### Title

Accelerating Sparse Data Orchestration via Dynamic Reflexive Tiling




### ISBN

9781450399180



### Authors

Odemuyiwa, Toluwanimi O

Asghari-Moghaddam, Hadi

Pellauer, Michael

et al.



### Publication Date

2023-03-25

### DOI

10.1145/3582016.3582064

# Accelerating Sparse Data Orchestration via Dynamic Reflexive Tiling

Toluwanimi O. Odemuyiwa[1], Hadi Asghari-Moghaddam[2], Michael Pellauer[3], Kartik Hegde[2], Po-An Tsai[3], Neal Crago[3], Aamer Jaleel[3], John D. Owens[1], Edgar Solomonik[2], Joel S. Emer[3,4], Christopher W. Fletcher[2]

University of California, Davis[1]; University of Illinois at Urbana-Champaign[2]; NVIDIA[3]; MIT[4]

## ABSTRACT

Tensor algebra involving multiple sparse operands is severely memory bound, making it a challenging target for acceleration. Furthermore, irregular sparsity complicates traditional techniques—such as tiling—for ameliorating memory bottlenecks. Prior sparse tiling schemes are sparsity unaware: they carve tensors into uniform coordinate-space shapes, which leads to low-occupancy tiles and thus lower exploitable reuse. To address these challenges, this paper proposes *dynamic reflexive tiling* (DRT), a novel tiling method that improves data reuse over prior art for sparse tensor kernels, unlocking significant performance improvement opportunities. DRT's key idea is dynamic sparsity-aware tiling. DRT continuously retiles sparse tensors at runtime based on the current sparsity of the active regions of *all input tensors*, to maximize accelerator buffer utilization while retaining the ability to co-iterate through tiles of distinct tensors.

Through an extensive evaluation over a set of SuiteSparse matrices, we show how DRT can be applied to multiple prior accelerators with different dataflows (ExTensor, OuterSPACE, MatRaptor), improving their performance (by 3.3×, 5.1× and 1.6×, respectively) while adding negligible area overhead. We apply DRT to higher-order tensor kernels to reduce DRAM traffic by 3.9× and 16.9× over a CPU implementation and prior-art tiling scheme, respectively. Finally, we show that the technique is portable to software, with an improvement of 7.29× and 2.94× in memory overhead compared to untiled sparse-sparse matrix multiplication (SpMSpM).

## CCS CONCEPTS

• **Computer systems organization** → **Special purpose systems**; • **Hardware** → Hardware accelerators.

## KEYWORDS

Tensor Algebra, Sparse Computation, Hardware Acceleration

**ACM Reference Format:**

Toluwanimi O. Odemuyiwa, Hadi Asghari-Moghaddam, Michael Pellauer, Kartik Hegde, Po-An Tsai, Neal Crago, Aamer Jaleel, John D. Owens, Edgar Solomonik, Joel S Emer, 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 (ASPLOS '23), March 25–29, 2023, Vancouver, BC, Canada.* ACM, New York, NY, USA, 15 pages. https://doi.org/10.1145/3582016.3582064



*ASPLOS '23, March 25–29, 2023, Vancouver, BC, Canada*



ACM ISBN 978-1-4503-9918-0/23/03.
https://doi.org/10.1145/3582016.3582064

## 1 INTRODUCTION

Tensor algebra—arithmetic with vectors, matrices, and higher-order tensors—is a ubiquitous primitive in numerical computations [3, 6, 38, 41, 47, 54]. In this work, we focus on the acceleration of sparse tensor algebra, in particular, the contraction of *multiple* sparse tensors. In the simplest case, this is the multiplication of two sparse matrices (SpMSpM), which has myriad applications in graph algorithms, solvers for linear systems and more [3, 6, 7, 14, 38, 41, 47, 54]. Beyond SpMSpM, sparse tensor contraction is an important primitive in many of the core areas of tensor computations, including computational chemistry methods [9, 33, 48] and sparse tensor decomposition [5, 23].

Operations on multiple sparse tensors are difficult to accelerate as they tend to be memory bound. Thus, maximum achievable performance is a function of arithmetic intensity, that is, the ratio of FLOPS to the data traffic (bytes transferred) from main memory. State-of-the-art sparse accelerators improve arithmetic intensity by first designing around a dataflow to improve *data reuse*, thus reducing DRAM traffic [16], and then making data representation and data orchestration decisions. Figure 1 shows the DRAM traffic of prior SpMSpM accelerators that explore using three main SpMSpM dataflows (outer-product [42, 61], row-wise Gustavson's [49, 60], and inner-product [30, 44]). Yet, as the figure shows, dataflow alone is not sufficient to bring DRAM traffic close to the lower bound.

For dense problems, significantly improving reuse (and therefore performance) beyond dataflow decisions is often achieved through *tiling*. Unfortunately, tiling sparse data for high data reuse is non-trivial. Consider a state-of-the-art scheme for sparse tiling, ExTensor [30], that statically tiles the input and output matrices offline into uniformly sized, *coordinate-space* regions, where coordinates correspond to the locations in Cartesian space such as row and column ids. Such a tiling is oblivious to data distribution. This can lead to *lower occupancy* tiles (i.e., those with few nonzeros), which leads to lower reuse per buffer fill and therefore lower performance.

To address the above challenges, we propose *dynamic reflexive tiling* (DRT), a novel tiling algorithm and hardware primitive that improves tile occupancy (and therefore reuse) in the presence of irregular sparsity.

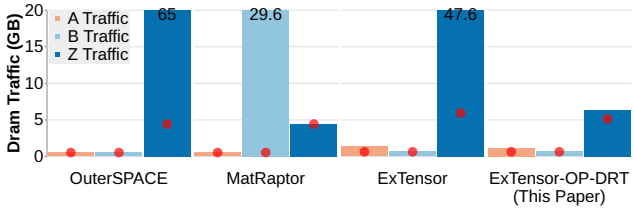



**Figure 1: DRAM traffic for each input operand ($A$, $B$) and output ($Z$) in SpMSpM ($A \cdot B = Z$), aggregated over the matrices used in our evaluation (Section 6) setting $B = A$. Each bar indicates actual traffic and each red square indicates the lower bound on traffic (read each of $A$ and $B$ once, write $Z$ once for each matrix). OuterSPACE [42], MatRaptor [49] and ExTensor [30] are representative accelerators that apply the outer-product, Gustavson's and inner-product dataflows, respectively. ExTensor-OP-DRT (this paper) using dynamic reflexive tiling achieves significantly closer to the lower bound for all operands/outputs. Different accelerators use similar but not identical compressed representations, and hence have slightly different traffic lower bounds.**

The key idea in DRT is to *dynamically tile* input and output tensors into *nonuniform coordinate-space* regions: tiles whose volumes differ when measured in coordinate space. As shown in Figure 1, this results in more efficient use of DRAM bandwidth.

Through dynamic nonuniform coordinate-space tiling, DRT takes into account data sparsity across all participating tensors to dynamically build tiles online. Tile occupancy, i.e., the number of non-zeros in the tile, is maximized, subject to the buffer capacity. Meanwhile, variation in occupancy across spatially distributed tiles is minimized. To maximize utilization across the entire duration of kernel execution, DRT not only changes tile shape across different regions of each tensor *but also over time for the same region*, based on how the data is later reused. For example, in SpMSpM ($A \cdot B = Z$), the ideal tile shape for a given set of rows in matrix $A$ changes *depending on the active set of columns for matrix B*.

A challenge that arises when tiling into nonuniform size regions is how to enable *co-iteration*. That is, when performing operations such as inner or outer products on tiles, participating tiles must have corresponding coordinate ranges. For example, inner-product matrix multiplication requires that the column coordinates in a tile of matrix $A$ match the row coordinates in a tile of matrix $B$. This facilitates operations such as coordinate intersections by ensuring that the set of coordinates from each tile in the intersection covers the same coordinate range.

To address the co-iteration problem, DRT *co-tiles* in the *coordinate space*. A co-tiling is one where co-iterated dimensions, shared between tiles mapped to each buffer, correspond to the same coordinate range in the original untiled tensors. Depending on the dataflow, co-tiling may require coordinating tile shape across many tiles. For example, if a tile of matrix $A$ is broadcast to all PEs, all tiles of $B$ later mapped to the PEs must be co-tiled with respect to that tile of $A$. This is nontrivial, as tile shapes become constrained as a function of one another and the available buffer space.

We propose an algorithm and hardware architecture to perform all of the above efficiently, including hiding the latency of dynamic tile construction. This is challenging, as finding optimal tile shapes implies performing a search that must be solved online and continuously for each set of tiles distributed to each accelerator buffer,

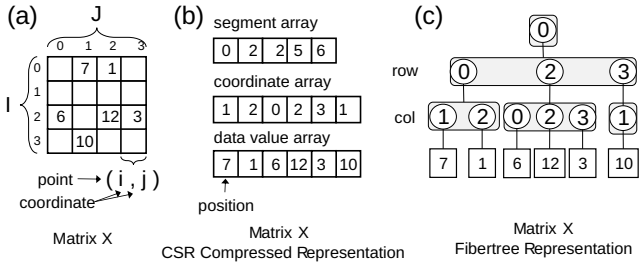



**Figure 2: (a) Tensor terminology using matrices as an example. (b) Example compression using CSR. (c) Fibertree representation of the compressed matrix.**

and each step in the search involves counting non-zeros from each untiled operand's compressed representation, e.g., CSR.

To more quickly find high-occupancy tile shapes that satisfy co-tiling constraints, DRT dynamically builds nonuniformly shaped *macro tiles* from statically-built and uniformly-shaped *micro tiles*, subject to co-tiling constraints. By working at micro tile granularity, DRT quickly creates high-occupancy macro tiles. By enabling buffers to store variable numbers of micro tiles via a single macro tile, DRT decouples buffer capacity from worst-case dense tile occupancy, enabling smaller buffers that better exploit data reuse.

To summarize, the paper makes the following contributions.

- We propose a novel sparsity-aware tiling algorithm, *dynamic reflexive tiling (DRT)*, that dynamically co-tiles input and output tensors to maximize buffer utilization. Moreover, we propose a hardware primitive, the *tile extractor*, which implements DRT using small hardware area.
- We integrate DRT into prior accelerators ExTensor, OuterSPACE and MatRaptor (representatives of major popular SpMSpM dataflows). Through an extensive evaluation over a set of SuiteSparse and SNAP matrices, we show how DRT enables significant performance improvement for these accelerators (by 3.3×, 5.1× and 1.6×, respectively). Using Accelergy [57], we model the area/energy overheads of DRT and demonstrate net energy improvement (over a design w/o DRT) with negligible area cost.
- We show the potential of DRT to improve computation on higher-order tensors, by demonstrating how DRT applied to ExTensor can reduce that design's memory traffic by 16.6× when running the Gram kernel.
- We evaluate the potential benefits of a software DRT implementation over an untiled (and statically tiled) software implementation, showing a 7.29× and 2.94× memory traffic improvement, respectively.

## 2 BACKGROUND AND TERMINOLOGY

We now provide background (summarized in Table 1).

### 2.1 Tensors, Tensor Kernels and Einsums

**Tensors:** Tensors are multi-dimensional arrays of arbitrary dimensionality $N$. We use the notation $N$-tensor for brevity. For example, 0-tensors are scalars, 1-tensors are vectors, 2-tensors are matrices. Figure 2 (a) shows an example matrix $X$ with dimensions $I$ (rows) and $J$ (columns). Each location in the matrix, identified by its row

and column numbers, is referred to as a *point*. An element in $X$ at point $(i, j)$ is written as $X_{ij}$. The row and column indices making up the point are called *coordinates*. We refer to the list and sizes of these dimensions—where size refers to the maximum number of coordinates in that dimension—as the tensor's *shape*. The previous example's shape is $I \times J$. Each point stores a scalar *data value*. For convenience, we denote dimension names and sizes with uppercase letters and index variables with the corresponding lowercase letters.

**Sparse tensor kernels:** We focus on the kernel of tensor contraction, which generalizes products of vectors and matrices. Tensor contractions are commonly described by 'Einsum' [21] (Einstein summation) expressions. The Einsum for matrix multiply is

$$Z_{ij} = A_{ik}B_{kj}. \quad (1)$$

This defines an iteration space of $I \times J \times K$. At each point in this space, the corresponding values of $A$ and $B$ are multiplied and stored in the corresponding $(i, j)$ point of the output. The contracted rank, $k$, implies a reduction over $k$ for each repeated $(i, j)$ output point. This can also be expressed as a sequence of dot products (made up of multiply-accumulates or *MACCs*) between vectors in the $I \times K$ matrix $A$ and the $K \times J$ matrix $B$, forming the $I \times J$ matrix $Z$. Unless otherwise stated, we assume tensors start at the *base* point $(0, \ldots, 0)$. Coordinate indices $i$, $j$, and $k$ range from 0 to the size given by their dimension, e.g., $0 \leq i < I$.

In Equation 1, dimension $K$ is called *contracted* [22] because a reduction occurs across it as indicated by the summation over $k$. We focus on the case where all tensors on the right-hand side of the Einsum are sparse. Then, implementations of sparse tensor contractions may avoid *ineffectual* computations (since $0 \cdot x = 0$) by performing intersections between coordinates of non-zero values [30]. Dimensions $I$ and $J$ are called *uncontracted*.

**Dataflow:** Given an Einsum, a kernel must now decide the order in which to traverse the iteration space. This order is typically expressed as loop nests. The loop order, which is an aspect of the *dataflow* [16], impacts data movement. For example, matrix multiply can be expressed as three loops, where the loop order is $J \rightarrow I \rightarrow K$ (iterate over $J$ in the outermost loop, $I$ in the second-to-outer loop, etc.). In this example, data in $Z$ is *stationary* [16], since $i$ and $j$ do not change in the innermost loop. Due to operation commutativity, these loops can be reordered, changing which operand is stationary. In general, for a given loop nest, we say a tensor is less stationary than another if it is indexed by a faster-changing index.

## 2.2 Compressed Representations

To reduce memory footprint, sparse tensors are stored in compressed representations. At a high level, these use *metadata* to reduce the cost of storing data values equal to zero. Because zeros are not stored, each non-zero's coordinates do not correspond to where the data value is physically stored, i.e., its *position*, in memory. The tensor's *footprint* is the memory storage needed for a representation, that is, the bytes of metadata and data for that format.

We assume tensors are stored in the T-[UC]$^+$ family of representations [30, 34]. In T-[UC]$^+$, a sequence based on the regular expression [UC]+ indicates whether each tensor dimension is Uncompressed or Compressed. For example, a matrix can have T-UU, T-UC, T-CU, or T-CC combinations [30, 34], where T-UU is a fully uncompressed representation. A compressed dimension conceptually stores pointers into the next dimension using coordinate-payload lists. In this classification, the popular CSR format (shown in Figure 2 (b)) is a T-UC representation. To hide details of the representation, we will explain ideas using the format-agnostic *fibertree representation* [53], which represents the tensor as a tree of coordinate-payload lists as shown in Figure 2 (c). In this form, each list of coordinates, e.g., coordinates 1, 2 in the first list of the second level, is called a *fiber*.

**Table 1: Terminology, which can be applied to tensors or tiles.**

| Name | Description |
|---|---|
| Data value | Float/double data values |
| Metadata | Non-data values required for the compressed representation (e.g., segment and coordinate array) |
| Coordinate | A logical location within a dimension |
| Point | Tuple of coordinates identifying a data value |
| Position | Memory location for a metadata or data value |
| Shape | List and sizes of a tensor's dimensions |
| Footprint | Tensor bytes for metadata + data |
| Occupancy | Tensor number of non-zero elements |

## 2.3 Tiling Compressed Representations

Sparse tensors can be tiled to improve reuse and arithmetic intensity by adding dimensions to the T-[UC]$^+$ representation [30]. To partition a matrix represented in CSR into two-dimensional tiles, we add two dimensions to the representation, giving us T-??UC, where '??' are two new outer dimensions that can be either U or C. Thus, each tile is represented in, e.g., CSR, and we traverse outer dimensions to index into each tile. Supporting multiple levels of tiles entails adding yet more dimensions.

Each tile has a base point. By convention, we refer to base points using parent tensor-relative coordinates. If the matrix in Figure 2 is tiled into $2 \times 2$ coordinate-space tiles, the tile base points are $(0, 0)$, $(2, 0)$, $(0, 2)$ and $(2, 2)$.

Prior work [30, 34, 60] performs *static, coordinate-based, uniform* tiling for simplicity. Static means the tiles are built for each tensor offline. Coordinate-based means tile size is specified in the coordinate (not position) domain. Uniform means for each tensor, each tile within that tensor has the same shape, i.e., same size measured in coordinate space.

More generally, we classify tiling schemes in a taxonomy given by **S**tatic/**D**ynamic-**U**niform/**N**onuniform-**C**oordinate/**P**osition where dynamic means chosen online and nonuniform means different tiles can have different footprints (if in position space) and shapes (if in coordinate space). We use this taxonomy as a shorthand. For example, an S-U-C tiling is Static-Uniform-Coordinate space and D-N-C is dynamic-nonuniform-coordinate space.

# 3 DYNAMIC REFLEXIVE TILING

We now describe dynamic reflexive tiling (DRT), an online heuristic to build D-N-C tiles. We define DRT for the general problem of sparse tensor kernels, expressed as an Einsum of sparse tensors. When operands are tiled, the calculation operates on a piece of the overall Einsum being computed, splitting the work into tasks that correspond to tile-wise calculations. Specifically, we define an *Einsum task* as a contraction of a set of tensors for a range of

coordinates. For example, using Equation 1, a task could be

$$Z_{ij} = A_{ik}B_{kj} \qquad i \in [I_0, I_1),\ j \in [J_0, J_1),\ k \in [K_0, K_1), \quad (2)$$

where $K_0 < K_1 \leq K$ and index $k$ iterates from $K_0$ to $K_1$ (same conventions for $i, I$ and $j, J$). In this task, a tile $A$ of shape $I_1 - I_0$ by $K_1 - K_0$ is multiplied by a tile of $B$ of shape $K_1 - K_0$ by $J_1 - J_0$, where the parent tensor-relative coordinates along the $K$ dimension match in both tiles. Completing the entire matrix multiply involves evaluating a set of tasks, each of which operates over a set of tiles that partition the compute space given by the original kernel (e.g., Equation 1).

## 3.1 Trade-offs in Changing Tile Shape

DRT's goal is to determine the shape of each tile—i.e., $I_0, I_1, K_0, K_1$, etc.—in each task so as to maximize buffer occupancy, which by extension reduces variation in buffer occupancy. This facilitates better data reuse. Depending on whether a dimension is contracted or uncontracted, changes to tile size along that dimension have different implications on tile footprint and data reuse. For simplicity, first consider the case when all tensors are dense. Increasing a dimension $G$ of a tile by a factor $\alpha$ increases the footprint of tiles participating in the Einsum task with a dimension indexed by $g$ (including the output) by $\alpha$. The tiles in the Einsum task of operands in which $g$ does not appear are unmodified, but the amount of computation in the Einsum task is increased by $\alpha$, hence the reuse achieved for the elements of these tiles increases by $\alpha$.

The presence of sparsity adds additional complexity to these tradeoffs. First, changing a dimension's size for a tile in coordinate space has an irregular impact on tile footprint. In some cases, increasing a dimension will not change tile footprint because all data values in the new tile region are zero. Second, when the tiles are sparse, only effectual computations contribute to the output. Thus, the output tile footprint is not known until after intersections are performed, and is difficult to predict/provision for before the intersections are performed. Nevertheless, even in the worst case, reuse increases monotonically with tile volume. In the best case, we expect to achieve the same improvements in reuse from tiling as in dense.

## 3.2 Reflexive Tiling Algorithm

**Algorithm 1** Dynamic reflexive tiling (DRT) pseudo-code.

```
   Inputs: tensors, base_points, loop_order
   Outputs: constraints, tile_sizes
   function DRT(tensors, base_points, loop_order)
       // Starting tile sizes are chosen statically at design time
 5:    tile_sizes = [initialTileSize(d) for d in allDims()]
       // Growing a dimension becomes a constraint on later tensors
       constraints = [None for d in allDims()]
       // Grow tensors into unused buffer space heuristically
       // Favor tensors whose tiles are resident longer
10:    for tensor in sortByStationarity(tensors, loop_order) do
           // Load the next tile using existing constraints
           try: tensor.loadNextTile(tile_sizes, constraints, base_points)
           catch TooLarge: return fallbackPath(constraints)
           // growDims() will update tile_sizes
15:        growDims(tensor, constraints, tile_sizes)
```

**Algorithm 2** Generic growDims() Function.

```
   Inputs: tensor, constraints, tile_sizes
   Outputs: constraints, tile_sizes
   // Grow until all dimensions of tensor are constrained
   function GROWDIMS(tensor, constraints, tile_sizes)
 5:    // selectDimToGrow: select which dimension to grow
       // and return None if all dimensions are constrained
       while d = selectDimToGrow(tensor.Dims(), constraints) is not None do
           // Grow if we haven't constrained this dimension
           if constraints[d] is None then
10:            // Try to grow this dimension by n and update the tile size
               try
                   // n → the amount to grow by, e.g.: 1
                   tile_sizes[d] = tensor.tryToGrow(d, n)
               catch TooLarge:
15:                constraints[d] = tile_sizes[d]
                   return fallbackPath(constraints)
```

We now describe how DRT uses the above tradeoffs to decide how to tile. Importantly, choosing a nonuniform tiling is (inherently) an online task, since nonuniform tile shapes are a function of all participating tensors. At the same time, the space of possible tilings is large. Prior work notes that finding an optimal tiling is NP-hard [2, 52, 58] and requires global visibility of the data. Thus, we face a tradeoff between spending time to find a higher-quality tiling vs. benefiting from that tiling. We propose a framework that enables greedy algorithms to choose tile shapes. We found that these strategies significantly reduce DRAM traffic while allowing us to overlap the process of building and computing on tiles (see Figure 1).

Algorithms 1 and 2 show pseudo-code for our scheme. For clarity, we describe the scheme focused on a single level of the buffer hierarchy, called the *fast memory*, and show the operations needed to form a single Einsum task (which specifies tile origins, e.g., $I_0, J_0, K_0$ in Equation 2). Given an Einsum task that contracts tensors $U, V, W, \ldots$, where tiles of $U$ are changed least frequently, $V$ second-least, etc., we begin with a static provisioning of the fast memory and an initial tile shape for each tensor (Lines 4–7 of Algorithm 1). In the main body of the algorithm (Lines 10–15), DRT proceeds tensor by tensor to determine tile shape along each dimension so as to maximize the number of non-zeros in each tile, prioritizing tiles that will be kept local in fast memory for longer, i.e., are more stationary (Line 10).

The algorithm then tries to grow the dimensions of each tensor (Alg. 1, Line 15 and Algorithm 2). Specifically, `growDims` first selects which dimension to grow, then attempts to grow it by a certain amount. Our implementation grows by one ($n = 1$ in Alg. 2, Line 13). There are a few approaches to selecting the order in which to grow dimensions (`selectDimToGrow` in Alg. 2, Line 7). We implement an approach that first tries to grow contracted dimensions in $U$ in a single pass, then grows uncontracted dimensions in $U$, then continues to contracted and then uncontracted dimensions in $V$, and so on, until the sum of tile footprints exceed buffer capacity. This approach enables relatively simple traversal of the compressed representation. It prioritizes maximizing output locality and minimizing output traffic, which tend to be the bottleneck in dataflows that require merging of partial output products [42]. Unless otherwise stated, this is the default DRT implementation for the rest of this paper.

Another approach, evaluated in Section 6.3 and 6.6, repeatedly alternates across the dimensions of a tensor, in order to maintain square tiles, until the buffer capacity is reached. This approach avoids shapes that are longer along one dimension and attempts to balance input/output locality. Other strategies are possible: beyond the sequence of dimensions over which to grow (Alg. 2, Line 7), one can change the starting tile shape (Alg. 1, Line 5).

All subdivisions are made in the coordinate space, and are akin to changing shape from Section 3.1. Specifically, each tile produced by DRT corresponds to a subtensor given by restricting the coordinate range of the original tensor along each dimension. Subsequent calls to DRT produce tiles for subsequent tasks. Tile base points from which to start building tiles change depending on the dataflow. For example, in an inner-product (output stationary) dataflow applied to Equation 2, the $K_1$ determined by the first call to DRT becomes the starting index for the $K$ dimension for the second call. In this way, successive calls to DRT tile the Einsum by considering the set of non-zero elements of the operand tensors involved in the computation of each Einsum task and selecting each tile's shape to be as large a hyper-rectangle as possible while still fitting into fast memory.

Since changing a dimension influences tile shapes for other tensors sharing that dimension (Section 3.1), it may not be possible to subdivide remaining dimensions while ensuring that all tiles fit into their allotted portions of the fast memory. This is represented by the try/catch blocks in Algorithm 1 and 2. In such cases, a fallback path is invoked. The fallback on Line 16 of Algorithm 2 is implemented as a continue (i.e., once we cannot grow further along one dimension, we try to grow along the next dimension). The fallback on Line 13 of Algorithm 1 is implemented by subdividing the most recent dimension of the tensor and recursively building new tiles on the subdivided tensor. This path is only invoked when particularly dense tiles from multiple tensors are encountered simultaneously.

*3.2.1 Coarsening and Hierarchy.* To reduce the effort it takes DRT to find a high-occupancy tiling, but potentially at the cost of finding the optimal tiling, the above scheme can be coarsened using an S-U-C tiling scheme, in which case scalar elements are replaced by tiles (called *micro tiles*) making the work a function of the number of micro tiles rather than the number of scalar elements. When coarsened to micro tiles, all sub-divisions occur at micro tile granularity. By physically tiling the tensor data into uniform micro tiles, the *macro tiles* produced by DRT are then treated as logical tiles. Macro tiles are then loaded into fast memory by collecting a set of micro tiles. This approach enables DRT to reuse micro tiles as constituents of distinct macro tiles that achieve maximal buffer occupancy for different Einsum tasks.

Finally, DRT can be applied hierarchically to achieve locality/load balance at different levels in the memory hierarchy.

## 3.3 Example for matrix multiplication

Figure 3a shows an example SpMSpM instance and how both DRT and the prior-art S-U-C tiling scheme [30] decompose it into tasks. This example assumes a $J \rightarrow K \rightarrow I$ dataflow—i.e., matrix $B$ is stationary. We assume a buffer (fast memory) that can fit up to 2 data values for each of matrix $A$ and $B$. Thus, the only tile shapes available to the S-U-C baseline are $2 \times 1$ and $1 \times 2$—as any larger shape would not fit the data in the worst case.

Algorithm 1 builds tiles for the first task in steps ⓐ–ⓒ. Each call to Algorithm 1 determines the next task's tile shapes for $B$ followed by $A$. We assume the initial tile size has shape $2 \times 1$ for $A$ and $1 \times 2$ for $B$ (Alg. 1, Line 5). To form task (1), whose tile base points are $I = J = K = 0$ (Alg. 1, Line 4), DRT (ⓐ) grows the $B$ tile along the $K$ (contracted) dimension, then (ⓑ) along the $J$ (uncontracted) dimension when it cannot grow further along $K$. DRT then builds the tile for $A$: due to `constraint[k]`, it sets the $K$ dimension to match (co-tile) that of $B$ (ⓐ), then ⓒ grows along the $I$ dimension until the buffer partition for $A$ is full. We form subsequent tasks (2) and (3) through subsequent calls to Algorithm 1, specifying the tile base points in the order given by the dataflow (`loop_order` in Algorithm 1, Line 1).

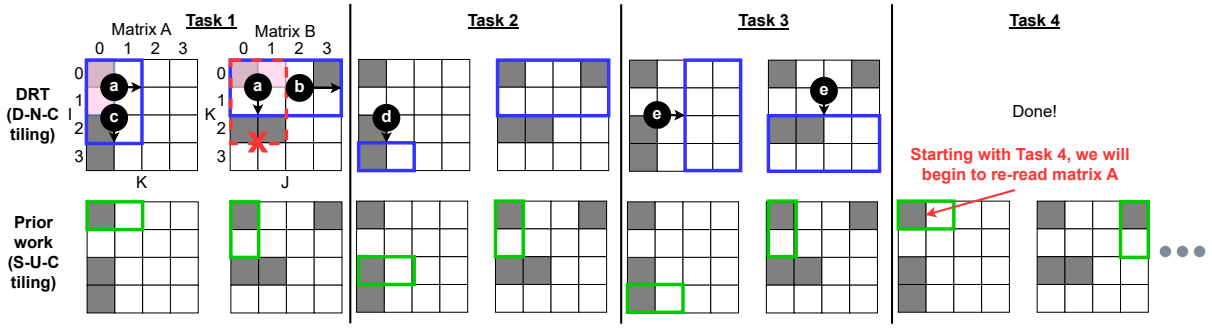
Figures 3b and 3c build on this illustration to show the lower-level actions of DRT on example compressed formats. We assume the shaded regions in 3a are scalars and the dataflow is $J \rightarrow K \rightarrow I$. For *concordant traversal* (i.e., traversal in the same order as the layout of the data format) [53], $A$ is in compressed sparse column (CSC) format [17] ($K$-major) and $B$ is in CSC format ($J$-major). Figure 3b shows the initial values of various variables from Alg. 1 and Figure 3c shows the way they change over time. Section 4.3 further discusses how the related hardware interacts with these formats.

**Comparison to S-U-C: Where does the benefit come from?** The key difference between DRT and the S-U-C tiling scheme is that DRT can increase the coordinate range of each tile beyond the buffer capacity in the presence of irregular sparsity. For example, the first tile of $B$ for DRT spans a $2 \times 4$ coordinate range because there are only two non-zero values in that range. The baseline (S-U-C) is limited to smaller tile shapes. Larger coordinate-space tiles can translate into memory traffic reduction. In this example, DRT completes SpMSpM after reading matrix $A$ once. Conversely, the baseline must read $A$ twice (see red text in the figure for Task (4)). Since SpMSpM is typically memory-bandwidth bound, this traffic reduction translates into increased performance.

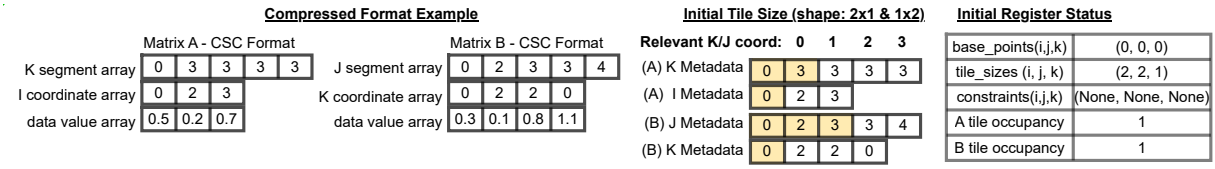
# 4 THE TILE EXTRACTOR

Accelerators have been particularly efficient in the sparse domain; thus, we focus on a hardware unit, called the *tile extractor*, that implements DRT while accounting for pipelining effects and buffer management. We further explain how to integrate the tile extractor into an accelerator buffer hierarchy. Section 6.2 looks at the impact of these changes to three representative accelerators.

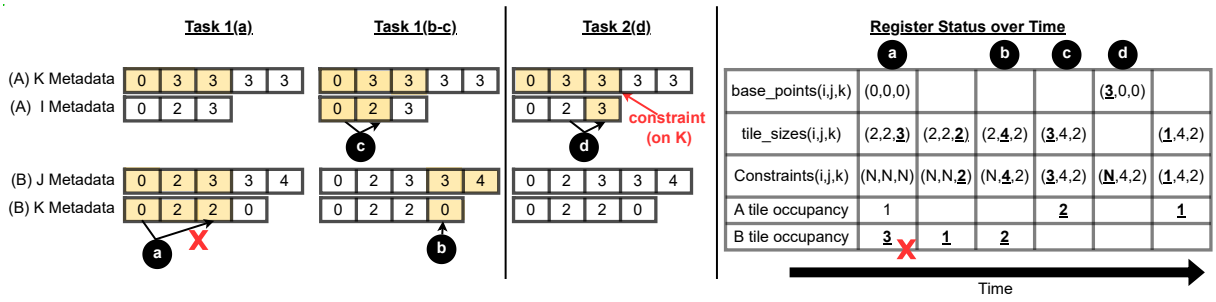
*Sparse Data-Orchestration Partitions (S-DOPs).* Figure 4 depicts an accelerator composed of multiple levels of on-chip buffers, routing fabric between the buffers, and compute resources at the lowest-level buffer [15, 16, 30, 42–44, 49, 60]. Buffers and compute at the lowest level are called processing elements (PEs). We note a common accelerator design pattern where each memory level contains surrounding logic for reading, distributing, and computing on compressed formats. We refer to these buffers and logic at the non-PE levels as *sparse data-orchestration partitions (S-DOP)*, which store




**(a) Top: Dynamic reflexive tiling example for SpMSpM, showing tile shapes for tasks (1)–(4). Arrows ⓐ–ⓒ show the order of tile growth in task (1), while ⓓ and ⓔ show how tile shape changes across subsequent tasks. The red, shaded squares show the initial tile shape (Alg. 1, Line 5) of 2×1/1×2 in in task (1). Bottom: An S-U-C tiling baseline (e.g., ExTensor [30]). The fast memory (buffer) is statically partitioned between $A$ and $B$, and each partition can store $\leq 2$ non-zero values. Shaded/un-shaded squares are non-zeros/zeros. Tasks involving empty tiles are skipped.**




**(b) Left: The compressed representation of $A$ and $B$ in DRAM assuming representations that are easily traversed by the $J \rightarrow K \rightarrow I$ dataflow. Data values are arbitrary. Middle: For initial tile shapes of 2×1 and 1×2, the highlighted regions match the red tiles in Figure 3a. Right: Initial status of counters for DRT.**




**(c) Left: Metadata memory accesses for tasks 1 and 2. Highlighted boxes indicate read accesses. The red X indicates an occupancy check failed as the current tile exceeds buffer capacity. Right: Changing counter values as DRT grows each tile. $N$ stands for "None" (Alg. 1, Line 7). The red X denotes a canceled operation due to buffer overflow (Alg. 2, Lines 16). Values that changed between one step and the next are underlined.**

**Figure 3: DRT example with CSR/CSC formats. See sections 3.3 and 4.3 for details.**

and distribute tensor metadata and data for distribution to next-level buffers. For example, if we map ExTensor to this accelerator hierarchy, its S-DOP-like logic forms and distributes data and metadata as uniform coordinate-space (S-U-C) tiles. S-DOP logic may be implemented using address generators [15, 16, 29, 31, 43] or finite-state machines capable of traversing compressed representations [30]. We design the tile extractor as a hardware unit within the S-DOP (Figure 4). Each tile extractor implements all of Algorithm 1 and multiple S-DOP levels apply DRT hierarchically. That is, following the accelerator template, the tile extractor in the DRAM S-DOP dynamically breaks tensors (kernels) into nonuniform coordinate-space D-N-C tiles (tasks), the tile extractor in the Global Buffer S-DOP breaks these D-N-C tiles into D-N-C sub-tiles (sub-tasks), etc.

The rest of the section is discussed with respect to the first level of hierarchy (when tensors and kernels are broken into tiles and tasks). Tile extractors interact with their local S-DOP buffer, which stores tensor compressed representations, to determine the shape

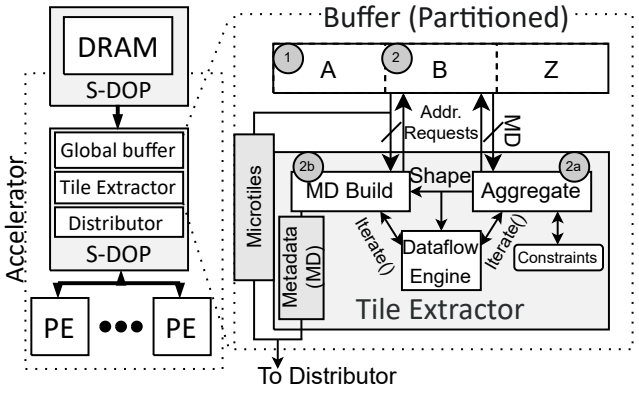



**Figure 4: (Left) Accelerator template composed of PEs/S-DOPs. (Middle) Tile Extractor units.**

of each tile involved in the next task. This process, discussed in

detail in Section 4.2, is non-trivial, as calculating how tile footprint changes as a function of tile shape requires a variable and possibly substantial number of reads into the tile's compressed representation.

## 4.1 Micro Tile-Granular Tile Extraction

Supporting coarsening (Section 3.2.1) in hardware is relatively straightforward. Pre-processing input tensors into micro tiles is equivalent to the pre-processing needed by prior S-U-C tiling schemes (Section 2.3). To support DRT (Section 4.2), we augment T-[UC]$^+$ metadata to include micro tile footprints, stored alongside the coordinates for each micro tile (see Figure 5). This ensures that as the tile extractor iterates through the compressed representation, it knows the underlying micro tile footprint without introspection of the micro tile's metadata.

Although micro tiles are S-U-C, accelerator performance is less sensitive to their exact shape than in a traditional S-U-C tiling scheme (e.g., ExTensor [30]). For example, buffers in ExTensor are explicitly managed [43], similar to scratchpads. Thus, the buffer capacity must be sufficiently large to fit the worst-case tile, i.e., when a region of the tensor is dense. This creates an undesirable trade off: smaller/larger buffers are more/less efficient to access, but imply less/more maximum reuse (which is a function of S-U-C tile shape).

Conversely, D-N-C decouples buffer size from the tile shape. With DRT, a single macro tile contains an arbitrary number of micro tiles, subject to capacity and co-tiling constraints. Micro tile shape can be small, and tuned based on what DRT needs to quickly determine macro tile shape.

## 4.2 Implementation

There are three steps in the tile extractor's implementation:

*4.2.1 Aggregate.* The Aggregate module determines which micro tiles should form each macro tile. Following Section 3.2, we statically partition the S-DOP buffer by each operand and output tensor (①, ② in Figure 4). The goal of Aggregate (②a, Figure 4) is to choose macro tile shapes such that each macro tile's footprint maximizes buffer partition occupancy without exceeding capacity, and while respecting co-tiling constraints.

As the tile extractor tries to expand along contracted and uncontracted dimensions (Algorithm 2, Figure 4), Aggregate reads the tensor's compressed representation and uses an accumulator to track the current macro tile footprint. Setting each dimension might require reading the compressed representation for multiple micro tiles. In that case, Aggregate reads the compressed representation in raster order. To improve throughput, our implementation reads $P$-word wide vectors (our evaluation uses $P = 32$) of the compressed representation and provisions a $P$-to-1 parallel adder to determine the occupancy of $P$ micro tiles each cycle.

*4.2.2 Metadata build.* Since tiling is hierarchical, sub-tiles formed via DRT must have proper T-[UC]$^+$ metadata to be interpreted correctly by the next level of S-DOPs. Metadata build takes the micro tile coordinates produced during the aggregate step and uses them to construct a macro tile in a T-[UC]$^+$ representation. In our implementation, we implement T-[UC]$^+$ representations using coordinate and segment array data structures similar to CSR/CSC and CSF (Section 2, see Figure 5). The metadata build unit builds these data structures from the bottom up. It starts with the coordinate and segment arrays that directly refer to micro tiles, then builds up to the root of the sub-tensor formed by the macro tile (②b in Figure 4).

*4.2.3 Pipelining.* After i) Aggregate and ii) Metadata build complete, the new macro tile (its metadata and micro tile data) is iii) Distributed to the next-level S-DOP.

Pipelining occurs at two levels. First, we overlap Distribution for tile $i$ and Aggregate+Metadata build for tile $i+1$ by adding a second port to S-DOP local buffers. Distribution typically dominates as it needs to read tile metadata *and data*, as opposed to just metadata. Thus, this strategy can usually hide Aggregate and Metadata build costs. Second, task formation in S-DOP level $j$ is overlapped with task processing time in level $j-1$ (where level 0 = the PEs). Thus, as long as the time to evaluate the task (e.g., MACC, intersection operations) rate matches tile build, the latter cost is likewise hidden.

## 4.3 Compressed Format Example

In task (1) of Figure 3c, Aggregate first reads the $J$ segment array of $B$ for $J = [0, 2)$, then sequentially traverses the $K$ coordinate array to see the number of non-zeros that fall under `tile_sizes[k]` $\leq 2$. Since there are 2 new non-zeros with $K$ coordinates less than or equal to 2 (see ⓐ column of the register status in Figure 3c), the accumulator for $B$ (`B tile occupancy` in Figure 3c) flags a buffer overflow and the Aggregator reverses the operation.

Figure 5 expands the memory layout to the accelerator hierarchy. Assume a $J \rightarrow K \rightarrow I$ dataflow from DRAM to the LLB (CSC/CSC for $A/B$), a $K \rightarrow I \rightarrow J$ dataflow from the LLB to the PEs (CSC/CSR for $A/B$), and an $I \rightarrow J \rightarrow K$ dataflow within the PEs (CSR/CSC for $A/B$). The DRAM S-DOP shows the memory layout for pre-processed tiled-CSC (T-[UC]$^+$) matrices (Section 4.1). Shaded regions from Figure 3a now represent *micro tiles* of shape $3 \times 3$ (arbitrarily chosen) instead of scalars. Metadata now includes micro tile footprints (`micro tile sizes`). Additionally, the data array of scalars is replaced with an array of *pointers* to micro tiles. Micro tiles are stored as standard $3 \times 3$ CSR/CSC matrices elsewhere in memory (e.g., `@addr0` in Figure 5).

In the DRAM S-DOP, Aggregate processes metadata in the same manner as Figure 3c. However, rather than accumulating by one as the tile grows, Aggregate increments by the the augmented micro tile footprint. Steps ⓐ–ⓑ under the DRAM Aggregate unit correspond to the same memory access patterns as task (1) in Figure 3c. For this compressed format, access within a step is sequential for each array. While Aggregate is determining B's tile shape (B1 macro tile), the Metadata (MD) build unit can begin creating the segment and coordinate arrays for the corresponding macro tile (MD build, ⓑ–ⓒ). It recomputes macro tile metadata (I, J, K coordinates) to start at base points of 0. In parallel, the Distributor streams the relevant micro tile pointers to the LLB S-DOP. It also sends the in-flight segment, coordinate, and footprint information computed by MD build. At the LLB S-DOP level, Aggregate can begin determining tile shapes once it starts receiving the initial metadata information for $B1$. The Distributor at this level schedules pairs of micro tiles to PEs for compute (Distribute, ⓔ–ⓖ).

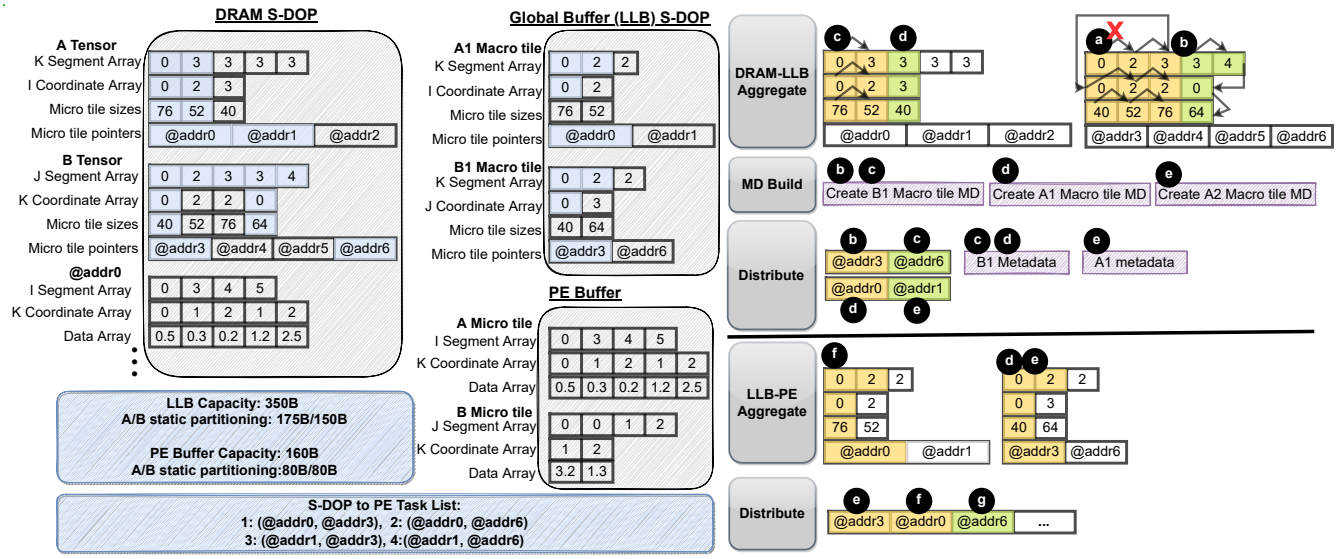



**Figure 5: Left: Example memory layout of two tensors at each level of the accelerator hierarchy. At the DRAM, assume $A$ and $B$ are the same as Figure 3a, but the shaded gray regions are now $3 \times 3$ *micro tiles* (shape arbitrarily chosen) instead of scalars. We explicitly add micro tile footprint (micro tile sizes array) to the representation. @addrX refers to the memory address location of a micro tile. Highlighted blue regions indicate the involved tiles for the first task at each S-DOP level. Right: Hardware actions and memory accesses at each time step, where we label time steps, in alphabetical order, as (a)–(g). Gold and green regions map to memory accesses into a tensor/tile for the labelled time step.**

# 5 METHODOLOGY

To evaluate DRT: (1) We first integrate DRT into the state-of-the-art S-U-C-based ExTensor [30] for a subset of linear algebra, graph analytics, and tensor kernels. (2) We evaluate DRT's potential in improving the performance of *other* accelerators that are built around *other* dataflows and *do not* natively tile, by integrating the tile extractor into those accelerators. (3) We implement a software variant of DRT and S-U-C and compare their memory overheads to an untiled SpMSpM implementation.

## 5.1 Workloads, metrics and datasets

*5.1.1 Arithmetic intensity and DRAM-bound performance:* We use the term *DRAM-bound performance* to mean the highest possible performance or throughput given the current arithmetic intensity (i.e., ideal use of on-chip compute).[1] This statistic is shown as the red dots in Figures 6, 7, 8, 10. We calculate arithmetic intensity as the number of *effectual* MACCs divided by the DRAM traffic of a particular workload on that accelerator. A given workload has the same number of effectual MACCs across all accelerators. Thus, red dots in the following figures also indicate how much DRAM traffic is reduced—and how much data reuse improves—over their corresponding baselines.

*5.1.2 Kernels:* Let $S$ be a square and sparse matrix, $F$ a tall-skinny sparse matrix, and $\chi$ a 3-D tensor. Given these inputs, we evaluate these kernels:

• *Linear Algebra—SpMSpM*: We evaluate SpMSpM for a matrix multiplied by its transpose for multiple input shapes: a square matrix ($S^2$), a tall-skinny matrix ($F \cdot F^T$), and a short-long matrix ($F^T \cdot F$). The square of a matrix ($S^2$) is used in Markov clustering [7, 41, 54] and in evaluating SpMSpM software implementations and accelerators [42, 61]. Kernels with non-square matrices ($F^T \cdot F$, $F \cdot F^T$) are found in applications such as the Jaccard similarity index [5, 11, 23, 27].

• *Graph analytics—Multi-source BFS*: We further evaluate multi-source parallel breadth first search (MS-BFS) and betweenness centrality which involve SpMSpM between a square matrix ($S$) and a tall-skinny matrix ($F$), or a short-long matrix by a sparse square matrix [3, 14, 41, 47]. We run MS-BFS (all iterations) with randomly selected initial source nodes in the frontier matrix ($F^T$) and set the aspect ratio of columns to rows [41] (which determines parallelism) to $2^7$, $2^9$ and $2^{11}$. Filtering of visited nodes after each iteration is performed offline and not included in runtime.

• *Tensor Algebra—Gram*: Beyond SpMSpM, we evaluate a higher-order tensor kernel called Gram, a common sub-routine for computing a Tucker decomposition [5, 23]. This is done by contracting the 3-tensor, $\chi$, with itself over two indices $(j, k)$, given by the Einsum $G^{(0)}_{i,l} = \chi_{ijk}\chi_{ljk}$ where $G^{(0)}_{i,l}$ has shape $I \times I$, and $l$ ranges from $[0, I)$.

*5.1.3 Datasets.* We use a subset of matrices from the SuiteSparse Matrix Collection [19]. The matrices represent real-world graphs with varying densities (from 0.0006% to 0.356% dense) and sparsity patterns.

## 5.2 Evaluation framework and baselines

Our performance evaluation showcases how DRT can be incorporated into prior accelerators, regardless of dataflow, and reduce their memory traffic. Source code is on GitHub at https://github.com/FPSG-UIUC/DRT [1].

[1] Recall, SpMSpM is typically memory bound. Thus, peak performance is a function of arithmetic intensity.

*5.2.1 Study 1: High-fidelity (cycle-level) analysis of ExTensor+DRT and comparison to CPU.* Our main result demonstrates how DRT can augment and improve ExTensor—the prior-art scheme in sparse tiling [30].

*Cycle-level simulation of ExTensor and DRT.* We model ExTensor using cycle-level simulation, including all on-chip components (buffers, intersection units, etc.). For configurations using DRT, we model the tile extractors (Section 4) at cycle-level. We use queuing models for the network-on-chip (NoC), buffers, and DRAM—which ensure data transfers are not allowed to exceed peak bandwidth. We note that ExTensor's memory access patterns (with and without DRT) exhibit high spatial locality and regular communication patterns (e.g., multicast) [30]. Thus, a queuing model is sufficient to capture memory-level effects (e.g., DRAM burst length). To ensure the simulator is functionally correct, we validate the output sparsity produced by the simulation against the results from Intel MKL.

*ExTensor architecture.* ExTensor uses a 3-level memory hierarchy (DRAM, global buffer, PE local buffers) with S-U-C tiling at each level. We evaluate two variants of ExTensor. First, the original design (denoted ExTensor) [30]. Second, a variant (ExTensor-OP) which uses an outer-product dataflow between the global and local buffers that removes a performance bottleneck we found in the original design. To reduce output traffic from outer-product, ExTensor-OP uses multiply-and-merge [42]; however, it performs local reductions of partial sums in output tiles until those tiles need to be spilled to memory. In addition, it uses a parallelized variant of ExTensor's skip-based intersection unit.

For ExTensor and ExTensor-OP's S-U-C tiling, we sweep different static tile shapes *for each workload* and use the shape that performs best *per workload*. Thus, our evaluation represents a best-case scenario for an S-U-C scheme.

*Integrating DRT.* We integrate DRT into ExTensor-OP, forming ExTensor-OP-DRT (or **TACTile**). Specifically, logic responsible for filling the global buffer and PE local buffers is modified to implement the S-DOPs with tile extractors from Section 4. Thus, DRT subdivides tiles twice. First, an S-DOP builds D-N-C macro tiles from DRAM to be stored in the global buffer. Each of these tiles are broken down into macro sub-tiles to be distributed to the PE buffers.

*CPU baseline.* We show results relative to Intel's MKL, evaluated on a 3 GHz (3.5 GHz turbo) Intel Xeon-E5-2687W with 12/24 cores/threads, a 30 MB LLC, and 4 DRAM channels with a peak bandwidth of 68.25 GB/s.

*Normalizing accelerator configurations.* Both accelerators run at a 1 GHz on-chip frequency with 128 PEs. We set accelerator DRAM bandwidth and on-chip storage (30 MB global buffer, 32 KB local buffers) to match the CPU LLC and local caches. Note, we found in our area analysis (Section 6.5) that on-chip memory dominated total area. Thus, we use a design-time search for the number of MACCs relative to the buffers to determine the PE count of 128.

*5.2.2 Study 2: DRAM-traffic analysis of OuterSPACE+DRT and MatRaptor+DRT.* DRT can be incorporated into and reduce the DRAM traffic of additional accelerators that represent different major SpMSpM dataflows. We evaluate DRT integrated into OuterSPACE [42] (an outer-product dataflow) and MatRaptor [49] (a row-wise Gustavson's dataflow). We idealize these accelerators' on-chip implementations, assuming they can reach their DRAM-bound performance. As such, these evaluations also provide insight into the potential performance improvements on SpArch [61], GAMMA [60], InnerSp [8], and other SpMSpM accelerators designed around the same dataflows. For each accelerator, three variants are evaluated: an untiled variant (which each paper originally reported), a variant that uses S-U-C tiling, and a variant with DRT (D-N-C tiling). The latter variants apply a single level of tiling across all dimensions.

*5.2.3 Study 3: SW Implementation/Traffic Analysis of DRT.* Given that DRT is a primitive that can be applied in a variety of accelerators, a natural question is whether it can be used outside of accelerators. We leave a thorough study for future work; however, we implement SW variants of S-U-C and DRT and perform an oracle, best-case analysis to determine their potential in improving SpMSpM memory traffic.

*5.2.4 DRT configuration time parameters (all studies).* At configuration time, all on-chip buffers (LLB and PE buffers) are statically split across all tensors (see ① and ② in Figure 4), with the same partition used for all workloads. For example, $A$ might get 5% of the buffer, $B$ 45% of the buffer, and $Z$ 50% of the buffer. Likewise, input matrices are pre-processed into micro tiles of shape $32 \times 32$ for all workloads. We chose these partitions/micro tile shape based on a sweep that resulted in best average performance.

# 6 EVALUATION

## 6.1 Study 1: Main Results

*6.1.1 Linear Algebra (SpMSpM).* Figure 6 compares the performance between ExTensor-OP-DRT, ExTensor-OP, and ExTensor relative to the CPU for ($S^2$). On average, ExTensor-OP-DRT performs 1.7x and 2.4x better than ExTensor-OP and ExTensor, respectively. Note that the only difference between ExTensor-OP-DRT and ExTensor-OP is in the tiling mechanism (DRT). *The main takeaway—which holds in many of the other evaluations we perform—is that ExTensor-OP-DRT's average **actual** performance exceeds other configurations' DRAM-bound (oracle) performances (red dots).* This showcases the benefit of ExTensor-OP-DRT's improved data reuse capabilities.

Workloads *bcsstk17* and *p2p-Gnutella31* fit entirely in the LLB. As expected, they have similar arithmetic intensity and speedup across S-U-C and DRT, since data need only be fetched once from main memory in both cases. For other workloads, ExTensor-OP-DRT consistently decreases the DRAM traffic, thereby increasing the achievable peak throughput (red dots) compared to ExTensor-OP. By maximally filling the on-chip buffer on each iteration of the compute space, DRT reduces the number of passes required over the tensors. The improvement in actual performance of ExTensor-OP-DRT over its static variant for workloads with unstructured patterns (right of the red line) points to the pre-compute, load balancing (or data-balancing) aspect of DRT in distributing tiles such that PEs are maximally occupied.

Additionally, compared to ExTensor-OP-DRT, ExTensor-OP nearly reaches its peak throughput for all workloads. For sparser

matrices, ExTensor-OP-DRT reaches its peak, but as matrices become more dense in both pattern regimes, a gap occurs between achieved performance and peak-achievable performance. On-chip effects such as intersection logic and round-robin distribution of tasks across PEs become prominent. There is further opportunity in optimizing on-chip logic to better convert improved data reuse into speedup.

**Tall-skinny matrices**: When SpMSpM involves tall-skinny matrices (Figure 7), ExTensor-OP-DRT improves performance by 3.5x, 3.5x, and 5.2x over CPU MKL, ExTensor, and ExTensor-OP, respectively. The arithmetic intensity of ExTensor-OP-DRT is 4.2x and 5.1x greater than that of ExTensor and ExTensor-OP, respectively, indicating an improvement in data reuse. Interestingly, static tiling at times does worse compared to CPU MKL, while DRT is able to consistently find tile shapes that reduce memory overhead. As explained in Section 5, our S-U-C configurations sweep tile shapes and use whichever achieves best performance. However, any static tile shape may lead to poor overall performance, e.g., if the matrix has irregular sparsity.

For the tall-skinny *cit* workload, we found that ExTensor-OP-DRT improves input traffic at the expense of increased output traffic, resulting in a net loss compared to ExTensor in this specific case. Additionally, the input tensors of both *p2p-Gnutella* workloads fit entirely into the on-chip buffer. As such, the results are an artifact of the difference in output handling between the two accelerators.

ExTensor-OP-DRT tends to reach DRAM-bound performance for workloads where the first operand ($A$) is short and long, but has larger headroom when $A$ is tall and skinny. Note that when $A$ is tall and skinny, the system must evaluate more micro tile-granular tasks compared to short-long. This causes the tall-skinny workloads to be more sensitive to intersection time. Coincidentally, further experiments indicate that the headroom is closed by an oracle intersection unit.

*6.1.2 Graph Analytics (MS-BFS).* Figure 8 shows the relative speedup for the combined BFS iterations per workload. On average, ExTensor-OP-DRT is 3.6x and 5.5x faster than ExTensor and the CPU, respectively. Matrices with a larger row length variation, such as the last 6 workloads in Figure 8 (ranging from 3 to 12 in row variation), have an average speedup of 7.2x compared to an average speedup of 2.7x for the remaining matrices with row variations of less than 2. At lower variation, an S-U-C accelerator is able to mine significant reuse since static tile shapes will have similar footprints.

*6.1.3 Tensor Algebra (Gram).* To understand the opportunity in reducing DRAM traffic for higher-order tensors, we run ExTensor-OP and ExTensor-OP-DRT for the Gram computation (Section 5.1.2). Figure 9 shows the arithmetic intensity compared to TACO [34] for ExTensor-OP and ExTensor-OP-DRT over a subset of 3D tensors from the FROSTT suite [46] and 3D tensors generated using Benson et al.'s framework [10]. Rather than growing along two dimensions, as in prior kernels, DRT must now grow across three dimensions, two of which are contracted (Section 5.1.2). We generate the CPU result by passing the Einsum to the TACO compiler [34]. On average, ExTensor-OP-DRT shows a 16.6x and 3.9x arithmetic improvement over ExTensor-OP and TACO, respectively. Again, note that S-U-C will not always see benefits over the CPU as its performance depends on the chosen static tile size. In particular, the gap between S-U-C and DRT decreases as density increases. This is expected, as at lower sparsity, a dynamic tiling strategy should be better able to collect sparse tiles together for reuse.

## 6.2 Study 2: Portability To Accelerators

Figure 10 (top) shows the performance comparison between OuterSPACE variants, with and without DRT. The red dots indicate that tiling increases arithmetic intensity by 3x and 5.1x in S-U-C-based schemes and DRT, respectively, over the baseline. This improvement is due to tiling $A$ and $B$. The untiled baseline (original OuterSPACE proposal) distributes columns of $A$ and rows of $B$, giving $A$ and $B$ perfect reuse, but $Z$ poor reuse. Tiling of $A$ and $B$ reduces the working set size of output partial products, allowing them to be partially reduced on-chip, which reduces memory traffic. Additionally, tiling enables partial reuse across all three tensors.

Figure 10 (bottom) shows the same for MatRaptor. The baseline tiles along the $M$ dimension, yielding perfect reuse on $A$, poor reuse on $B$, and partial reuse on $Z$. Using S-U-C and DRT tiling increases $B$'s input reuse which in turn reduces overall DRAM traffic.

In both cases, the untiled baseline (dashed-line) assumes ideal on-chip behavior. The actual untiled baseline's performance will be lower. Importantly, the DRAM-bound limit with DRT is greater than the untiled baseline in nearly all cases. For the actual throughput (height of each bar), we use a round-robin distributor to choose which PEs evaluate each task. This is not fundamental, but can lead to poor load balancing. With a more sophisticated work-distribution strategy, we believe the actual performance will approach the ideal.

## 6.3 Study 3: Software Best-Case Analysis

Although we propose DRT as a hardware unit for accelerators, we also evaluate its potential in a software environment. We implement DRT for the CPU, apply it to SpMSpM, and track its memory traffic. This variant follows an inner-product dataflow when computing on macro tiles in the LLC. Inner-product has perfect reuse on the output; thus, we use the alternating DRT variant as it promotes reuse on the inputs. Figure 11 shows the memory overhead of DRT and S-U-C, compared to the untiled CPU SpMSpM implementation. DRT provides a 7.29x and 2.94x improvement over no tiling and S-U-C tiling, respectively. For diagonal matrices, the gap between S-U-C and DRT decreases as input density increases. We expect this as S-U-C is better able to exploit reuse with denser tiles. At lower density, DRT is better able to collect sparse tiles to maximally fill the on-chip LLC. Likewise, for the random, unstructured pattern workloads, DRT consistently outperforms S-U-C. Tiling provides no benefit for two workloads (circled in red). These two workloads have a metadata overhead of over 8x their untiled variants, leading to an increase in raw DRAM traffic which tiling is unable to overcome. This is not fundamental and rather due to the U rank in the micro tiles' T-UC representations. We expect a T-CC representation will resolve this.

## 6.4 Bandwidth Scaling Study

In Section 6.1.1 we note that intersection was one of the potential bottlenecks. We now evaluate that claim. Figure 12 analyzes how a DRT-based accelerator's performance scales as memory bandwidth

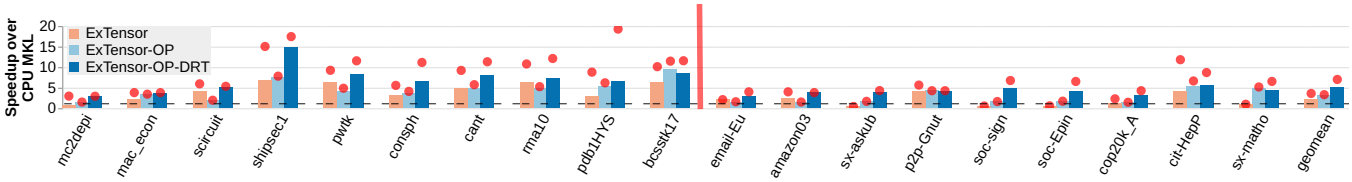



**Figure 6: ExTensor-OP-DRT, ExTensor-OP, and ExTensor performance relative to CPU MKL ($S^2$). Workloads are in two groups: to the left of the red line are matrices with a predominant diamond band sparsity pattern, and to the right are matrices with unstructured patterns. Within each sparsity pattern group, matrices are sorted by increasing input density. Red dots indicate DRAM-bound performance given the arithmetic intensity of that workload.**

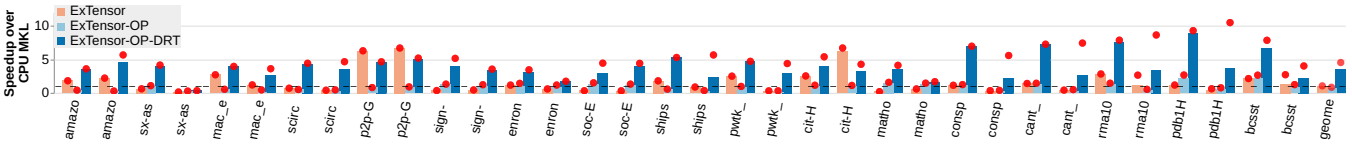



**Figure 7: ExTensor, ExTensor-OP, and ExTensor-OP-DRT performance ($F^T \cdot F$ and $F \cdot F^T$). Workloads are sorted by increasing input density. The first workload is a short-long matrix by its transpose and the second workload is a tall-skinny matrix by its transpose. Red dots indicate DRAM-bound performance.**

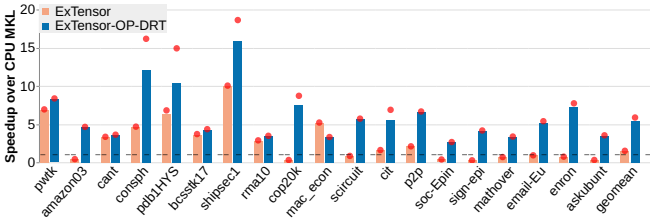



**Figure 8: Accelerators ExTensor and ExTensor-OP-DRT performance across all MS-BFS iterations ($F^T \cdot S$). Workloads are sorted in increasing coefficient of row variation (of $S$), a measure of the variance in the number of non-zeros of each matrix row [39]. The aspect ratio of matrix columns to rows is $2^7$. Results for aspect ratios of $2^9$ and $2^{11}$ follow a similar pattern. Red dots indicate DRAM-bound performance.**

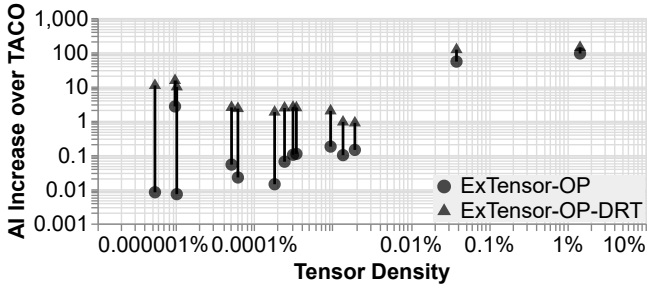



**Figure 9: Arithmetic intensity compared to TACO [34] for the Gram computation for S-U-C and D-N-C. Higher is better. Each S-U-C, D-N-C pair indicates the gap in arithmetic intensity, and thus DRAM traffic, for a given workload.**

increases, i.e., by raising the roof. To show an upper bound on bandwidth scaling, we compare three variants of ExTensor-OP-DRT: (1) ExTensor-OP-DRT using the serial, Skip-Based intersection unit from ExTensor [30], (2) ExTensor-OP-DRT with a parallelized intersection unit, (3) ExTensor-OP-DRT with an ideal "Serial-Optimal" intersection that enables one MACC per cyle per PE, regardless of sparsity pattern. We show Serial-Optimal to avoid any intersection-related bottleneck, i.e., to visualize potential.

We observe two things. First, performance scales as DRAM bandwidth improves. This suggests that DRT enables workloads to be sufficiently close to the roofline and be bandwidth limited. Secondly, performance is still bottlenecked by intersection. An improved intersection unit enables better performance scaling. For example, at the 8x bandwidth point, Serial-Optimal achieves a 3.9× performance improvement over its Baseline, which is a 1.78× improvement relative to ExTensor-OP-DRT with the same bandwidth.

## 6.5 Extraction Overhead, Area, and Energy

We evaluate the performance behavior of two different tile extractor implementations built into the ExTensor design. First, an ideal extractor that performs DRT in 0 cycles. Second, the parallel extractor described in Section 4.2. We observed that the performance difference between both tile extractor implementations was minimal (< 1% for each workload). This is due to the pipelining discussed in Section 4.2.3.

Finally, we use Accelergy [57], an energy/area estimation tool, to model ExTensor, ExTensor-OP, and ExTensor-OP-DRT. ExTensor-OP and ExTensor-OP-DRT both have a 0.1% area overhead compared to ExTensor. The global buffer (which is the same across designs) accounts for the lion's share of the area—99.75%. The tile extractor takes 45% of the remaining area (.25%), with the remainder going to intersection, the network on chip, and other logic. Using the geomean of energy consumption across various workloads, ExTensor-OP-DRT uses 85% and 22% less energy than ExTensor-OP and ExTensor, respectively. Figure 13 shows the area breakdown of ExTensor-OP-DRT.

## 6.6 Design Space Exploration

We explore various parameters tied to ExTensor-OP-DRT, with some results shown in Figures 14–15.

**Sweep Buffer Partitioning Allocations**. One design point is the choice of buffer allocation to tensors. Figure 14 shows the

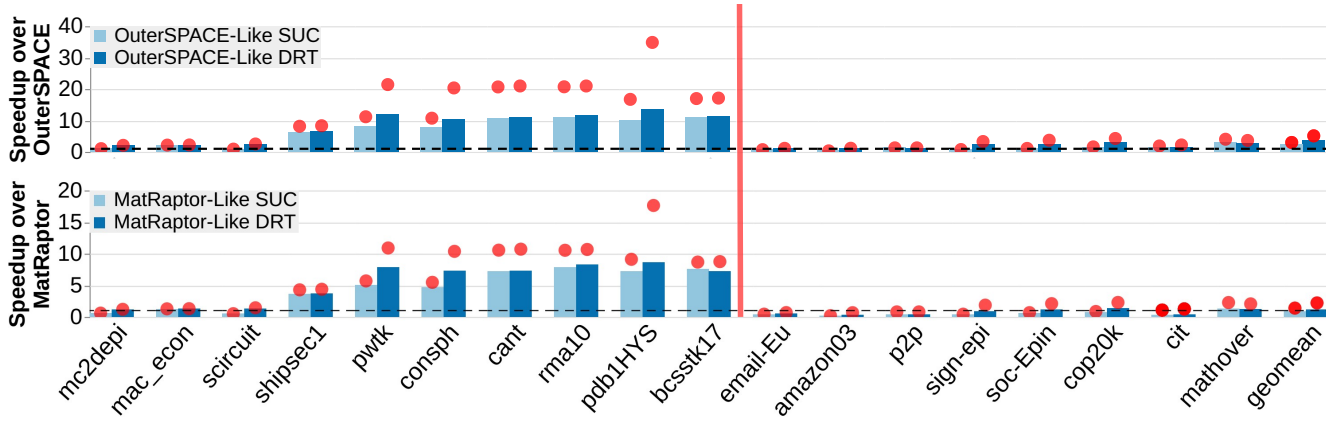



**Figure 10: (Top) OuterSPACE-SUC and OuterSPACE-DRT performance relative to the baseline OuterSPACE accelerator ($S^2$). (Bottom) MatRaptor-SUC and MatRaptor-DRT performance relative to untiled MatRaptor ($S^2$). We group workloads according to sparsity pattern, with diagonal-like patterns on the left and unstructured patterns on the right. Within each group, we order workloads by increasing input density (see Figure 6). Red dots indicate DRAM-bound performance.**

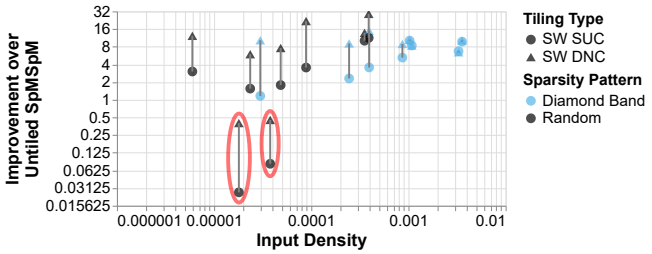



**Figure 11: Improvement in memory traffic over an untiled implementation for S-U-C tiling and DRT tiling in software ($S^2$). Red ovals indicate outliers whose tiled metadata overhead was over 8× compared to no tiling (this is not fundamental). Further discussed in Section 6.3.**

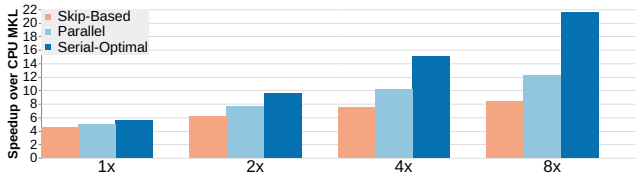



**Figure 12: Performance scaling as a function of DRAM bandwidth (GB/s).**

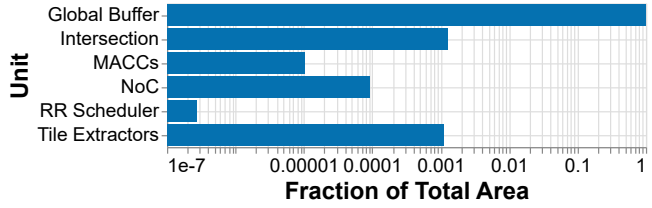



**Figure 13: Area breakdown of ExTensor-OP-DRT. Overall, ExTensor-OP-DRT adds 0.1% die area to the baseline design (ExTensor [30]).**

runtime performance for various points in the A/B/O partitioning space for the LLB. The dataflow at this level is $B$ stationary. The red line, corresponding to when $O$ receives 0% of the buffer space, indicates the capacity limit on possible partitions for tensors $A$ and $B$. Results indicate that smaller/larger allocations for $A/B$ perform better. When we zoom in on this region (small A partition), most workloads are insensitive to an increase in $B$'s partition beyond 30%. However, since the dataflow produces partial outputs, workloads with dense outputs, require enough space for $O$. Thus, static buffer allocations should consider allocating more space to the stationary tensor ($B$ in this case), while providing enough space for the output to better store partial outputs. We consider dynamic allocations for future work.

**Alternating DRT.** ExTensor-OP-DRT grows tiles by first growing along the contracted $K$ rank as much as possible, then growing along $J$, then growing along $I$. Section 3.2 (Alg 2, Line 7) notes a different growth strategy where DRT alternatively grows by one on each dimension. For the $S^2$ workload, this translates to growing once on contracted rank $J$, once on uncontracted rank $K$, and so on until the $B$ macro tile reaches its buffer capacity. DRT then creates the $A$ macro tile by growing $I$. Figure 15 shows the overhead of this growing variant, compared to the original ExTensor-OP-DRT variant. In nearly all cases, the alternating variant incurs a memory and runtime overhead, which, on further analysis, is due to an increase in output traffic. By growing along the contracted dimension first, ExTensor-OP-DRT's default variant leads to tile shapes that are longer in the contracted dimension, which in turn leads to better reuse of the output macro tile for the current task.

**Sweep starting tile size**. As noted in Section 3.2, the DRT algorithm can have an initial macro tile shape from which it grows. This affects how long or short the final tile shape will be along a particular dimension. Figure 16 shows the results of sweeping the starting tile shape.

**Sweep micro tile shape.** Figure 17 shows the impact of micro tile shape on overall memory traffic. As micro tile shape increases, DRT approaches S-U-C tiling, reducing the available opportunity in maximally filling the buffer. Smaller micro tile shapes, in the general case, incur metadata overhead, since smaller sizes lead to more micro tiles to describe. Future work will consider deciding

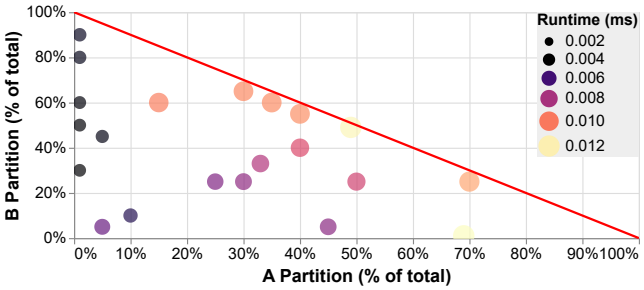



**Figure 14: Geomean performance (across the same matrices in Figure 16) as A, B, and O partitionings change. O's allocation shrinks when moving diagonally from the bottom left corner (100% allocation) to the red line (0% allocation for O).**

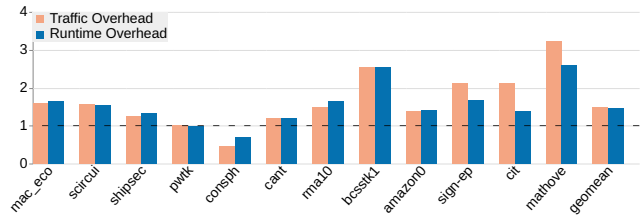



**Figure 15: Overhead of alternating DRT algorithm compared to ExTensor-OP-DRT's greedy approach. Lower is better.**

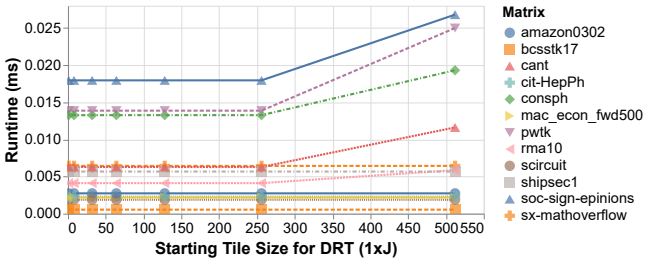



**Figure 16: Performance as the initial tile shape from which DRT grows changes. We vary along the $J$ rank since this affects the stationary $B$ matrix.**

the micro tile shape at runtime based on the sparsity pattern of an input.

**Sweep on-chip bandwidth and LLB size.** For each tested workload, the NoC bandwidth has no significant effect on performance. This is due to the overhead of main memory accesses. Additionally, when sweeping the LLB capacity, most workloads are insensitive to an increase in capacity beyond 15 MB.

## 7 RELATED WORK

Due to the irregular memory access patterns of sparse tensor kernels, traditional architectures are unable to reach peak performance. This has spurred accelerator research [4, 26, 30, 40, 42, 44, 45, 49, 50, 61] with MatRaptor, GAMMA, OuterSPACE, SpArch, and ExTensor representing state-of-the-art for SpMSpM. ExTensor and GAMMA are two recent accelerators that tile sparse data. ExTensor adopts

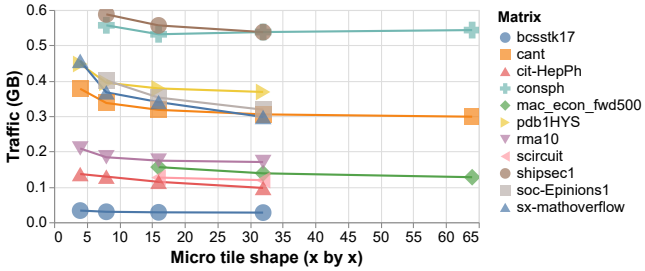



**Figure 17: Overall traffic as micro tile shape changes. We do not show runs suffering from out-of-memory issues.**

**Table 2: Sparse tiling in prior work.**

| Prior Work | Method | Kernel | Tiling |
|---|---|---|---|
| OuterSPACE [42] | HW | SpMSpM, SpMV | no explicit tiling |
| SpArch [61] | HW | SpMSpM | S-N-P |
| MatRaptor [49] | HW | SpMSpM | no explicit tiling |
| GAMMA [60] | HW | SpMSpM | D-N-C (limited) |
| ExTensor [30] | HW | SpMSpM, SpMM, TTM/V, SDDMM | S-U-C |
| ALRESCHA [4] | HW | SpMV, PCG | S-U-C |
| Near Memory SpMM [24] | SW(GPU) | SpMM | D-N-C |
| ASpT [32] | SW(CPU, GPU) | SpMM, SDDMM | S-U-P dense tiles, S-N-P sparse tiles |
| Locally Adaptive SpMV [51] | SW(GPU) | SpMV | S-U-P |
| Hierarchical 1-D Tiling [25] | SW(GPU) | SpMM/V, SDDMM | S-N-P |
| Merge-based SpMM/V [39, 59] | SW(GPU) | SpMM/V | S-U-P |
| GrateTile [37] | Storage format | CNN (SpMM, SDDMM) | S-N-C |
| J Stream [35] | SW | SpMM, SDDMM | S-U-C |
| Split Unaligned Blocks [55] | Storage format | SpMV | S-U-P |

S-U-C tiling [4, 30, 60], where sparse tiles are treated similar to dense tiles. GAMMA could be viewed as a nascent form of D-N-C tiling. It distributes rows of the $A$ matrix, not tiles, and either statically partitions $A$ or not at all in the context of Gustavson's dataflow. DRT is more general: it is dataflow-independent and tiles nonuniformly along all dimensions for any dimensional problem.

Table 2 summarizes some of the common tiling schemes used in prior work. Works are classified according to their implementation method—HW, SW, or a proposed sparse storage format—and their approach to tiling. On the software side there are more systematic works regarding sparse tensor tiling [24, 25, 25, 32, 32, 32, 39, 39, 51, 59] focusing on sparse kernels containing a single, sparse input operand. To the best of our knowledge, no such method exists for kernels with multiple sparse operands, such as SpMSpM, as the pattern of the output matrix is difficult to predict and co-tiling is non-trivial.

## 8 CONCLUSION

We present a novel dynamic reflexive tiling (DRT) mechanism, which builds nonuniform, coordinate space *macro tiles* from collections of *uniform coordinate-space micro tiles*. Our primary result is that DRT—instantiated in both hardware and software modalities—has the potential to improve performance across a range of prior accelerators and CPU implementations. Future work will mature the software effort and validate DRT's ability to effectuate a net-win performance-wise on CPUs, as it does in an accelerator setting.

## ACKNOWLEDGMENTS

The authors appreciate the financial support from a Microsoft Research Fellowship and a Facebook PhD Fellowship. This material is based upon work supported by the Defense Advanced Research Projects Agency (DARPA) under Contract No. HR0011-18-3-0007, and the National Science Foundation (NSF) under Contract No. 1909999 and Contract No. 1942888. Any opinions, findings and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of the U.S. Government.

# A APPENDICES

## A.1 SpMSpM Workloads

Table 3 lists the matrices used in this work. Abbreviated matrix names appear in the paper.

**Table 3: Sparse matrices used in the evaluation. The matrices are from HB [20], Bova [13], DNVS [18], Hamm [28], LAW [12], Williams [56], and SNAP collection [36]. Each group is separated with a divider.**

| Matrix | Dimensions | Non-zeros (Density) |
|---|---|---|
| bcsstk17 [20] | $11k \times 11k$ | 428.6k (0.356%) |
| pwtk [20] | $218k \times 218k$ | 11.5M (0.024%) |
| rma10 [13] | $47k \times 47k$ | 2.3M (0.106%) |
| shipsec1 [18] | $141k \times 141k$ | 3.6M (0.018%) |
| scircuit [28] | $171k \times 171k$ | 1M (0.0033%) |
| enron [12] | $69k \times 69k$ | 276k (0.0058%) |
| pdb1HYS [56] | $36k \times 36k$ | 4.3M (0.328%) |
| cant [56] | $63k \times 63k$ | 4M (0.103%) |
| consph [56] | $83k \times 83k$ | 6M (0.087%) |
| mac_econ_fwd500 [56] | $207k \times 207k$ | 1.3M (0.003%) |
| cop20k_A [56] | $121k \times 121k$ | 2.6M (0.018%) |
| mc2depi [56] | $526k \times 526k$ | 2.1M (0.00076%) |
| sx-mathoverflow [36] | $25k \times 25k$ | 240k (0.0389%) |
| cit-HepPh [36] | $35k \times 35k$ | 421k (0.0353%) |
| soc-Epinions1 [36] | $76k \times 76k$ | 509k (0.0088%) |
| p2p-Gnutella31 [36] | $63k \times 63k$ | 148k (0.0038%) |
| soc-sign-epinions [36] | $132k \times 132k$ | 841k (0.0048%) |
| sx-askubuntu [36] | $159k \times 159k$ | 597k (0.0024%) |
| email-EuAll [36] | $265k \times 265k$ | 420k (0.0006%) |
| amazon0302 [36] | $262k \times 262k$ | 1.2M (0.0018%) |

## REFERENCES

[1] 2023. *Dynamic Reflexive Tiling*. https://github.com/FPSG-UIUC/DRT
[2] Peter Ahrens and Erik G. Boman. 2020. On Optimal Partitioning For Sparse Matrices In Variable Block Row Format. *CoRR* abs/2005.12414 (2020). arXiv:2005.12414 https://arxiv.org/abs/2005.12414
[3] Hasan Metin Aktulga, Aydin Buluç, Samuel Williams, and Chao Yang. 2014. Optimizing Sparse Matrix-Multiple Vectors Multiplication for Nuclear Configuration Interaction Calculations. In *International Parallel and Distributed Processing Symposium (IPDPS)*. 1213–1222. https://doi.org/10.1109/IPDPS.2014.125
[4] Bahar Asgari, Ramyad Hadidi, Tushar Krishna, Hyesoon Kim, and Sudhakar Yalamanchili. 2020. ALRESCHA: A Lightweight Reconfigurable Sparse-Computation Accelerator. In *International Symposium on High Performance Computer Architecture (HPCA)*. 249–260. https://doi.org/10.1109/hpca47549.2020.00029
[5] W. Austin, G. Ballard, and T. G. Kolda. 2016. Parallel Tensor Compression for Large-Scale Scientific Data. In *International Parallel and Distributed Processing Symposium (IPDPS)*. 912–922. https://doi.org/10.1109/IPDPS.2016.67
[6] Ariful Azad, Aydin Buluc, and John Gilbert. 2015. Parallel Triangle Counting and Enumeration Using Matrix Algebra. In *International Parallel and Distributed Processing Symposium Workshop (IPDPS)*. 804–811. https://doi.org/10.1109/ipdpsw.2015.75
[7] Ariful Azad, Georgios A Pavlopoulos, Christos A Ouzounis, Nikos C Kyrpides, and Aydin Buluç. 2018. HipMCL: a high-performance parallel implementation of the Markov clustering algorithm for large-scale networks. *Nucleic Acids Research* 46, 6 (Jan. 2018), e33:1–11. https://doi.org/10.1093/nar/gkx1313
[8] Daehyeon Baek, Soojin Hwang, Taekyung Heo, Daehoon Kim, and Jaehyuk Huh. 2021. InnerSP: A Memory Efficient Sparse Matrix Multiplication Accelerator with Locality-Aware Inner Product Processing. In *30th International Conference on Parallel Architectures and Compilation Techniques (PACT 2021)*, Jaejin Lee and Albert Cohen (Eds.). IEEE, 116–128. https://doi.org/10.1109/PACT52795.2021.00016
[9] Gerald Baumgartner, Alexander A. Auer, David E. Bernholdt, Alina Bibireata, Venkatesh Choppella, Daniel Cociorva, Xiaoyang Gao, Robert J. Harrison, So Hirata, Sriram Krishnamoorthy, Sandhya Krishnan, Chi-Chung Lam, Qingda Lu, Marcel Nooijen, Russell M. Pitzer, J. Ramanujam, P. Sadayappan, and Alexander Sibiryakov. 2005. Synthesis of High-Performance Parallel Programs for a Class of ab Initio Quantum Chemistry Models. *Proc. IEEE* 93, 2 (Feb. 2005), 276–292. https://doi.org/10.1109/JPROC.2004.840311
[10] Austin R. Benson and Grey Ballard. 2015. A Framework for Practical Parallel Fast Matrix Multiplication. In *Proceedings of the 20th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP 2015)*. 42–53. https://doi.org/10.1145/2688500.2688513
[11] Maciej Besta, Raghavendra Kanakagiri, Harun Mustafa, Mikhail Karasikov, Gunnar Rätsch, Torsten Hoefler, and Edgar Solomonik. 2020. Communication-Efficient Jaccard similarity for High-Performance Distributed Genome Comparisons. In *International Parallel and Distributed Processing Symposium (IPDPS)*. 1122–1132. https://doi.org/10.1109/IPDPS47924.2020.00118
[12] Paolo Boldi and Sebastiano Vigna. 2004. The WebGraph Framework I: Compression Techniques. In *Proc. of the Thirteenth International World Wide Web Conference (WWW 2004)*. ACM Press, 595–601.
[13] Steve Bova. 1997. Model of Charleston Harbor. https://sparse.tamu.edu/bova Nichols Research Corporation.
[14] Aydın Buluç and John R. Gilbert. 2012. Parallel Sparse Matrix-Matrix Multiplication and Indexing: Implementation and Experiments. *SIAM Journal of Scientific Computing* 34, 4 (2012), 170–191. https://doi.org/10.1137/110848244
[15] Yunji Chen, Tao Luo, Shaoli Liu, Shijin Zhang, Liqiang He, Jia Wang, Ling Li, Tianshi Chen, Zhiwei Xu, Ninghui Sun, and Olivier Temam. 2014. DaDianNao: A Machine-Learning Supercomputer. In *International Symposium on Microarchitecture (MICRO)*. 609–622. https://doi.org/10.1109/MICRO.2014.58
[16] Yu-Hsin Chen, Joel Emer, and Vivienne Sze. 2016. Eyeriss: A Spatial Architecture for Energy-Efficient Dataflow for Convolutional Neural Networks. In *International Symposium on Computer Architecture (ISCA)*. 367–379. https://doi.org/10.1109/isca.2016.40
[17] 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
[18] Christian Damhaug. 1999. Positive definite matrices from Christian Damhaug, DNV Software. https://sparse.tamu.edu/DNVS
[19] Timothy A. Davis and Yifan Hu. 2011. The University of Florida Sparse Matrix Collection. *ACM Trans. Math. Software* 38, 1 (Nov. 2011), 1:1–1:25. https://doi.org/10.1145/2049662.2049663
[20] Iain S Duff, Roger G Grimes, and John G Lewis. 1989. Sparse matrix test problems. *ACM Transactions on Mathematical Software (TOMS)* 15, 1 (March 1989), 1–14.
[21] A. Einstein. 1916. The Foundation of the General Theory of Relativity. *Annalen der Physik* 354, 7 (Jan. 1916), 769–822. https://doi.org/10.1002/andp.19163540702
[22] Glen Evenbly. 2020. Tutorial 1: Tensor Contractions. https://www.tensors.net/tutorial-1
[23] Zisen Fang, Xiaowei Yang, Le Han, and Xiaolan Liu. 2019. A Sequentially Truncated Higher Order Singular Value Decomposition-Based Algorithm for Tensor Completion. *IEEE Transactions on Cybernetics* 49, 5 (May 2019), 1956–1967. https://doi.org/10.1109/TCYB.2018.2817630
[24] Daichi Fujiki, Niladrish Chatterjee, Donghyuk Lee, and Mike O'Connor. 2019. Near-Memory Data Transformation for Efficient Sparse Matrix Multi-Vector Multiplication. In *International Conference for High Performance Computing, Networking, Storage and Analysis*. ACM, Article 55, 17 pages. https://doi.org/10.1145/3295500.3356154
[25] Trevor Gale, Matei Zaharia, Cliff Young, and Erich Elsen. 2020. Sparse GPU Kernels for Deep Learning. In *International Conference for High Performance Computing, Networking, Storage and Analysis (SC)*. IEEE/ACM, Article 17, 14 pages. https://doi.org/10.1109/SC41405.2020.00021
[26] Ashish Gondimalla, Noah Chesnut, Mithuna Thottethodi, and T. N. Vijaykumar. 2019. SparTen: A Sparse Tensor Accelerator for Convolutional Neural Networks. In *International Symposium on Microarchitecture (MICRO)*. ACM, 151–165. https://doi.org/10.1145/3352460.3358291
[27] Giulia Guidi, Marquita Ellis, Daniel Rokhsar, Katherine Yelick, and Aydın Buluç. 2019. BELLA: Berkeley Efficient Long-Read to Long-Read Aligner and Overlapper. *bioRxiv* (Oct. 2019). https://doi.org/10.1101/464420
[28] Steve Hamm. 2001. Semiconductor simulation matrices from Steve Hamm, Motorola, Inc. https://sparse.tamu.edu/Hamm

[29] Kartik Hegde, Rohit Agrawal, Yulun Yao, and Christopher W. Fletcher. 2018. Morph: Flexible Acceleration for 3D CNN-based Video Understanding. In *IEEE/ACM International Symposium on Microarchitecture (MICRO '18)*. 933–946. https://doi.org/10.1109/MICRO.2018.00080

[30] 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 *International Symposium on Microarchitecture (MICRO)*. 319–333. https://doi.org/10.1145/3352460.3358275

[31] Kartik Hegde, Jiyong Yu, Rohit Agrawal, Mengjia Yan, Michael Pellauer, and Christopher W. Fletcher. 2018. UCNN: Exploiting Computational Reuse in Deep Neural Networks via Weight Repetition. In *International Symposium on Computer Architecture (ISCA)*. IEEE, 674–687. https://doi.org/10.1109/isca.2018.00062

[32] Changwan Hong, Aravind Sukumaran-Rajam, Israt Nisa, Kunal Singh, and P. Sadayappan. 2019. Adaptive Sparse Tiling for Sparse Matrix Multiplication. In *Proceedings of the 24th Symposium on Principles and Practice of Parallel Programming*. ACM, 300–314. https://doi.org/10.1145/3293883.3295712

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

[34] Fredrik Kjolstad, Shoaib Kamil, Stephen Chou, David Lugato, and Saman Amarasinghe. 2017. The Tensor Algebra Compiler. *Proceedings of the ACM on Programming Languages* 1, OOPSLA (Oct. 2017), 77:1–77:29. https://doi.org/10.1145/3133901

[35] Süreyya Emre Kurt, Aravind Sukumaran-Rajam, Fabrice Rastello, and P. Sadayyapan. 2020. Efficient Tiled Sparse Matrix Multiplication through Matrix Signatures. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*. Article 87, 14 pages. https://doi.org/10.1109/SC41405.2020.00091

[36] Jure Leskovec and Andrej Krevl. 2014. SNAP Datasets: Stanford Large Network Dataset Collection. http://snap.stanford.edu/data.

[37] Y. Lin, Hung Chang Lu, Yang-Bin Tsao, Yi-Min Chih, Weichao Chen, and S. Chien. 2020. GrateTile: Efficient Sparse Tensor Tiling for CNN Processing. In *IEEE Workshop on Signal Processing Systems (SiPS)*. 1–6. https://doi.org/10.1109/SiPS50750.2020.9195243

[38] Tim Mattson, David A. Bader, Jonathan W. Berry, Aydin Buluç, Jack J. Dongarra, Christos Faloutsos, John Feo, John R. Gilbert, Joseph Gonzalez, Bruce Hendrickson, Jeremy Kepner, Charles E. Leiserson, Andrew Lumsdaine, David A. Padua, Stephen Poole, Steven P. Reinhardt, Mike Stonebraker, Steve Wallach, and Andrew Yoo. 2013. Standards for Graph Algorithm Primitives. In *IEEE High Performance Extreme Computing Conference*. IEEE, 1–2. https://doi.org/10.1109/HPEC.2013.6670338

[39] Duane Merrill and Michael Garland. 2016. Merge-Based Sparse Matrix-Vector Multiplication (SpMV) Using the CSR Storage Format. In *Proceedings of the 21st ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*. Article 43, 2 pages. https://doi.org/10.1145/2851141.2851190

[40] Anurag Mukkara, Nathan Beckmann, and Daniel Sanchez. 2019. PHI: Architectural Support for Synchronization-and Bandwidth-Efficient Commutative Scatter Updates. In *International Symposium on Microarchitecture (MICRO)*. ACM, 1009–1022. https://doi.org/10.1145/3352460.3358254

[41] Yusuke Nagasaka, Satoshi Matsuoka, Ariful Azad, and Aydın Buluç. 2019. Performance optimization, modeling and analysis of sparse matrix-matrix products on multi-core and many-core processors. *Parallel Comput.* 90, Article 102545 (Dec. 2019), 13 pages. https://doi.org/10.1016/j.parco.2019.102545

[42] 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 *International Symposium on High Performance Computer Architecture (HPCA)*. IEEE, 724–736. https://doi.org/10.1109/hpca.2018.00067

[43] 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 *International Conference on Architectural Support for Programming Languages and Operating Systems*. ACM, 137–151. https://doi.org/10.1145/3297858.3304025

[44] Eric Qin, Ananda Samajdar, Hyoukjun Kwon, Vineet Nadella, Sudarshan Srinivasan, Dipankar Das, Bharat Kaul, and Tushar Krishna. 2020. SIGMA: A Sparse and Irregular GEMM Accelerator with Flexible Interconnects for DNN Training. In *International Symposium on High Performance Computer Architecture (HPCA)*. IEEE, 58–70. https://doi.org/10.1109/hpca47549.2020.00015

[45] Fazle Sadi, Joe Sweeney, Tze Meng Low, James C. Hoe, Larry Pileggi, and Franz Franchetti. 2019. Efficient SpMV Operation for Large and Highly Sparse Matrices using Scalable Multi-way Merge Parallelization. In *International Symposium on Microarchitecture (MICRO)*. ACM, 347–358. https://doi.org/10.1145/3352460.3358330

[46] 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/

[47] Edgar Solomonik, Maciej Besta, Flavio Vella, and Torsten Hoefler. 2017. Scaling Betweenness Centrality using Communication-Efficient Sparse Matrix Multiplication. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*. Article 47, 14 pages. https://doi.org/10.1145/3126908.3126971

[48] Edgar Solomonik and Torsten Hoefler. 2015. Sparse Tensor Algebra as a Parallel Programming Model. *CoRR* abs/1512.00066 (2015). arXiv:1512.00066 http://arxiv.org/abs/1512.00066

[49] 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 *International Symposium on Microarchitecture (MICRO)*. 766–780. https://doi.org/10.1109/MICRO50266.2020.00068

[50] 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 *International Symposium on High Performance Computer Architecture (HPCA)*. IEEE, 689–702. https://doi.org/10.1109/hpca47549.2020.00062

[51] Markus Steinberger, Rhaleb Zayer, and Hans-Peter Seidel. 2017. Globally Homogeneous, Locally Adaptive Sparse Matrix-Vector Multiplication on the GPU. In *Proceedings of the International Conference on Supercomputing*. Article 13, 11 pages. https://doi.org/10.1145/3079079.3079086

[52] Michelle Mills Strout, Larry Carter, Jeanne Ferrante, and Barbara Kreaseck. 2004. Sparse Tiling for Stationary Iterative Methods. *Int. J. High Perform. Comput. Appl.* 18, 1 (2004), 95–113. https://doi.org/10.1177/1094342004041294

[53] Vivienne Sze, Yu-Hsin Chen, Tien-Ju Yang, and Joel S. Emer. 2017. Efficient Processing of Deep Neural Networks: A Tutorial and Survey. *Proc. IEEE* 105, 12 (Dec. 2017), 2295–2329. https://doi.org/10.1109/JPROC.2017.2761740

[54] Stijn Marinus van Dongen. 2000. *Graph Clustering by Flow Simulation*. Ph. D. Dissertation. Center for Math and Computer Science (CWI), Utrecht University.

[55] Richard W. Vuduc and Hyun-Jin Moon. 2005. Fast Sparse Matrix-Vector Multiplication by Exploiting Variable Block Structure. In *High Performance Computing and Communications (HPCC)*, Vol. 3726. 807–816. https://doi.org/10.1007/11557654_91

[56] Samuel Williams, Leonid Oliker, Richard Vuduc, John Shalf, Katherine Yelick, and James Demmel. 2009. Optimization of Sparse Matrix-Vector Multiplication on Emerging Multicore Platforms. *Parallel Comput.* 35, 3 (March 2009), 178–194. https://doi.org/10.1016/j.parco.2008.12.006

[57] Yannan Nellie Wu, Joel S. Emer, and Vivienne Sze. 2019. Accelergy: An Architecture-Level Energy Estimation Methodology for Accelerator Designs. In *Proceedings of the International Conference on Computer-Aided Design (ICCAD 2019)*, David Z. Pan (Ed.). ACM, 1–8. https://doi.org/10.1109/ICCAD45719.2019.8942149

[58] Abdurrahman Yaşar, Muhammed Fatih Balin, Xiaojing An, Kaan Sancak, and Ümit V. Çatalyürek. 2022. On Symmetric Rectilinear Partitioning. *ACM Journal of Experimental Algorithmics* 27 (Aug. 2022), 1–26. https://doi.org/10.1145/3523750

[59] Carl Yang, Aydın Buluç, and John D. Owens. 2018. Design Principles for Sparse Matrix Multiplication on the GPU. In *Euro-Par 2018: Proceedings of the 24th International European Conference on Parallel and Distributed Computing*, Marco Aldinucci, Luca Padovani, and Massimo Torquati (Eds.). 672–687. https://doi.org/10.1007/978-3-319-96983-1_48

[60] 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 (ASPLOS 2021)*. 687–701. https://doi.org/10.1145/3445814.3446702

[61] Zhekai Zhang, Hanrui Wang, Song Han, and William J. Dally. 2020. SpArch: Efficient Architecture for Sparse Matrix Multiplication. In *International Symposium on High Performance Computer Architecture (HPCA)*. IEEE, 261–274. https://doi.org/10.1109/HPCA47549.2020.00030


Received 2022-10-20; accepted 2023-01-19