ExTensor: An Accelerator for Sparse Tensor Algebra

Kartik Hegde
University of Illinois at Urbana-Champaign
kvhegde2@illinois.edu

Hadi Asghari-Moghaddam
University of Illinois at Urbana-Champaign
asghari2@illinois.edu

Michael Pellauer
NVIDIA
mpellauer@nvidia.com

Neal Crago
NVIDIA
ncrago@nvidia.com

Aamer Jaleel
NVIDIA
ajaleel@nvidia.com

Edgar Solomonik
University of Illinois at Urbana-Champaign
solomon2@illinois.edu

Joel Emer
NVIDIA/MIT
emer@csail.mit.edu

Christopher W. Fletcher
University of Illinois at Urbana-Champaign
cwfletch@illinois.edu

ABSTRACT

Generalized tensor algebra is a prime candidate for acceleration via customized ASICs. Modern tensors feature a wide range of data sparsity, with the density of non-zero elements ranging from 10^{-6}% to 50%. This paper proposes a novel approach to accelerate tensor kernels based on the principle of *hierarchical elimination of computation in the presence of sparsity*. This approach relies on rapidly finding *intersections*—situations where both operands of a multiplication are non-zero—enabling new data fetching mechanisms and avoiding memory latency overheads associated with sparse kernels implemented in software.

We propose the *ExTensor* accelerator, which builds these novel ideas on handling sparsity into hardware to enable better bandwidth utilization and compute throughput. We evaluate ExTensor on several kernels relative to industry libraries (Intel MKL) and state-of-the-art tensor algebra compilers (TACO). When bandwidth normalized, we demonstrate an average speedup of 3.4×, 1.3×, 2.8×, 24.9×, and 2.7× on SpMSpM, SpMM, TTV, TTM, and SDDMM kernels respectively over a server class CPU.

CCS CONCEPTS

• **Computer systems organization → Special purpose systems**.

KEYWORDS

Tensor Algebra, Sparse Computation, Hardware Acceleration

ACM Reference Format:
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 *The 52nd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO-52), October 12–16, 2019, Columbus, OH, USA.* ACM, New York, NY, USA, 15 pages. https://doi.org/10.1145/3352460.3358275

MICRO-52, October 12–16, 2019, Columbus, OH, USA

ACM ISBN 978-1-4503-6938-1/19/10…$15.00
https://doi.org/10.1145/3352460.3358275

1 INTRODUCTION

Recently, there has been a surge of interest in generalized tensor algebra in the fields of deep learning [1, 15], machine learning [5], data science [7, 9, 27, 35, 48], physical sciences [18], engineering [20, 28], and graph analytics [34]. Tensors generalize vectors and matrices to N-dimensions, and tensor kernels combine two or more tensors using low-level computations similar to traditional 2-dimensional dense/sparse linear algebra, i.e., sequences of arithmetically intensive operations such as matrix multiplications.

These tensor operations often operate on very sparse data (i.e., with a small percentage of data which is non-zero). Figure 1 shows that the percentage of non-zero elements ranges from 10^{-6}% to 50% depending on the problem domain, with many domains featuring tensors with density less than 0.1%. As a result, tensors are stored in compressed representations and looked-up via *metadata*, e.g., compressed sparse row (CSR) or compressed sparse fiber (CSF) [46], which indicate where non-zeros occur in the tensor.

The variety of tensor kernels, their extreme sparsity, and their compressed representations make tensor algebra challenging on today's platforms. On one hand, the architecture community has proposed accelerators for sparse linear algebra, but they deal with relatively dense data (e.g., deep neural networks [4, 13, 21, 42, 51]) or specific kernels (e.g., sparse matrix vector multiply [2, 8, 24, 33, 36, 40, 41]). On the other hand, general purpose platforms such as CPUs and GPUs cannot reach peak performance for a variety of reasons, e.g., main memory latency [17, 38].

This paper proposes a novel approach for accelerating generalized tensor algebra that is efficient when handling very sparse tensors. At a basic level, the main opportunity provided by sparsity in tensor operations is the potential to exploit the axiom $0 \cdot x = 0$ for any x. For a scalar x, various platforms have exploited this opportunity to remove ineffectual computation. For example, we can avoid delivering x to the staging buffers and performing a multiply [4, 12, 13, 21, 26, 31, 40, 42, 50, 51]. Our key insight is that in

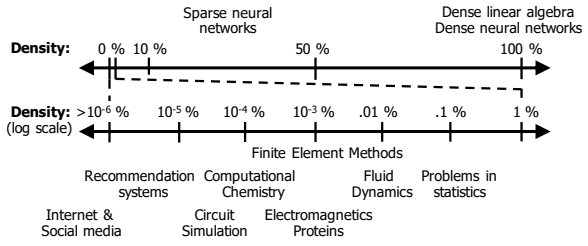

Figure 1: Tensor sparsity by workload domain.

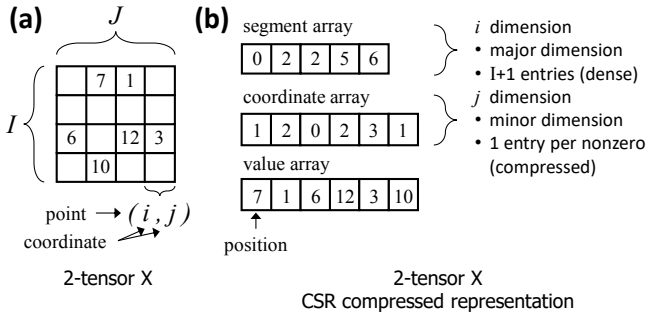

Figure 2: Tensor terminology & example compression using CSR.

higher-order tensor algebra, this opportunity applies *even when x is not a scalar*. For example, x might be a tile, in which case recognizing that the other operand is 0 means we don't have to transfer data (or metadata) for the entire tile. x might even represent an un-evaluated tensor computation, and recognizing that the other operand is 0 early can result in skipping all operations, data, and metadata transfers required to create x. These opportunities are even further magnified when the *0* corresponds to a tile or larger region of the tensor.

Mathematically, to detect the work that will not be skipped, we can calculate the *intersection* of coordinates—logical locations—of non-zero elements in each tensor operand. Importantly, coordinates can be associated with scalars, tiles, sub-computations, etc., allowing us to hierarchically eliminate work using these intersections.

With this in mind, we propose ExTensor[1], an accelerator architecture built around the idea of performing hierarchical compositions of intersections to eliminate ineffectual computation. In ExTensor, tensor metadata and data processing are decoupled so that the metadata engines can aggressively look ahead in the computation to discover ineffectual computation before they are delivered to the arithmetic units. This idea is applied to each level in the buffer hierarchy to stage sub-tiles and sub-computations at the next level so that, at the bottom level, the arithmetic units perform a minimal amount of work.

Overall, we make the following contributions.

(1) To the best of our knowledge, we propose the first accelerator for general, sparse tensor algebra.
(2) We describe a general abstraction—based on intersections between coordinates of non-zero data—for describing intersections (opportunities to skip work due to sparsity) in sparse tensor algebra, that applies at different granularities in the problem (e.g., scalar level, tile level).
(3) We propose a hardware mechanism, and optimizations, for performing these intersections at multiple levels of an accelerator's memory hierarchy.
(4) We extensively evaluate ExTensor over a range of tensor kernels, and implement a core hardware component in RTL. When bandwidth normalized, we demonstrate an average (geomean) speedup of 3.4×, 1.3×, 2.8×, 24.9×, and 2.7× on SpMSpM, SpMM, TTV, TTM, and SDDMM kernels respectively over a server class CPU.

[1]In biology, an *extensor* is a muscle whose contraction causes a limb to straighten, which we see as analogous to performing intersections that unravel a sparse tensor computation into a linear stream of the desired operations.

2 BACKGROUND

We review the key details of tensor algebra that are relevant to computer architecture and our proposed approach. For additional information see [27]. When possible, we use terminology and mathematical notation from the TACO tensor algebra compiler [26].

2.1 Tensor Terminology and Representation

Tensors are multi-dimensional arrays of arbitrary *order* (dimensionality) N, and 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 tensor $X \in \mathbb{R}^{I \times J}$, i.e., a 2-tensor (matrix) of real numbers (floats) of size $I \times J$. Locations in a tensor are referred to as *points*, which are identified by a tuple of *coordinates*, one for each dimension. Our notation for the coordinates of a point uses concatenated subscripts. For example, the element of tensor X at point (i, j) for $i < I, j < J$ is written as X_{ij}. In cases where concatenation introduces ambiguity we clarify with parentheses, e.g., the element at $(i + 1, j)$ is $X_{(i+1)j}$.

Large tensors are frequently sparse (see Figure 1 and [26]), so compressed representations are used to avoid storing zero data values in memory. Various domain-specific compressed representations exist to minimize storage while also enabling high-performance access to the non-zero data. Generally, all compressed formats store *metadata* for indexing the compressed data structure as well as the actual *data* (non-zero tensor values), each element of which is stored at a *position* in memory.

For dense (uncompressed) tensors, there is an $O(1)$-cost translation from coordinate to data position, which permits efficient random access. In compressed representations, random access can require a search through a list of coordinates, but sequential traversal of the tensor can often leverage the metadata format to improve efficiency. For example, Figure 2 (b) shows a tensor represented in the Compressed Sparse Row (CSR) format. Note that access to a completely random point (i, j) requires a memory indirection in order to find the row bounding positions (via the segment array), followed by a search to locate the appropriate inner dimension coordinate, if present (through the coordinate array). However, also note that once this search is performed, the position of the next non-zero in row i (if any) is derivable from the position found for (i, j), meaning that an operation acting on the subsequent non-zero in the row may be performed in $O(1)$ time.

In this paper, we support tensor representations where each dimension is stored either dense/uncompressed (U) or compressed

Table 1: Example tensor kernels and BLAS routines. α, β, γ are scalars (0-tensors); A, B, C, Z, O, I, F are higher-order tensors. Symbols for convolution match [12] where possible for consistency. We note that $\sum$ has precedence, meaning GEMV should be read as $\alpha(\sum_k A_k B_{ki}) + \beta C_i$.

Name	Tensor index notation
GEMV	$Z_i = \alpha \sum_k A_k B_{ki} + \beta C_i$
GEMM	$Z_{ij} = \alpha \sum_k A_{ik} B_{kj} + \beta C_{ij}$
TTV	$Z_{ij} = \sum_k A_{ijk} B_k$
TTM	$Z_{ijk} = \sum_l A_{ijl} B_{kl}$
SDDMM	$Z_{ij} = C_{ij} \sum_k A_{ik} B_{kj}$
MTTKRP	$Z_{ij} = \sum_{kl} A_{ikl} B_{kj} C_{lj}$
2D Conv	$O_{xy} = \sum_{rs} I_{(x+r)(y+s)} F_{rs}$
CNN layer	$O_{zuxy} = \sum_{crs} I_{zc(\gamma x+r)(\gamma y+s)} F_{ucrs}$

(C) [26]. We refer to specific formats in this family using notation defined by the following regular expression: "T-[UC]$^+$". In this taxonomy, a 2-tensor (matrix) in the CSR format is categorized as T-UC, corresponding to the first dimension being uncompressed and second dimension compressed. The Compressed Sparse Column (CSC) format is also categorized as T-UC but with rows as the first dimension and columns as the second dimension [14]. Many other representations are also possible. For example, the Compressed Sparse Fiber (CSF) [46] format corresponds to class of representations described by T-C$^+$.

Given an N-dimensional tensor, choosing whether each dimension should be uncompressed or compressed is a trade-off between random access time and storage. In terms of random access, translating a tuple of coordinates to a data position is done dimension-by-dimension, where each uncompressed dimension is traversed in $O(1)$ time and each compressed dimension requires a search. In terms of storage, an uncompressed dimension must allocate metadata proportional to the size of the dimension whereas a compressed dimension need only allocate storage proportional to the number of non-zero elements in the dimension. Importantly, representing a dimension in compressed form gives the opportunity to avoid storing any data *and* metadata for lower dimensions. For example, a 2-tensor compressed as T-CC stores no metadata/data for a row that contains only zeros.

2.2 Tensor Algebra Kernels

Tensor algebra is the process of performing binary operations (e.g., multiplies and adds forming dot products) between tensors to produce new tensors. We follow the tensor index notation found in [26], which provides a compact way to describe a kernel's functionality. For example, matrix multiply $Z = AB$ can be written as:

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

That is, the output point (i, j) is formed by taking a dot product of k values along the i-th row of A and the j-th column of B.

We give more examples of important kernels written in this form in Table 1. Each kernel is useful in different problem domains. GEMV/GEMM are well known and have many applications [10, 34]. The SDDMM kernel is used in machine learning [52] and triangle counting [6]. MTTKRP is used in tensor decompositions [27], 2D Conv is used in image processing applications, and CNN layers are the core kernels in state-of-the-art image recognition [22, 29].

Tiled Tensor Algebra: Sparse tensors can be tiled to improve locality, by adding dimensions to the tensor representation. For example, to break 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 which can be either U or C. Thus, each tile is represented in CSR, and we traverse outer dimensions to index into each tile. Importantly, compressed outer dimensions imply no storage for tiles that contain only zeros. Supporting multiple levels of tiles entails adding yet more dimensions. As the most efficient tiling is dependent on the traversal order of any given kernel, offline pre-processing is considered acceptable in the tensor community [26], as it can be thought of selecting the most appropriate compression format for the data.

3 INTERSECTION OPPORTUNITIES

From Table 1, we see that tensor kernels are a composition of computations on values generated by *co-iterations* through (possibly different) dimensions of two or more tensors. For example, in Equation 1 (matrix multiply) we perform co-iterations through the k dimension, which corresponds to traversing corresponding elements from rows of matrix A and columns of matrix B. The co-iterations *merge* the elements from the source tensors using either multiply or add, which is sometimes additionally reduced via accumulate (e.g., via dot product, min/max). An important observation is that, because $0 \cdot x = 0$, a co-iteration with multiplication is a merge intersection operation (intersection for short) between two sets of coordinates corresponding to non-zero values in each operand. Such intersections can enable skipping computation, data transfers, or both. We now describe different intersection opportunities for the matrix multiply example (Equation 1), using the matrices in Figure 3. The key takeaway is that, in all cases, *intersection can be modeled as comparisons between sets of coordinates.* This will allow us to build a single hardware mechanism to perform the different types of intersections we illustrate below.

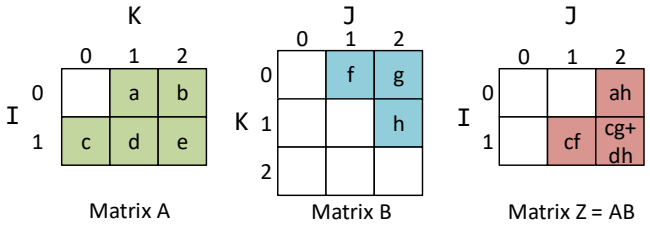

Figure 3: Example matrices. White space indicates zero value. Numbers along each dimension are coordinates for that dimension.

3.1 Two-Operand Intersections

For the following examples, we visualize tensors as N-level trees, where each level indicates the coordinates for one dimension, and each unique path through the tree represents a non-zero data element and its associated coordinates. (See Figure 4 for examples of the matrices in Figure 3.) Dimensions are labeled level by level, with the outermost immediately below the root of the tree. A subtree rooted at a coordinate represents a subtensor. When a subtensor

contains only zero data elements, we omit its subtree. The order of dimensions (levels) in the tree corresponds to different data layouts, and is set offline based on the intended traversal order. For example, different dataflows [12] such as output- and A-stationary (Figure 4, left and right) prefer matrix B to be stored in different layouts.

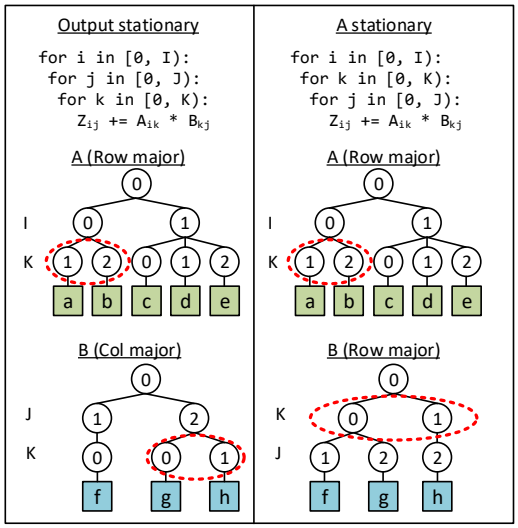

Figure 4: Example dataflows for matrix multiply. Intersections are between sets of coordinates for each dataflow (red dashed ovals).

Importantly, the tree representation conveys the same information as T-[UC]$^+$ representations such as CSR and CSF (Section 2.1), while hiding implementation details. Furthermore, in T-[UC]$^+$ formats it is generally the case that coordinates within a level are stored sequentially in memory, which means we have good spatial locality when traversing a level.

Intersections Within Dot Products: Suppose the matrix multiply kernel is implemented using an output-stationary dataflow, where each output Z_{ij} is computed one at a time, shown in Figure 4 (left). In output stationary, we only need to do a multiply when corresponding values in the active row and column are both non-zero. These operations can be viewed as intersections of sets of coordinates between rows of A and columns of B.

By laying out A and B in row- and column-major order, respectively, this corresponds to intersecting coordinates in the second level (dimension K) of each matrices' tree representation. The coordinates involved in the intersection to compute Z_{02}, a dot product, are shown in the figure as dashed red ovals. Because only coordinate 1 appears in both ovals, only data elements a from matrix A and h from matrix B need to be read and computed upon. Because the subtrees under each intersected coordinate are scalars, this intersection saves work (both loads of values and computes) associated with scalar operations.

Intersections Across Dot Products: We can also intersect sets of coordinates in levels (tensor dimensions) closer to the root of the tree and between different levels of the tree, which eliminates work associated with entire subtensors with a single intersection. For example, consider the A-stationary dataflow in Figure 4 (right), where Matrix B is laid out in row-major order. Intersections are now between sets of coordinates from the second level (dimension K) of A and the first level (dimension K) of B. As we see in the example, the missing 0 coordinate in the second dimension of A eliminates a number of multiplies that would have been done with elements from the subtree with coordinate 0 in the first dimension of B (broadcasting the scalar across the row of B). This can eliminate work fetching the data and metadata from the row of B as well as the multiplies themselves. Also, the missing coordinate 2 in the first dimension of B (a zero row) eliminates work associated with the scalar b in matrix A.

Intersections at Tile Granularity: Recall from Section 2.2, tiling can be implemented by adding additional dimensions to the compressed representation. Thus, the opportunities above apply when scalars a, b, c, etc. in matrices A and B are replaced with tiles (subtrees) of A and B, meaning we can perform intersections at the tile granularity to eliminate the work done to load/compute on tiles, when entire tiles contain only zero. That is, intersection opportunities are hierarchical.

3.2 Compositions of Intersections

More intersection opportunities arise in kernels involving more than two tensors. For example, the SDDMM kernel performs element-wise multiplications between matrix C and the results of a matrix multiplication between A and B (Table 1). This can be viewed as a composition of intersections, where we compute intermediate matrix $D = AB$ and the coordinates of C and D are subsequently intersected. Suppose we intersect C and D *before* (or while) D is being computed. Since missing coordinates in A are known at the start, this early intersect tells us when a dot product in $AB = D$ is ineffectual and can be skipped. Likewise, if dot products in AB result in zero, which occurs if there is an empty intersection between corresponding rows and columns in A and B, we can eliminate work in fetching/computing on data in C.

4 INTERSECTING STREAMS

From the previous section, we saw multiple opportunities where intersection could save various types of work (arithmetic, data and metadata transfers, whole sub-computations, etc). A key observation is that these various intersection opportunities require the same computation: namely an intersection operation between sets of coordinates in the dashed red ovals. In hardware, we call the contents of these ovals *streams* of coordinates. In this and the next section, we design hardware to efficiently intersect these streams.

4.1 Iterating over Streams

A tensor kernel is made up of many streams which are hierarchically intersected as described in Section 3. For simplicity, we start by describing the simplest intersection: between coordinates of two streams. Before we consider intersection, however, we describe the unit that stores individual coordinate streams. It consists of metadata storage, that interfaces with a hardware FSM that iterates through the storage. We call the combined unit a *Scanner*, shown in Figure 5.

Scanners must perform one operation: ITERATE(), which outputs coordinates stored in the metadata storage, in increasing order by coordinate. (We denote explicit command interfaces supported in

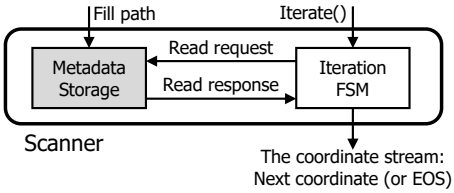

Figure 5: Scanner hardware. Storage is shaded.

the hardware with method call syntax.) For now, we assume each ITERATE() call outputs all stored coordinates. Once the iteration is complete, the Scanner outputs the symbol EOS to indicate the end of the stream.

The metadata storage can be implemented in multiple ways, e.g., as cache, (double buffered) scratchpad, or buffet [43]. Depending on the implementation, a separate mechanism may be required to *fill* the coordinates into the metadata storage. For example, a cache-based or scratchpad-based storage fills through demand loads from the iteration FSM or an explicit fetch unit, respectively. To simplify the ITERATE() command's implementation, coordinates are filled and stored in increasing order.

4.2 Intersecting Streams

Intersecting two streams requires co-iterating over two Scanners in parallel and processing the resulting streams using a hardware block called *Intersect*, as shown in Figure 6. The design generalizes to intersecting any number of parallel streams, but we discuss two for simplicity.

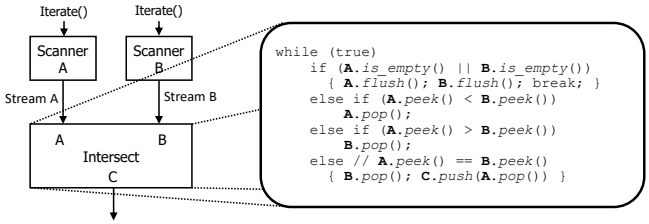

Figure 6: Basic intersection hardware and algorithm.

We use FIFO terminology: the current first/last element in a stream as seen from a consumer is denoted head/tail, respectively. In each clock tick, the head coordinate of each stream A and B, generated by Scanner A and B, respectively, is checked by value (via peek). If the head coordinates of both streams match, those coordinates are both effectual, and the common coordinate is passed to the output C. Otherwise, if the head coordinate of one stream is less than the head of the other, the head of the lagging stream is ineffectual and is dropped (via pop). In this case, no output is sent to C in that cycle. This is functionally correct because the coordinates in each stream are ordered. Finally, if one stream is empty, the intersection exits (via `flush`) as no more effectual coordinates will be produced.

We note that it is straight-forward to employ a vector pipeline for Intersect by having each scanner read out vectors of coordinates. We assume each Scanner produces a single coordinate per cycle for simplicity.

4.3 Efficiency

In hardware, the Scanner is a simple pipelined iterator that produces one coordinate per cycle. Likewise, the Intersect block can be pipelined to perform one iteration of the while loop per cycle. Thus, the above design completes an intersection in $O(|\text{Stream}_A \cup \text{Stream}_B|)$ cycles.

From this, we see that our intersection hardware has a performance deficiency. Ideally, we want Intersect to take no more than $|\text{Stream}_A \cap \text{Stream}_B|$ cycles, as this is the minimum amount of time needed to output the common coordinates. Yet, even when $|\text{Stream}_A \cap \text{Stream}_B|$ is small, intersection time is still proportional to the sum of the length of both streams in the worst case. Consider two example streams in Figure 7 (a) and (b). Streams are given on the left, and timing of Intersect for each stream on the right.

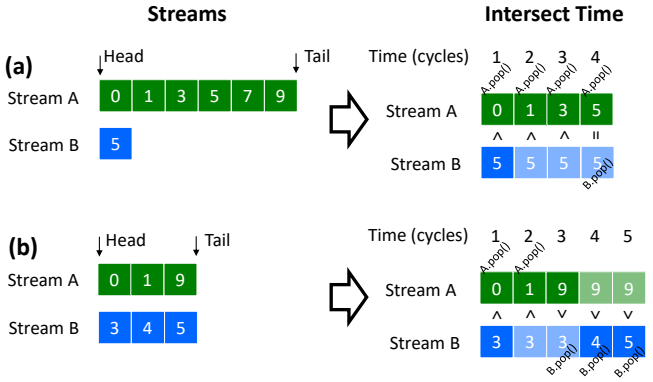

Figure 7: Examples for the basic intersect architecture, with cycle-level timing. Coordinates for each stream are shown in boxes. Lightened boxes indicate the value has not changed from the previous cycle (i.e., was not popped in the previous cycle).

In Figure 7 (a), Stream B has fewer coordinates than Stream A, but the head of Stream B will not be dropped (which would terminate the intersection early) until Stream A has been iterated partially (or completely), because Stream B's coordinate is greater than some (or all) of Stream A's coordinates. This scenario occurs frequently when intersecting data from two tensors of highly different density, which occurs in important applications such as breadth first search [34]. In Figure 7 (b), the two streams result in an empty intersection. One would like to detect this case and terminate intersection immediately, outputting the empty set.

5 OPTIMIZED INTERSECTION

In this section, we propose an optimized intersection architecture to remove the bottlenecks discussed in the previous section (Section 4.3). Our design is based on two key observations, which we illustrate using Figure 7.

First, in Figure 7 (a), the only work necessary to complete the intersection is to determine if Stream A contains coordinate 5. More generally, intersection requires *searching* for coordinates in one stream (e.g., Stream A) which match those in the other stream (e.g., Stream B). Semantically, this is a content addressable lookup which the strawman design implements in linear time (i.e., by iterating

through coordinates 0, 1, 3, 5). Thus, to speedup intersections the core task is to architect a faster content addressable search.

Second, since coordinates in streams are ordered, consecutive coordinates in one stream can be interpreted as *a range of coordinates that will be ineffectual in the opposite stream*. For example, in Figure 7 (b), the consecutive coordinates 1, 9 in Stream A can be interpreted as a hint to Scanner B that all coordinates in the open interval (1, 9) – namely coordinates 3, 4, 5 – are ineffectual and do not have to be output. Such a scheme can "rule out" a large number of coordinates in a single shot. Importantly, to efficiently skip through ranges of coordinates within a stream, we require a similar content addressable lookup, as in the previous paragraph, to identify the start and end of each range.

5.1 Skip Mechanism Design

Combining the above observations, we add a new functionality to scanners called SkipRange() which takes a (*begin*, *end*) tuple of coordinates as input. Scanners send each other SkipRange() commands in a bidirectional fashion to decrease the number of coordinates sent to the Intersect block and more quickly reach the end of their respective streams.

Suppose a scanner is in the middle of an Iterate() operation (i.e., has output some but not all of its stream). If that scanner receives a SkipRange() command, it is allowed to skip over (i.e., not output) coordinates in the range (*begin*, *end*). Depending on the value of coordinate *coord* at the head of the scanner's stream, the scanner must handle three cases:

(1) *coord* ≤ *begin*: Output *coord* as usual. Do not consume the SkipRange() command this cycle.
(2) *begin* < *coord* < *end*: Do not output *coord*. Do not consume the SkipRange() command this cycle.
(3) *end* ≤ *coord*: Output *coord* as usual. Consume the SkipRange() command.

SkipRange() may have no effect, if the scanner's head coordinate has already past *end* in the range (Case 3). Once the scanner enters Case 2, it may require multiple cycles to transition to Case 3 depending on the number of coordinates in the range. In Section 5.2, we architect an efficient content addressable lookup to jump immediately to Case 3 once we have reached Case 2.

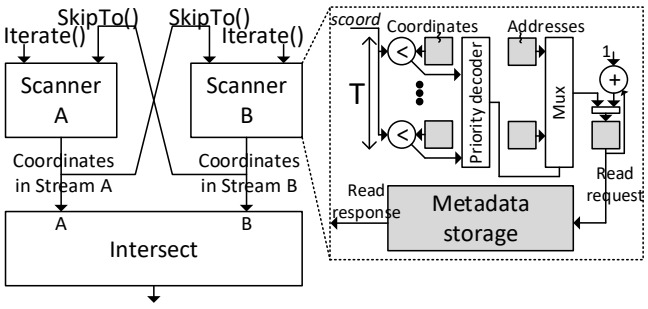

Figure 8: (Left) Intersection based on bidirectionally skipping ranges of coordinates. (Right) The SkipTo() implementation, with input coordinate *scoord* and CAM capacity T (Section 5.2). Double buffering the CAM registers is not shown. Storage is shaded.

The overall design is shown in Figure 8. Iterate() is called on both scanners. Conceptually, each scanner initiates SkipRange() operations on its neighbor, for each pair of consecutive coordinates in its stream. SkipRange() internally performs the content addressable lookup to advance to the end of the range (Section 5.2). Notice that consecutive (*begin*, *end*) tuples share one coordinate. For example, Stream A in Figure 7 (b) might trigger SkipRange() calls with coordinate ranges (0, 1) and (1, 9) – notice the '1' is common. Thus, as an optimization we implement SkipRange() to take a single coordinate – the next coordinate in the stream denoted *scoord* – and call this variant SkipTo() for clarity. Importantly, for synchronization correctness, the scanner must not consume a SkipTo() command and its coordinate until the head of the stream is greater than or equal to the coordinate argument to the previous SkipTo() call.

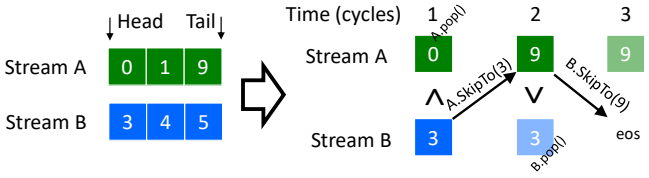

Figure 9: Example timing using the optimized intersect architecture. For simplicity, the examples assumes SkipTo() skips through any range of coordinates in 1 cycle.

Crucially, SkipTo() operations are only triggered by coordinates successfully *output* by each scanner. We show a timing example in Figure 9, which corresponds to Figure 7 (b), assuming we can implement the content addressable lookup in a single cycle. Scanner A need not initiate a SkipTo() operation on Scanner B for coordinate 1, since Scanner B previously sent a SkipTo() command to Scanner A which skipped over coordinate 1 (and moved the iteration for Scanner A to coordinate 9).

5.2 Content Addressable Lookup

Given the coordinate input to SkipTo(), *scoord*, we wish to find the address in the metadata storage corresponding to the largest coordinate *tcoord* such that *tcoord* < *scoord*. The ability to quickly find *tcoord* when the head coordinate *coord* is in (*begin*, *end*) is critical to the effectiveness of SkipTo().

There are different possible implementations for this search with different time/space trade-offs, e.g., a binary search or a hash table. For simplicity, we implement the search using a hardware CAM with T parallel comparators, shown in Figure 8 (right), which we call the coarse-grain CAM. Each comparator tracks a coordinate and address (for that coordinate, in the metadata storage) in the stream. Given a stream with S coordinates: if $S \leq T$, every coordinate is assigned a comparator; else the stream is partitioned into $T + 1$ regions of as close to equal size as possible. Note, the number of coordinates in each stream is pre-computed in the T-[UC]$^+$ representation. We assume the spacing between coordinates assigned to consecutive comparators—i.e., $\lfloor S/T \rfloor$—is also calculated offline and stored in stream metadata.

To perform a SkipTo() command, the Scanner advances the head of its stream to the largest coordinate associated with a comparator whose value is less than *scoord*. Since the comparators are parallel,

this lookup takes a single cycle. T is a design-time parameter that controls SkipTo() effectiveness: smaller T yields a coarser-grain look-up (potentially skipping less), while larger T increases design area.

Lastly, we note that the T coordinates must be loaded into registers connected to the comparators at the beginning of the Iterate() calls. Although we have discussed a single stream, Scanners process sequences of streams back to back. To hide comparator load time, we double buffer the registers associated with the comparators, and use a second read port into metadata storage to fill the registers while an earlier stream is being intersected.

6 STAGING INTERSECTIONS

So far, we have discussed how to intersect a single pair of coordinate streams. To evaluate a tensor kernel, we must intersect multiple pairs of streams, in a sequence, and stage the results of those intersections for use in subsequent intersections or arithmetic operations. We perform these tasks using a hardware unit called the Stream Coordinator (Coordinator for short). Shown in Figure 10, the Coordinator is made up of two Scanners (Sections 4.1, 5.1), an Intersect unit (Section 4.2), an FSM that schedules the order in which streams are produced by each Scanner (Section 6.1), and memories to store tensor data (Section 6.2). This logic corresponds to one or more levels in the loop nest representation of a tensor kernel.

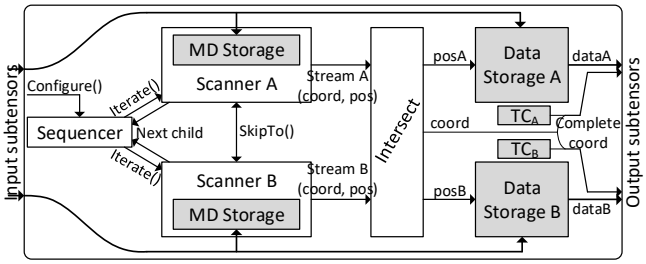

Figure 10: The Coordinator. Next child indicates the next non-empty child for Iterate() (Section 6.1). MD stands for metadata. TC_A and TC_B append tile-level coordinates to each output so that the point (Section 2.1) of any data value output can be derived later on.

Throughout the section, we use Figure 4, output stationary, as a running example.

6.1 Sequencing & Sourcing Streams

Up to this point, we have assumed a single stream has been stored in each scanner's metadata storage. We now generalize this to a multi-dimensional tile of coordinates which corresponds to a subtensor. Each tile is stored in a T-[UC]$^+$ representation (Section 2.1) and is conceptually a tree or subtree from Section 3.

We add two arguments to the Scanner's Iterate() functionality: *level* and *parent_node*, which represent a level (dimension) of the tree and a node (coordinate) whose child coordinates should form the stream.[2] Level 0 is reserved for the root of the tree. For example, to iterate through the coordinates in row 0 of matrix A from Figure 4, we specify *level* 1 (for the I dimension) and *parent_node* 0 (for the coordinate in level 1 with value 0, for row 0). If the child is not present, Iterate() returns an empty stream. For T-[UC]$^+$ representations, the extra hardware needed per Scanner to support this feature is similar to simple tree traversal logic.

A new unit called the Sequencer is responsible for calling Iterate() for each Scanner in the order required by the tensor kernel. For this purpose, the Sequencer is configured with a table (indicated by Configure() in Figure 10) that is programmed at kernel start and stores a representation of the kernel loop nest. In our implementation, it stores the bound and stride of each loop in the kernel, a map which associates loop variables to tensor dimensions for indexing purposes, and counters to keep track of the current loop state. For example, for the output-stationary dataflow in Figure 4, the Scanner for matrix A stores $[I, J, K]$ to indicate loop bounds, [1, 1, 1] to indicate stride 1 in all dimensions, and [0, N/A, 1] to indicate that the outermost loop (I, index 0) corresponds to the outer dimension (0) in A, and that the innermost loop (K, index 2) corresponds to the inner dimension (1) in A.[3]

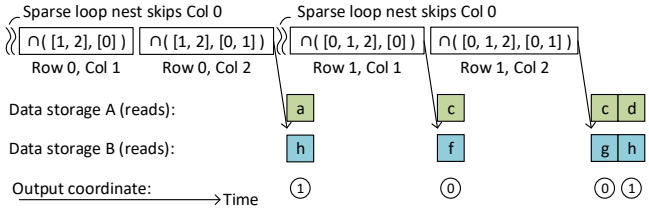

Figure 11: Sparse stream sequencing for Figure 4, output stationary. ∩ is short for calling Iterate on each scanner and performing an intersection. The coordinates for each stream are shown in brackets, where the first/second argument is for matrix A/B respectively.

Efficiently Supporting Sparse Sequences: We optimize the Sequencer by adding support to traverse over sparse loop nests. For example, in Figure 4 for output stationary, a dense loop nest would call Iterate() $J = 3$ times for each row of matrix A, or once for each column of matrix B. Yet, the column with coordinate 0 in matrix B is empty. This case is common in sparse tensors, whose matricizations are often hypersparse (have empty rows or columns) [16, 45]. We would like to exploit this fact and skip a call to Iterate() to save cycles.

To skip Iterate() calls when we encounter runs of empty streams, we add a feedback path from each Scanner back to the Sequencer to inform the Sequencer of the next value of *parent_node* (in increasing order) that has a non-empty stream in the current level (or that there are no such streams left in the current level). This is effective because tensor representation and tensor kernel loop nest are co-designed so that tensor dimensions are iterated over in order, level by level. For example, output stationary iterates over matrix B column by column. The resulting sequence of intersection operations for Figure 4, output stationary, is shown in Figure 11.

[2] This new information may also be specified as a list of coordinates forming the subpath from the root of the tile to the node whose children we are iterating over.

[3] We note that many of the kernels in Table 1 require only this simple logic, but several, e.g., convolution, require support for simple affine expressions. While straightforward to add, we do not discuss those here for simplicity.

Determining the next *parent_node* with a non-empty stream is straightforward. For example, using the CSF representation (compressed in each dimension; c.f., Section 2.1) the sequence of coordinates corresponding to non-empty streams is stored sequentially in memory for each dimension.

Sequencing Tiles: The above discussion concerns sequencing streams within a tile. To run the entire kernel, we add state in the Sequencer for the outer loops that sequence over tiles. Depending on the Scanner implementation, the next tile is filled based on how the metadata storage is implemented (see Section 4.1). As it is loaded, each tile is tagged with coordinates that identify the tile. The Sequencer uses this information to adjust its internal counters, which enables sparse loops (Section 6.1), and passes tile-level coordinates along with the results of each intersection (see below).

6.2 Using Intersected Streams

Post-intersection, surviving coordinates are used to lookup and output tensor data for both tensors. For example, the intersection between row 0 of matrix A and column 2 of matrix B in Figure 4 (left) has the common coordinate 1, which is associated to data a in A and h in B.

To complete these data lookups, we transport data along with coordinates and other T-[UC]$^+$ metadata in each tile. When a tile is initially loaded (*filled*) into the Coordinator, coordinate and other metadata is stored in the metadata storage while data is stored in a separate memory called data storage, shown in Figure 10. To reduce ineffectual data storage reads, we only fetch stored data after intersections have been performed, as shown for our running example in Figure 11.

Implementing this scheme naively requires a content-addressable lookup (by coordinate) into the data storage. We eliminate this expensive lookup by generating a pointer (a *position* or *pos*) into the data storage for each coordinate that is used in an intersection. Positions are generated on the fly, as each tensor is filled into metadata/data storage. Now, the intersection unit operates over tuples of (coordinate, position). The positions of coordinates that survive the intersection are used to lookup the data memory.

As discussed in Section 3.1, tensor data may correspond to scalars or subtiles containing coordinates, other metadata, and data. In both cases, the Coordinator outputs only effectual data, and tags that data with all coordinates required to identify it in the overall tensor (i.e., provides sufficient information to form points; c.f., Section 2.1). In the next section, this will be important when calculating partial outputs.

Pre- vs. Post-Intersection Fills: As an optimization, tensor data within a tile need not be filled into the Coordinator (or fetched at all) unless at least one intersection for that data deems it effectual. We call this scheme *post-intersection fill*, whereas the original scheme fills tensor data *pre-intersection*.

Post-intersection fill can save bandwidth at the Coordinator input (by fetching less data) and storage (as only post-intersection data need be loaded into data memory). To support such a scheme, we need mechanisms to hide the latency of fetching data later, and a new level of indirections to store the post-intersected data in contiguous positions in the data memory. This makes it difficult to implement close to the arithmetic units, where data per coordinate is small (e.g., a scalar) and latency is critical. Yet, we can implement such a scheme close to DRAM, as latency and indirection cost is amortized by the tile fill time and size.

7 ACCELERATOR-LEVEL DESIGN

Finally, we discuss how to integrate Coordinators into the ExTensor accelerator architecture to evaluate tensor kernels. The main new idea is to *hierarchically intersect streams* by placing intersection logic (Sections 4-6) at different levels of an accelerator's memory hierarchy. Coordinators closer to the main memory perform coursergrain intersections, e.g., at tile-level, and pass on effectual work to the next level (closer to the arithmetic) which performs finergrain intersections, e.g., on scalars, before it is finally computed on. Precisely what intersections occur where, in our current design, is discussed in Section 7.4. The resulting architecture streams data from main memory to compute, reducing ineffectual work and data/metadata transfers at multiple granularities as soon as each granularity of ineffectual work is detected.

7.1 Macro Architecture

The architecture we evaluate is shown in Figure 12. DRAM channels connect to a *last-level buffer* (LLB), which stores input tensors, and a *partial output buffer* (POB), which stores partial outputs (e.g., partial sums). These buffers connect over a NoC to *processing elements* (PEs), each of which have *PE-level buffers* (PEBs), arithmetic units and datapath-level registers for performing effectual scalar computations. Sequencers, Scanners and Coordinators (Sections 4-6) are instantiated between each level of the memory hierarchy to sequence and intersect streams of coordinates, which correspond to different levels of tiles. We use a NoC that is configurable to unicast, multicast and broadcast data, similar to previous accelerator proposals [23, 30].

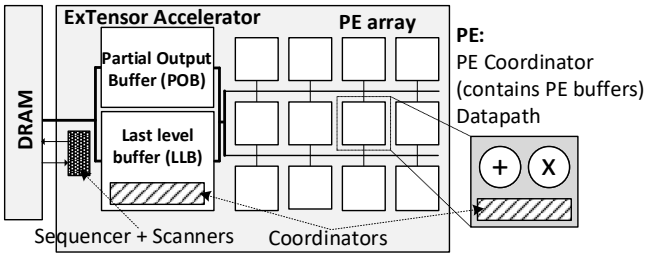

Figure 12: The ExTensor accelerator.

Support for sparse-sparse, sparse-dense and dense-dense computations. A tensor kernel is made up of two or more tensors. We optimize the accelerator for a common case where two sparse, one sparse and one dense, or two dense tensors are the operands of a computation. When evaluating sparse-sparse problems (both tensors are sparse), we perform intersection as described previously. When evaluating a sparse-dense problem, intersection is degenerate as all coordinates of the sparse operand are effectual and those coordinates are used (as positions) to access the values of the dense operand. For dense-dense, intersection is disabled.

Some kernels, e.g., SDDMM in Table 1, benefit from performing analysis across more than two tensors simultaneously, e.g., to intersect un-evaluated subtensors (Section 3.2). For these cases, we allocate an additional Scanner at each PE which shares metadata storage with other Scanners and is idle if not needed.

7.2 Dataflow and Tiling

Coordinators have configuration-time flexibility to support different dataflows by configuring Sequencer state as described in Section 6.1. Figure 13 (right) shows, to continue our running example, a specific dataflow used for matrix multiply that we use in our evaluation. Dataflows for other kernels look similar.

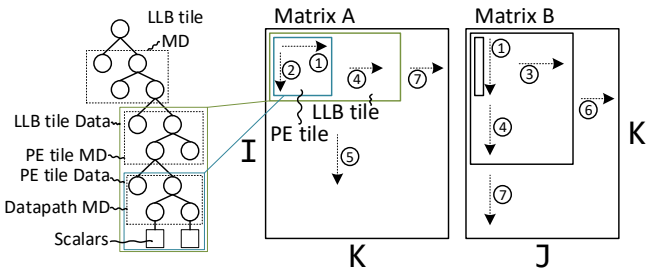

Figure 13: (Left) Tiled matrix representation. (Right) Dataflow for matrix multiplication. MD stands for metadata. Circled numbers indicate loop order, ① in the innermost loop; ⑦ in the outermost loop.

In our dataflow, input matrices A and B are broken into LLB tiles, which are broken into PE tiles, which are broken into datapath tiles. Datapath tiles (a row or column of scalars in the PE) are output stationary (i.e., dimension K is iterated for both matrices simultaneously ①). PE tiles are matrix A-stationary, since we move across all columns in the LLB tile of B before switching to the next PE tile of A ③. LLB tiles are matrix B-stationary, as we move down all LLB tiles of A within a column before going to the next LLB tile of B ⑤.

7.3 Data and Tile Representations

Input tensors are stored in T-$[\text{UC}]^+$ representation at all levels of the memory hierarchy. For the rest of the paper, we assume CSF (T-C^+). To tile at each level, we pre-process each tensor (offline), adding levels of compressed metadata for each level of tiling, as discussed in Section 2.2. The fully tiled representation is shown for matrix A in Figure 13 (left). There are two levels of coordinates, which specify a rectangular subtile, in each level (LLB, PE and datapath). As tiles are broken into subtiles and stored closer to the arithmetic, outer levels of metadata are dropped. For example, only PE tile data is stored in the PEs (PE tile metadata and above is not).

7.3.1 Choosing Tile Sizes. Since tile sizes are chosen in the coordinate space, one tile may contain more non-zeros than another tile. We select the LLB tile size such that each LLB tile fits in the LLB in the worst case. For the tensors we evaluate (Section 8), we found this conservative strategy untenable at the PEs, without incurring a large PEB size. Thus, we size the PEB tile to fit into the PEB in the common case. In the uncommon case, we split the tile across two or more PEs to emulate a larger PEB. Since this design is distributed, our SkipTo() architecture (Section 5) is no longer effective, thus we perform intersections using the baseline scheme (Section 4).

7.4 Staging Data Transfers and Intersections

Sequencers, Scanners and Coordinators between each level of the memory hierarchy perform intersections and stage work to the next level of the hierarchy. We continue our example for matrix multiply; see Figure 13 for terminology.

1.) DRAM-level: Closest to DRAM, a Sequencer and Scanners (i.e., without intersection logic or data memories) bootstrap the kernel by traversing LLB metadata to determine the sequence of LLB tiles to send to the LLB. As they are closest to DRAM, we implement these Scanners' metadata storage as cache memory backed by DRAM.

2.) LLB-level: A Coordinator at the LLB intersects PE tile metadata and fills data post-intersection (Section 6.2). Each intersected coordinate corresponds to a PE tile. Since this level is post-intersection fill, the design avoids reading ineffectual PE tile data from DRAM, within each LLB tile.

3.) PE-level: A Coordinator intersects Datapath metadata and fills Datapath tiles (scalars) pre-intersection (Section 6.2). In our dataflow, each datapath tile is a row or column of scalars in the PE tile for each matrix. Results from each intersection read scalars which are sent to arithmetic units.

The Sequencers traverse their respective loop nests sparse (Section 6.1) at all levels (1)-(3) above, to avoid spending cycles iterating over empty streams. Aside from the DRAM-level Scanners (1), all Scanner metadata storage is implemented as a buffet [43], and tiles from memory closer to DRAM are pushed to lower levels, closer to compute, as they are needed.

7.5 Partial Output Management

After computation (e.g., a dot product in matrix multiply) is performed at the PEs, we need a mechanism to store and combine partial outputs (e.g., partial sums) generated by those computations. Storage for partial outputs is complicated because, unlike input tensors, their sparsity changes dynamically (i.e., the output starts completely empty, but becomes more dense as the computation proceeds).

We make two observations about partial outputs which simplifies their handling. First, if all inputs are sparse tensors, computing on a PE tile will likely generate sparse partial outputs. For example, in output-stationary matrix multiply (Figure 4 (left)) a new partial output will only be generated if the intersection between streams corresponding to a row and column *of that PE tile* was non-empty. PE tiles are small relative to the overall matrix, thus computations at the PE have a smaller chance to generate a partial output. Second, co-iterations in tensor kernels are reduced via associative operators (e.g., sum reduction as in Table 1, max/min [34]), meaning such partial output reductions can be reordered.

Putting the observations together, ExTensor manages partial outputs as follows. First, partial outputs are stored using a content addressable memory alongside the LLB called the partial output buffer (POB), which loads/unloads output tiles from DRAM. Output tiles are stored un-ordered in the COO format [14] in contiguous

Table 2: Real-world tensors used in the evaluation. The top group is matrices from SparseSuite [16]. The bottom group are higher order tensors from FROSTT [45].

Tensor (Domain)	Dimensions	Non-zeros (Density)
mbeacxc (Economic Problem)	496×496	49.9k (20.3%)
bcsstk13 (Structural)	$2k \times 2k$	83.9k (2.09%)
bcsstk10 (Structural)	$1.1k \times 1.1k$	22k (1.87%)
bcsstk17 (Structural)	$11k \times 11k$	428.6k (0.356%)
pdb1HYS (Protein Data Bank)	$36.4k \times 36.4k$	4.3M (0.328%)
rma10 (Fluid Dynamics)	$46.8k \times 46.8k$	2.3M (0.106%)
cant (FEM cantilever)	$62.5k \times 62.5k$	4M (0.103%)
consph (FEM concentric spheres)	$83.3k \times 83.3k$	6M (0.087%)
pwtk (Pressurized wind tunnel)	$217.9k \times 217.9k$	11.5M (0.024%)
shipsec1 (FEM Ship section / detail)	$140.8k \times 140.8k$	3.6M (0.018%)
mac_econ_fwd500 (Macroeconomic model)	$206.5k \times 206.5k$	1.3M (0.003%)
Higher order tensors		
Chicago Crime Data (Security)	$6.2k \times 24 \times 2.5k$	5.3M (1.46%)
Uber Pickups (Transportation)	$4.4k \times 1.1k \times 1.7k$	3.3M (0.0385%)
NIPS Publications (Academia)	$2.5k \times 2.8k \times 239k$	3.1M (0.0002%)

regions of DRAM.[4] Each DRAM region is allocated based on the worst-case output tile size. When output tiles are filled to the POB, the COO is converted to a content addressable hash table representation; the hash table is iterated sequentially to unload back to DRAM. Partial outputs produced by the PEs are sent to the POB, along with their coordinates (Section 6.1), where the coordinates are hashed to locate the old output value and the new value is locally accumulated. If the hash table contains no entry (coordinate) for the given output, one is allocated. Note, accumulations occur in the background, off the critical path.

7.6 Programming Model

The complete ExTensor accelerator is configured to run a specific tensor algebra kernel, e.g., kernels like those discussed in Section 2.2. A configuration specifies the number of input tensors, the tiling for each tensor, a representation of the kernel loop nest/dataflow and connectivity between buffers, intersection units and math datapaths. Details of the low-level specification are omitted, but involve configuring DMA engines, partitioning buffers, setting control FSMs, and defining on-chip network routes between physical components.

The accelerator is not Turing complete, but is designed to compute all algebra equations accepted by the TACO Tensor Algebra Compiler [26]. Generating configurations directly from TACO is future work.

8 EVALUATION

We now evaluate the final ExTensor design proposed in Section 7. In this evaluation, we compare ExTensor variants to optimized CPU codes, perform a design space exploration for ExTensor, and provide hardware area figures for the ExTensor Scanner unit.

8.1 Methodology

Tensors. We evaluate performance using real-life tensors from the FROSTT [45] tensor data set and the SuiteSparse matrix collection [16] shown in Table 2. These datasets span multiple domains (from fluid dynamics to macroeconomic models), non-zero densities, and number of dimensions. The tensors are pre-processed into the T-[UC]$^+$ representation T-C$^+$, also known as CSF (Section 7.3). We also evaluate several layers of pruned AlexNet [29] with their densities taken from [42].

Tensor kernels. We evaluate these tensors on kernels selected from Table 1, including GEMM, TTV, TTM, and SDDMM. Of these, TTV and TTM compute on 3-dimensional tensors. SDDMM computes on 3 matrices. All the tensors used in the evaluation use double precision data.

Simulation framework. We evaluate ExTensor using a model written in Python that evaluates performance for a given tensor kernel and input tensors. The performance model has two components. First, the core intersection logic—that includes the Coordinator and ALUs—are modeled at cycle level for high fidelity. This evaluates the number of cycles required to intersect two PE-level tiles (Section 7.3). Second, the rest of the accelerator components—such as the LLB, NoC, DRAM interface, and scheduling logic—are modeled analytically. These components evaluate data accesses, scheduling, output handling, and the on-chip dataflow (Section 7.2).

DRAM and NoC use a queuing model where data transfers are not allowed to exceed a peak bandwidth. We believe this queuing model should be sufficient to model DRAM and NoC because of ExTensor's data access/movement patterns. Specifically, since ExTensor accesses data at tile granularity, we achieve good spatial locality (at DRAM burst- and row buffer-granularity) and therefore achieve high DRAM utilization during periods where data is being read. Similarly, based on our dataflow (Section 7.2), we multicast/unicast tiles across PEs in a fixed and regular order. This allows us to model NoCs analytically such that the peak bandwidth is never exceeded.

Baseline platform. We compare ExTensor against an Intel Xeon E5-2687W v4 part, a high-end server CPU which has 12/24 cores/threads running at 3 GHz (Turbo 3.5 GHz) with a 30 MB LLC. The CPU has 4 DRAM channels, providing a theoretical maximum bandwidth of 68.256 GB/s. For each kernel, we compare to either Intel's MKL library [49] or the TACO tensor compiler [26], depending on which one gives better performance.

8.2 Main Result

We now present our main result: ExTensor performance relative to the CPU baseline from Section 8.1 on all tensor kernels. We compare two design variants:

ExTensor-skip. Our final design from Section 7, assuming a 1 GHz clock.[5] To make the comparison apples-to-apples, we set ExTensor's LLB size and DRAM bandwidth to match the CPU. Based on design space studies, we use 128 PEs with 64 KB of PEB (local buffer including metadata and data storage for all Scanners) each. Each Scanner uses a coarse-grain CAM with $T = 32$ entries.

ExTensor-no-skip. Same as ExTensor-skip, but without the intersection optimizations described in Section 5. We note that this version still performs intersection; a design that does not intersect would perform the computation dense.

[4] In COO, each output is stored alongside its point/coordinates, which is flattened to save space.

[5] Note, this is lower than the CPU clock. Increasing ExTensor's clock frequency would significantly improve performance.

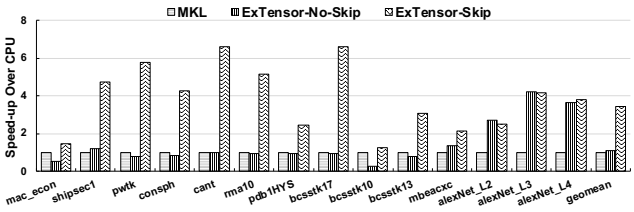

Figure 14: ExTensor speed-up relative to MKL (SpMSpM).

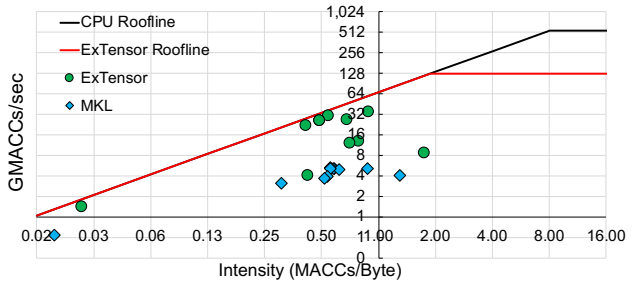

Figure 15: Roofline analysis of ExTensor and CPU (SpMSpM).

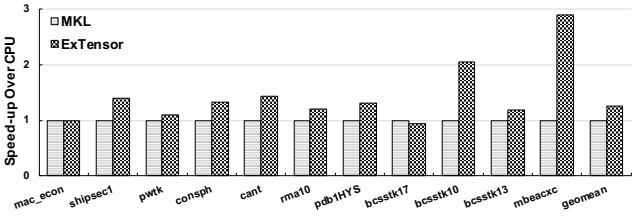

Figure 16: ExTensor speed-up relative to MKL (SpMM).

8.2.1 GEneralized Matrix Multiplication (GEMM). GEMM is a core kernel in linear algebra and continues to remain as one of the most widely used kernels in modern computing with its applications ranging from deep learning to graph processing. We evaluate two variants of GEMM:

Sparse-Sparse GEMM (SpMSpM, or SpGEMM): In this kernel, a sparse matrix is multiplied by a sparse matrix. SpMSpM has a wide range of applications, including linear algebra based graph processing [34] and deep learning with sparse neural networks [21]. We evaluate SpMSpM by multiplying matrices from the SuiteSparse collection by themselves, similar to graph algorithms such as finding nearest neighbors and triangle counting [34].

Figure 14 compares ExTensor variants and MKL for SpMSpM. ExTensor-skip is 3.4× faster than the CPU on average and outperforms the CPU for every matrix. ExTensor improves performance in two ways. First, logic close to the memory (Sequencers and Scanners, hierarchical Coordinators; Section 7) look ahead in the computation to ensure the DRAM bus is highly utilized. Second, intersection logic at each level (e.g., between DRAM and the LLB, between the PE and datapath) removes ineffectual data reads (e.g., using tile metadata) — thus removing redundant bandwidth utilization. ExTensor-skip is more effective than ExTensor-no-skip in workloads with higher sparsity, evident from the results where

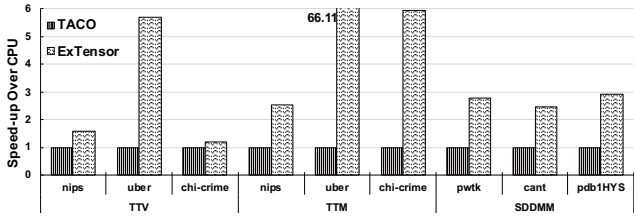

Figure 17: Performance comparison between ExTensor variants and TACO for generalized tensor algebra.

ExTensor-skip outperforms ExTensor-no-skip in all workloads other than AlexNet, which has higher density. Figure 15 presents the roofline analysis, which shows that MKL is memory latency bound on SpMSpM while ExTensor is close to memory bandwidth bound.

Interestingly, speedup is not a function of matrix sparsity. This is because the core strength of ExTensor is intersection. Intersection eliminates work based on the sparsity *pattern*, not simply the sparsity, at both the LLB at PE tile granularity and within the PE at scalar granularity (Section 7.4). In particular, we see a ~ 3.1× performance improvement for ExTensor-skip relative to ExTensor-no-skip, which illustrates the importance of sparsity pattern at the PE level where the kernel is sensitive to intersection time.

Sparse-Dense GEMM (SpMM): In this kernel, a sparse matrix is multiplied by a dense matrix. ExTensor evaluates SpMM by multiplying each matrix in Table 2 by a randomly generated dense matrix with 32 columns. The multiplication of a sparse matrix by a tall-and-skinny dense matrix is a key kernel in several applications such as algebraic graph algorithms [3, 47].

Figure 16 compares ExTensor variants and MKL for the SpMM kernel. ExTensor-skip performance improves the performance by 1.3× over the CPU on average. The performance gap between ExTensor and CPU is narrowed in SpMM compared to SpMSpM kernel because SpMM is more DRAM bandwidth bound and ExTensor is provisioned with the same bandwidth. Recall from Section 7, intersection of streams of coordinates from a sparse and a dense tensor simply yields the coordinate stream from the sparse tensor. Thus, our performance improvement comes from higher DRAM utilization using Scanners close to memory. We performed an analogous roofline analysis (not shown for space) which shows both the CPU and ExTensor DRAM bandwidth bound for SpMM, which points to the memory-bound nature of SpMM for both CPU and ExTensor leaving little room to improve over the CPU.

8.2.2 Generalized Tensor Algebra. Figure 17 evaluates higher-order tensor kernels TTV and TTM, along with the SDDMM kernel, against TACO, a state of the art compiler for tensor algebra [26]. TTV and TTM contract a 3-dimensional sparse tensor with a dense vector or matrix, respectively. SDDMM computes a filtered matrix-matrix product, i.e., a Hadamard product between a sparse matrix and a product of two smaller dense rectangular matrices. The Hadamard product in SDDMM provides opportunity for intersection to avoid computation of whole dot-products.

We see 2.8×, 24.9×, and 2.7× average speedups for TTV, TTM, and SDDMM kernels respectively. In TTM and TTM kernels, ExTensor uses a 3-level tiling strategy to improve data reuse. TACO is

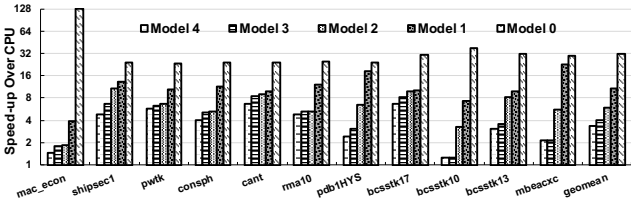

Figure 18: ExTensor bottleneck study with design variants.

slower for all the kernels, mainly due to its lack of tiling and inability to efficiently avoid ineffectual work. Hierarchical intersection permits ExTensor to avoid ineffectual tiles being scheduled to the LLB/PEs, which is pronounced in these extremely sparse higher order tensors. Similarly in SDDMM, most of our benefit comes from the ability to skip entire dot products/tiles based on the sparsity of the C matrix (Table 1).

8.3 Bottleneck Analysis

Figure 18 studies a set of ExTensor variants, ranging in design realism, to give insight in what future changes can improve the design by how much. This methodology is similar to the Eyexam in [13]. We study the following design variants:

• **Model 0** (most idealized) takes into account only the number of MACC units available in ExTensor. For each matrix, the number of effectual multiplications is calculated. Then, runtime calculated by dividing the number of multiplications by available MACC units.

• **Model 1** adds DRAM bandwidth constraints to the previous model. Specifically, runtime is calculated as the maximum of model 0's time and the time it takes to read/write back the input matrices/result matrix once over the DRAM bus.

• **Model 2** adds a realistic LLB tiling scheme to the previous model. LLB tiling is modeled accurately, while on-chip behavior is idealized. PE tiles are distributed to PEs in an idealized fashion that tries to minimize load imbalance (as opposed to being constrained to a specific on-chip dataflow) and intersection time is calculated as the number of effectual MACCs.

• **Model 3** adds a realistic CAM-based PE intersect unit to the previous model.

• **Model 4** (most realistic) is the ExTensor model we evaluate in Section 8.2, which adds a realistic PE tile distribution unit to the previous model.

Model 0 shows the performance ceiling given any accelerator. Model 0 and 1 are likely un-implementable. We believe Models 2 and 3 may be implementable with improved methods for distributing PE tiles and performing intersection. On average, there is a 1.8×, 1.4× and 1.2× performance gap between Model 1-2, Model 2-3 and Model 3-4, respectively.

8.4 Synthetic Data Studies

In Section 8.2, we presented results for real-life matrices. This makes analyzing ExTensor's efficiency hard, because each real-life matrix has different dimensions, sparsity, and sparsity pattern. To isolate performance effects, Figure 19 shows a study on synthetically generated matrices using a uniform sparsity distribution, varying sparsity and matrix dimension. Specifically, for fixed numbers of non-zeros

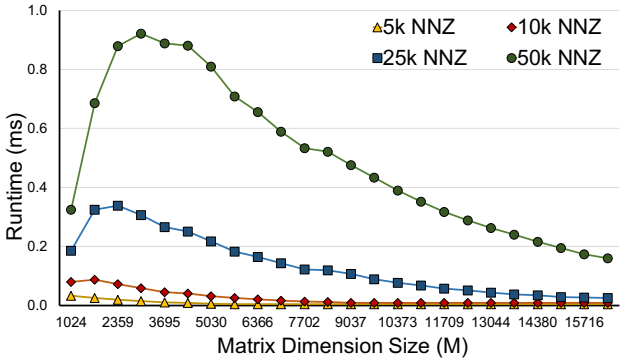

Figure 19: ExTensor's SpMSpM performance across varying dimension sizes with constant number of non-zeros (NNZ) per matrix.

(NNZ), we generate square matrices ($I = J = K$) of different dimension sizes and evaluate runtime on SpMSpM. For all points, we set PE tile size to a constant 128x128 elements in coordinate space.

For each NNZ count, we observe three performance regimes. First, when the dimension sizes are small and sparsity is low (e.g., when $I < 3600$ for 50k NNZ), the matrices only contain empty scalars but not empty PE tiles or empty rows/columns within a tile (on average). In this case, as we increase the dimension size, the runtime increases because the number of non-empty tiles increases, outweighing the benefit of scalar-granularity skips. Second, once we reach a sufficiently high sparsity (namely, when NNZ = $O(I)$ per tile), the trend reverses. During this phase, although the number of non-empty tiles continues to increase, we begin to see empty rows/columns within a PE tile which means the sparse loop optimization (Section 6.1) activates, which reduces overall runtime. Third, once the sparsity is high enough to see empty PE tiles, the runtime flattens with increasing sparsity as any non-empty tile is likely to have very few (e.g., 1) non-zeros.

8.5 Area and Bandwidth Studies

In the previous sections, we normalized ExTensor on-chip storage and DRAM bandwidth to exemplify the benefit of our intersection technique and architecture. We now perform two studies showing design sensitivity to these parameters.

Sweep DRAM bandwidth, fixing area. First, Figure 20 shows the impact of increasing DRAM bandwidth for ExTensor running SpMSpM while keeping the same on-chip area as in the previous section. We sweep from the baseline bandwidth to ~ 550 GB/s, which is more representative of (but still less than) modern GPUs. We found that ExTensor Model 4 (the complete design) has bottlenecks that prevent performance scaling with DRAM bandwidth. Specifically, increasing DRAM bandwidth to ~ 550 GB/s improves performance by 3% only for Model 4. Thus, the results in Figure 20 are for ExTensor Model 2, which idealizes intersection and PE tile distribution (Section 8.3). At this fidelity, large matrices that do not fit into the LLB see performance scaling. Small matrices (e.g., mbeacxc) see less benefit, as those kernels are dominated by other factors such as latency. Average performance improvement is 2.1× for an 8× bandwidth increase. We consider addressing the bottlenecks in Model 4 that prevent scaling to be future work.

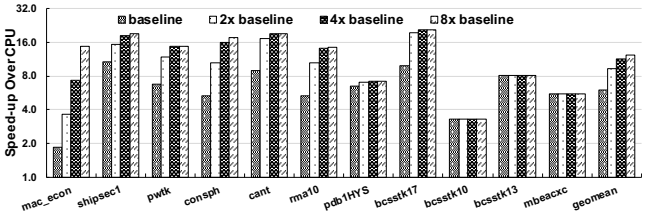

Figure 20: ExTensor (Model 2) SpMSpM speed-up over the CPU, scaling ExTensor DRAM bandwidth.

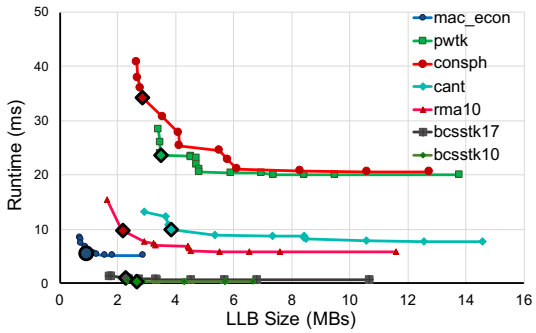

Figure 21: Pareto frontiers for SpMSpM design space. Performance per area optimal points are shown with a black outline.

Fix DRAM bandwidth, sweep area. Second, Figure 21 shows the impact of changing total LLB size, which is a dominant source of overall area. Once a kernel's runtime levels off, we stop increasing the LLB. The takeaway is that for many matrices, performance is not sensitive to LLB size until the LLB shrinks to 5-10 MB. Similar to the effectiveness of intersection, this highly depends on exact sparsity distribution in each matrix as opposed to absolute matrix size, as even a small matrix can generate a relatively dense output depending on non-zero distribution.

8.6 Hardware Implementation

We focused our RTL implementation on the novel parts of the accelerator, in particular the Coordinator, since the rest of the modules (ALUs [19], NoC [30], and buffers [37], etc.) are already well studied. We implement the Coordinator design in Verilog, assuming a 64 KB PE buffer and a content addressable memory with $T = 32$ entries with 8-bit coordinates (sufficient to represent the tile sizes used in our evaluation). We synthesize the design using a 32 nm commercial process, which meets timing at 1 GHz to match our performance evaluation. Area numbers for SRAM were obtained from CACTI [37] and for compute were taken from [19] scaled to 32nm [11]. Area for intersection is taken from synthesis. Overall, we find that PE SRAM buffer takes 79% of the area, double precision arithmetic takes 12% and logic for intersection takes 9%.

Extrapolating from PE area plus projected LLB+POB area, we estimate total chip area (using the parameters in Section 8.2) to be $96.89mm^2$, with the intersection logic across PEs consuming only $2.38mm^2$ (2.46% of total) in 32nm. The main take away is that the Coordinator logic does not dominate the accelerator area budget.

9 RELATED WORK

Accelerator designs targeting tensor algebra [2, 8, 24, 33, 36, 40, 41] and machine learning [4, 12, 13, 21, 25, 32, 42, 44, 51] are receiving significant attention. Furthermore, these accelerator designs also exploit sparsity to reduce both time and energy. We now compare ExTensor to tensor algebra and machine learning accelerators.

Tensor Algebra Accelerators. The ubiquity of matrix operations in big data applications has spurred research work on accelerating sparse matrix-dense vector (SpMV) and SpMM kernels [2, 8, 24, 33, 39]. Intersecting coordinates from sparse matrices to dense data is trivial, and these works skip ineffectual computations (e.g. single multiplications or entire dot products based on empty rows/columns) as a side effect of the compressed format for the sparse matrix (e.g., CSR/CSC). Yet, none of these designs extend to tensors with degrees higher than two or to compound expressions such as SDDMM, which is the focus of ExTensor.

Recent work has investigated accelerators for sparse matrix sparse vector (SpMSpV) multiplication and SpMSpM [41, 50]. Several works perform inner-product matrix multiplication by loading the entire sparse matrix B into the processor and then processing sparse matrix A on a row-by-row basis [31, 40, 50]. These proposals apply coordinate intersection after fetching the values and meta data from memory, which leads to cases where data loaded is never used. ExTensor avoids this memory traffic by hierarchically intersecting tiles and subtiles at each level of the memory/buffer hierarchy and lazily loading values after intersection. Alternately, accelerators have focused on outer-product multiplication to avoid the need for coordinate intersection [36, 41]. With this approach, each *column* in sparse matrix A is multiplied by the elements in each *row* of sparse matrix B to generate partial products. In essence, this approach improves memory traffic efficiency for the multiplication step but introduces an additional merge step where the many sets of partial products are read from memory and reduced into the final products. ExTensor performs efficient inner-product multiplication that achieves similar memory traffic by using intersections, skips, and tiling to reduce value and meta data memory traffic.

Machine Learning Accelerators. Recent CNN accelerators have also recognized the prevalence of ineffectual computations (e.g., [4, 12, 13, 21, 42, 51]). While these designs focus on saving energy and time by skipping *single* multiplications, they still incur memory access overhead of bringing the operands on-chip. ExTensor can skip entire dot products when a tensor row/column/tile is zero. In doing so, ExTensor reduces time and energy by avoiding memory accesses associated with ineffectual computations.

10 CONCLUSION

General tensor algebra represents a ripe domain for contributions from computer architects. This paper presented ExTensor, a new approach for performing general tensor algebra using hierarchical and compositional intersection. There are multiple avenues for future work. For example, efficient conversion between compressed formats on the fly, online as opposed to offline tiling, support for SIMD datapaths, and how to address the performance artifacts hindering bandwidth scaling (Section 8.5). In the long run, we believe that the broad applicability of algebra on compressed tensors means that they should become a foundational building block of accelerator construction.

ACKNOWLEDGMENTS

This work was funded in part by a DARPA SDH grant, number HR0011-18-3-0007. We thank Fredrik Kjolstad, Stephen Chou and Saman Amarasinghe for inspiring discussions, and thank the anonymous reviewers for their feedback and insights during the review process. This research was, in part, funded by the U.S. Government. The views and conclusions contained in this document are those of the authors and should not be interpreted as representing the official policies, either expressed or implied, of the U.S. Government.

REFERENCES

[1] Martín Abadi, Paul Barham, Jianmin Chen, Zhifeng Chen, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Geoffrey Irving, Michael Isard, et al. 2016. Tensorflow: a system for large-scale machine learning. In *OSDI'16*.

[2] Seher Acer, Oguz Selvitopi, and Cevdet Aykanat. 2016. Improving performance of sparse matrix dense matrix multiplication on large-scale parallel systems. *Parallel Comput.* (2016).

[3] Hasan Metin Aktulga, Aydin Buluç, Samuel Williams, and Chao Yang. 2014. Optimizing sparse matrix-multiple vectors multiplication for nuclear configuration interaction calculations. In *IPDPS'14*.

[4] Jorge Albericio, Patrick Judd, Tayler Hetherington, Tor Aamodt, Natalie Enright Jerger, and Andreas Moshovos. 2016. Cnvlutin: Ineffectual-neuron-free Deep Neural Network Computing. In *ISCA'16*.

[5] Animashree Anandkumar, Rong Ge, Daniel Hsu, Sham M Kakade, and Matus Telgarsky. 2014. Tensor decompositions for learning latent variable models. *The Journal of Machine Learning Research* (2014).

[6] A. Azad, A. Buluc, and J. Gilbert. 2015. Parallel Triangle Counting and Enumeration Using Matrix Algebra. In *IPDPS'15 Workshop*.

[7] Brett W Bader, Michael W Berry, and Murray Browne. 2008. Discussion tracking in Enron email using PARAFAC. In *Survey of Text Mining II*. Springer, 147–163.

[8] Michael A Bender, Gerth Stølting Brodal, Rolf Fagerberg, Riko Jacob, and Elias Vicari. 2010. Optimal sparse matrix dense vector multiplication in the I/O-model. *Theory of Computing Systems* (2010).

[9] James Bennett, Stan Lanning, et al. 2007. The netflix prize. In *Proceedings of KDD cup and workshop (2007)*.

[10] L Susan Blackford, Antoine Petitet, Roldan Pozo, Karin Remington, R Clint Whaley, James Demmel, Jack Dongarra, Iain Duff, Sven Hammarling, Greg Henry, et al. 2002. An updated set of basic linear algebra subprograms (BLAS). *ACM Trans. Math. Software* (2002).

[11] Mark Bohr. 2010. The new era of scaling in an SoC world. In *ISSCC'00*.

[12] Yu-Hsin Chen, Joel Emer, and Vivienne Sze. 2016. Eyeriss: A Spatial Architecture for Energy-efficient Dataflow for Convolutional Neural Networks. In *ISCA'16*.

[13] Yu-Hsin Chen, Joel Emer, and Vivienne Sze. 2018. Eyeriss v2: A Flexible and High-Performance Accelerator for Emerging Deep Neural Networks. arXiv:1807.07928

[14] Stephen Chou, Fredrik Kjolstad, and Saman Amarasinghe. 2018. Format Abstraction for Sparse Tensor Algebra Compilers. *Proceedings of the ACM on Programming Languages* (2018).

[15] Andrzej Cichocki, Namgil Lee, Ivan Oseledets, Anh-Huy Phan, Qibin Zhao, and Danilo P. Mandic. 2016. Tensor Networks for Dimensionality Reduction and Large-scale Optimization: Part 1 Low-Rank Tensor Decompositions. *Foundations and Trends in Machine Learning* (2016).

[16] Timothy A Davis and Yifan Hu. 2011. The University of Florida sparse matrix collection. (2011). https://sparse.tamu.edu

[17] Mehmet Deveci, Simon D Hammond, Michael M Wolf, and Sivasankaran Rajamanickam. 2018. Sparse Matrix-Matrix Multiplication on Multilevel Memory Architectures: Algorithms and Experiments. *arXiv preprint arXiv:1804.00695* (2018).

[18] Richard P Feynman, Robert B Leighton, and Matthew Sands. 2011. *The Feynman lectures on physics, Vol. I: The new millennium edition: mainly mechanics, radiation, and heat.* Basic books.

[19] Sameh Galal and Mark Horowitz. 2011. Energy-efficient floating-point unit design. *IEEE Transactions on computers* (2011).

[20] Lars Grasedyck, Daniel Kressner, and Christine Tobler. 2013. A literature survey of low-rank tensor approximation techniques. *GAMM-Mitteilungen* (2013).

[21] Song Han, Xingyu Liu, Huizi Mao, Jing Pu, Ardavan Pedram, Mark A Horowitz, and William J Dally. 2016. EIE: efficient inference engine on compressed deep neural network. In *ISCA'16*.

[22] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. 2016. Deep Residual Learning for Image Recognition. In *CVPR'16*.

[23] K. Hegde, R. Agrawal, Y. Yao, and C. Fletcher. 2018. Morph: Flexible Acceleration for 3D CNN-basedVideo Understanding. In *MICRO'18*.

[24] Alexander Friedrich Heinecke. 2008. Cache Optimised Data Structures and Algorithms for Sparse Matrices. *Bachelorarbeit in Informatik, Technische Universität München* (2008).

[25] Norman P. Jouppi, Cliff Young, Nishant Patil, David Patterson, Gaurav Agrawal, Raminder Bajwa, Sarah Bates, Suresh Bhatia, Nan Boden, Al Borchers, Rick Boyle, Pierre-luc Cantin, Clifford Chao, Chris Clark, Jeremy Coriell, Mike Daley, Matt Dau, Jeffrey Dean, Ben Gelb, Tara Vazir Ghaemmaghami, Rajendra Gottipati, William Gulland, Robert Hagmann, Richard C. Ho, Doug Hogberg, John Hu, Robert Hundt, Dan Hurt, Julian Ibarz, Aaron Jaffey, Alek Jaworski, Alexander Kaplan, Harshit Khaitan, Andy Koch, Naveen Kumar, Steve Lacy, James Laudon, James Law, Diemthu Le, Chris Leary, Zhuyuan Liu, Kyle Lucke, Alan Lundin, Gordon MacKean, Adriana Maggiore, Maire Mahony, Kieran Miller, Rahul Nagarajan, Ravi Narayanaswami, Ray Ni, Kathy Nix, Thomas Norrie, Mark Omernick, Narayana Penukonda, Andy Phelps, Jonathan Ross, Amir Salek, Emad Samadiani, Chris Severn, Gregory Sizikov, Matthew Snelham, Jed Souter, Dan Steinberg, Andy Swing, Mercedes Tan, Gregory Thorson, Bo Tian, Horia Toma, Erick Tuttle, Vijay Vasudevan, Richard Walter, Walter Wang, Eric Wilcox, and Doe Hyun Yoon. 2017. In-Datacenter Performance Analysis of a Tensor Processing Unit. In *ISCA'17*.

[26] Fredrik Kjolstad, Shoaib Kamil, Stephen Chou, David Lugato, and Saman Amarasinghe. 2017. The Tensor Algebra Compiler. *Proceedings of the ACM on Programming Languages* (2017).

[27] T. Kolda and B. Bader. 2009. Tensor Decompositions and Applications. *SIAM Rev.* (2009).

[28] Joseph C Kolecki. 2002. An introduction to tensors for students of physics and engineering. (2002).

[29] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E. Hinton. 2012. Imagenet classification with deep convolutional neural networks. In *NIPS'12*.

[30] Hyoukjun Kwon, Ananda Samajdar, and Tushar Krishna. 2018. MAERI: Enabling Flexible Dataflow Mapping over DNN Accelerators via Reconfigurable Interconnects. *SIGPLAN Not.* (2018).

[31] C. Y. Lin, Z. Zhang, N. Wong, and H. K. So. 2010. Design space exploration for sparse matrix-matrix multiplication on FPGAs. In *FPT'10*.

[32] Daofu Liu, Tianshi Chen, Shaoli Liu, Jinhong Zhou, Shengyuan Zhou, Olivier Teman, Xiaobing Feng, Xuehai Zhou, and Yunji Chen. 2015. PuDianNao: A Polyvalent Machine Learning Accelerator. *SIGPLAN Not.* (2015).

[33] Kiran Matam, Siva Rama Krishna Bharadwaj Indarapu, and Kishore Kothapalli. [n. d.]. Sparse matrix-matrix multiplication on modern architectures. In *HiPC'12*.

[34] T. Mattson, D. Bader, J. Berry, A. Buluc, J. Dongarra, C. Faloutsos, J. Feo, J. Gilbert, J. Gonzalez, B. Hendrickson, J. Kepner, C. Leiserson, A. Lumsdaine, D. Padua, S. Poole, S. Reinhardt, M. Stonebraker, S. Wallach, and A. Yoo. 2013. Standards for graph algorithm primitives. In *HPEC'13*.

[35] Julian McAuley and Jure Leskovec. 2013. Hidden factors and hidden topics: understanding rating dimensions with review text. In *Proceedings of the 7th ACM conference on Recommender systems*. ACM.

[36] Asit K Mishra, Eriko Nurvitadhi, Ganesh Venkatesh, Jonathan Pearce, and Debbie Marr. 2017. Fine-grained accelerators for sparse machine learning workloads. In *ASP-DAC'17*.

[37] Naveen Muralimanohar and Rajeev Balasubramonian. 2009. CACTI 6.0: A Tool to Understand Large Caches.

[38] Yusuke Nagasaka, Satoshi Matsuoka, Ariful Azad, and Aydin Buluc. 2018. High-performance sparse matrix-matrix products on Intel KNL and multicore architectures. *arXiv preprint arXiv:1804.01698* (2018).

[39] Eriko Nurvitadhi, Asit Mishra, and Debbie Marr. 2015. A Sparse Matrix Vector Multiply Accelerator for Support Vector Machine. In *Proceedings of the 2015 International Conference on Compilers, Architecture and Synthesis for Embedded Systems*. IEEE.

[40] Eriko Nurvitadhi, Asit Mishra, Yu Wang, Ganesh Venkatesh, and Debbie Marr. 2016. Hardware accelerator for analytics of sparse data. In *DATE'16*. EDA Consortium.

[41] 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 *HPCA'18*.

[42] Angshuman Parashar, Minsoo Rhu, Anurag Mukkara, Antonio Puglielli, Rangharajan Venkatesan, Brucek Khailany, Joel Emer, Stephen W Keckler, and William J Dally. 2017. Scnn: An accelerator for compressed-sparse convolutional neural networks. In *ISCA'17*.

[43] M. Pellauer, Y. Shao, J. Clemons, N. Crago, K. Hegde, R. Venkatesan, S. Keckler, C. Fletcher, and J. Emer. 2019. Buffets: An Efficient and Composable Storage Idiom for Explicit Decoupled Data Orchestration. In *ASPLOS'19*.

[44] Raghu Prabhakar, Yaqi Zhang, David Koeplinger, Matt Feldman, Tian Zhao, Stefan Hadjis, Ardavan Pedram, Christos Kozyrakis, and Kunle Olukotun. 2017. Plasticine: A Reconfigurable Architecture For Parallel Paterns. *SIGARCH Computer Architecture News* (2017).

[45] 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/

[46] Shaden Smith and George Karypis. 2015. Tensor-matrix products with a compressed sparse tensor. In *Proceedings of the 5th Workshop on Irregular Applications:*

Architectures and Algorithms.

[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*.

[48] Bimal Viswanath, Alan Mislove, Meeyoung Cha, and Krishna P Gummadi. 2009. On the evolution of user interaction in facebook. In *Proceedings of the 2nd ACM workshop on Online social networks*. ACM.

[49] Endong Wang, Qing Zhang, Bo Shen, Guangyong Zhang, Xiaowei Lu, Qing Wu, and Yajuan Wang. 2014. Intel math kernel library. In *High-Performance Computing on the Intel® Xeon Phi*. Springer.

[50] L. Yavits, A. Morad, and R. Ginosar. 2015. Sparse Matrix Multiplication On An Associative Processor. *IEEE Transactions on Parallel and Distributed Systems* (2015).

[51] S. Zhang, Z. Du, L. Zhang, H. Lan, S. Liu, L. Li, Q. Guo, T. Chen, and Y. Chen. 2016. Cambricon-X: An accelerator for sparse neural networks. In *MICRO'16*.

[52] Huasha Zhao. 2014. High Performance Machine Learning through Codesign and Rooflining. *PhD Thesis* (2014).