low overhead in the initial data tiling to perform statistics collection and tiling optimization, and does not require additional hardware for sparse accelerators, making these techniques easy to deploy in other sparse tensor algebra systems.

Data Availability Statement

The code/artifact for this work is made available on Zenodo [35].

Acknowledgments

We would like to thank Sai Gautham Ravipati for their help in running experiments on Opal. We would also like to thank Toluwanimi O. Odemuyiwa for help in understanding and running experiments with their work. Ritvik Sharma was supported by the Stanford Graduate Fellowship. Furthermore, this work was also partially supported by the Stanford Portal Center and the System X Alliance. Any opinions, findings, conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of the aforementioned funding agencies.

References

[1] Willow Ahrens, 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. https://doi.org/10.1145/3519939.3523442

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

[3] Po-Han Chen, Bo Wun Cheng, Michael Oduoza, Zhouhua Xie, Kalhan Koul, Sai Gautham Ravipati, Yuchen Mei, Rupert Lu, Alex Carsello, Mark Horowitz, and Priyanka Raina. 2025. Opal: A 16nm Coarse-Grained Reconfigurable Array for Full Sparse ML Applications. In *2025 IEEE Custom Integrated Circuits Conference (CICC)*. 1–3. https://doi.org/10.1109/CICC63670.2025.10983811

[4] Yuze Chi, Licheng Guo, and Jason Cong. 2022. Accelerating SSSP for Power-Law Graphs. In *Proceedings of the 2022 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays* (Virtual Event, USA) *(FPGA '22)*. Association for Computing Machinery, New York, NY, USA, 190–200. https://doi.org/10.1145/3490422.3502358

[5] 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. https://doi.org/10.1145/3276493

[6] Edith Cohen. 1998. Structure Prediction and Computation of Sparse Matrix Products. *Journal of Combinatorial Optimization* 2, 3 (Dec. 1998), 307–332. https://doi.org/10.1023/A:1009716300509

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

New York, NY, USA, 924–939. https://doi.org/10.1145/3352460.3358276

[8] 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. https://doi.org/10.1145/2049662.2049663

[9] SciPy Developers. [n. d.]. Compressed Sparse Row (CSR) Matrix. SciPy documentation. https://docs.scipy.org/doc/scipy/reference/generated/scipy.sparse.csr_matrix.html Accessed: September 25, 2025.

[10] SciPy Developers. 2024. *scipy.sparse.coo_matrix*. https://docs.scipy.org/doc/scipy/reference/generated/scipy.sparse.coo_matrix.html Accessed: 2024-11-20.

[11] Gaël Guennebaud, Benoît Jacob, et al. 2010. Eigen v3. http://eigen.tuxfamily.org.

[12] Fred G. Gustavson. 1978. Two Fast Algorithms for Sparse Matrices: Multiplication and Permuted Transposition. *ACM Trans. Math. Softw.* 4, 3 (sep 1978), 250–269. https://doi.org/10.1145/355791.355796

[13] 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. https://doi.org/10.1145/3352460.3358275

[14] Erik Orm Hellsten, Artur Souza, Johannes Lenfers, Rubens Lacouture, Olivia Hsu, Adel Ejjeh, Fredrik Kjolstad, Michel Steuwer, Kunle Olukotun, and Luigi Nardi. 2024. BaCO: A Fast and Portable Bayesian Compiler Optimization Framework. In *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 4* (Vancouver, BC, Canada) *(ASPLOS '23)*. Association for Computing Machinery, New York, NY, USA, 19–42. https://doi.org/10.1145/3623278.3624770

[15] 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. https://doi.org/10.1145/3485505

[16] Yu hsin Chen, Tien-Ju Yang, Joel S. Emer, and Vivienne Sze. 2018. Eyeriss v2: A Flexible Accelerator for Emerging Deep Neural Networks on Mobile Devices. *IEEE Journal on Emerging and Selected Topics in Circuits and Systems* 9 (2018), 292–308. https://api.semanticscholar.org/CorpusID:131771552

[17] 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. https://doi.org/10.1145/3582016.3582051

[18] Aamer Jaleel, Kevin B. Theobald, Simon C. Steely, and Joel Emer. 2010. High performance cache replacement using re-reference interval prediction (RRIP). *SIGARCH Comput. Archit. News* 38, 3 (jun 2010), 60–71. https://doi.org/10.1145/1816038.1815971

[19] 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. https://doi.org/10.1145/3133901

[20] Kalhan Koul, Maxwell Strange, Jackson Melchert, Alex Carsello, Yuchen Mei, Olivia Hsu, Taeyoung Kong, Po-Han Chen, Jake Ke, Keyi Zhang, Qiaoyi Liu, Gedeon Nyengele, Akhilesh Balasingam, Jayashree Adivarahan, Ritvik Sharma, Zhouhua Xie, Christopher Torng, Joel Emer, Fredrik Kjolstad, Mark Horowitz, and Priyanka Raina. 2024. Onyx: A Programmable Accelerator for Sparse Tensor Algebra. *to appear in IEEE Hot Chips Symposium (Hot Chips)* (August 2024).

[21] Eunji Lee, Hyokyung Bahn, and Sam H. Noh. 2014. A Unified Buffer Cache Architecture that Subsumes Journaling Functionality via Nonvolatile Memory. *ACM Trans. Storage* 10, 1, Article 1 (jan 2014), 17 pages. https://doi.org/10.1145/2560010

[22] Zhiyao Li, Jiaxiang Li, Taijie Chen, Dimin Niu, Hongzhong Zheng, Yuan Xie, and Mingyu Gao. 2023. Spada: Accelerating Sparse Matrix Multiplication with Adaptive Dataflow. In *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2* (Vancouver, BC, Canada) *(ASPLOS 2023)*. Association for Computing Machinery, New York, NY, USA, 747–761. https://doi.org/10.1145/3575693.3575706

[23] Asit K. Mishra, Jorge Albericio Latorre, Jeff Pool, Darko Stosic, Dusan Stosic, Ganesh Venkatesh, Chong Yu, and Paulius Micikevicius. 2021. Accelerating Sparse Deep Neural Networks. *ArXiv* abs/2104.08378 (2021). https://api.semanticscholar.org/CorpusID:233296249

[24] Nandeeka Nayak, Toluwanimi O. Odemuyiwa, Shubham Ugare, Christopher Fletcher, Michael Pellauer, and Joel Emer. 2023. TeAAL: A Declarative Framework for Modeling Sparse Tensor Accelerators. In *Proceedings of the 56th Annual IEEE/ACM International Symposium on Microarchitecture* (Toronto, ON, Canada) *(MICRO '23)*. Association for Computing Machinery, New York, NY, USA, 1255–1270. https://doi.org/Teeal

[25] Tony Nowatzki, Vinay Gangadhar, Newsha Ardalani, and Karthikeyan Sankaralingam. 2017. Stream-Dataflow Acceleration. In *Proceedings of the 44th Annual International Symposium on Computer Architecture* (Toronto, ON, Canada) *(ISCA '17)*. Association for Computing Machinery, New York, NY, USA, 416–429. https://doi.org/10.1145/3079856.3080255

[26] 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. https://doi.org/10.1145/3582016.3582064

[27] 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. https://doi.org/10.1109/HPCA.2018.00067

[28] Michael Pellauer, Yakun Sophia Shao, Jason Clemons, Neal Crago, Kartik Hegde, Rangharajan Venkatesan, Stephen W. Keckler, Christopher W. Fletcher, and Joel Emer. 2019. Buffets: An Efficient and Composable Storage Idiom for Explicit Decoupled Data Orchestration. In *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems* (Providence, RI, USA) *(ASPLOS '19)*. Association for Computing Machinery, New York, NY, USA, 137–151. https://doi.org/10.1145/3297858.3304025

[29] Raghu Prabhakar, Yaqi Zhang, David Koeplinger, Matt Feldman, Tian Zhao, Stefan Hadjis, Ardavan Pedram, Christos Kozyrakis, and Kunle Olukotun. 2017. Plasticine: A Reconfigurable Architecture For Parallel Paterns. In *Proceedings of the 44th Annual International Symposium on Computer Architecture* (Toronto, ON, Canada) *(ISCA '17)*. Association for Computing Machinery, New York, NY, USA, 389–402. https://doi.org/10.1145/3079856.3080256

[30] Chiara Ravazzi, Roberto Tempo, and Fabrizio Dabbene. 2018. Learning Influence Structure in Sparse Social Networks. *IEEE Transactions on Control of Network Systems* 5, 4 (2018), 1976–1986. https://doi.org/10.1109/TCNS.2017.2781367

[31] MMG Ricci and Tullio Levi-Civita. 1900. Méthodes de calcul différentiel absolu et leurs applications. *Math. Ann.* 54, 1 (1900), 125–201.

[32] Nathalie Henry Riche, Jean-Daniel Fekete, and Michael J. McGuffin. 2007. NodeTrix: a Hybrid Visualization of Social Networks. *IEEE Transactions on Visualization and Computer Graphics* 13 (2007). https://api.semanticscholar.org/CorpusID:8451881

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

[34] 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. https://doi.org/10.1145/3428226

[35] Ritvik Sharma. 2025. D2T2 Compiler. https://doi.org/10.5281/zenodo.17067445

[36] 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/

[37] Bimal Viswanath, Alan Mislove, Meeyoung Cha, and Krishna P. Gummadi. 2009. On the evolution of user interaction in Facebook. In *Proceedings of the 2nd ACM Workshop on Online Social Networks* (Barcelona, Spain) *(WOSN '09)*. Association for Computing Machinery, New York, NY, USA, 37–42. https://doi.org/10.1145/1592665.1592675

[38] Jaeyeon Won, Charith Mendis, Joel S. Emer, and Saman Amarasinghe. 2023. WACO: Learning Workload-Aware Co-optimization of the Format and Schedule of a Sparse Tensor Program. In *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2* (Vancouver, BC, Canada) *(ASPLOS 2023)*. Association for Computing Machinery, New York, NY, USA, 920–934. https://doi.org/10.1145/3575693.3575742

[39] Yannan N. Wu, Po-An Tsai, Angshuman Parashar, Vivienne Sze, and Joel S. Emer. 2022. Sparseloop: An Analytical Approach To Sparse Tensor Accelerator Modeling . In *ACM/IEEE International Symposium on Microarchitecture (MICRO)*.

[40] Zi Yu Xue, Yannan Nellie Wu, Joel S. Emer, and Vivienne Sze. 2023. Tailors: Accelerating Sparse Tensor Algebra by Overbooking Buffer Capacity. In *Proceedings of the 56th Annual IEEE/ACM International Symposium on Microarchitecture* (Toronto, ON, Canada) *(MICRO '23)*. Association for Computing Machinery, New York, NY, USA, 1347–1363. https://doi.org/10.1145/3613424.3623793

[41] 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. https://doi.org/10.1145/3445814.3446702