A Tensor Marshaling Unit for Sparse Tensor Algebra on General-Purpose Processors

Marco Siracusa
marco.siracusa@bsc.es
Barcelona Supercomputing Center
Universitat Politècnica de Catalunya,
Barcelona, Spain

Víctor Soria-Pardos
victor.soria@bsc.es
Barcelona Supercomputing Center
Universitat Politècnica de Catalunya,
Barcelona, Spain

Francesco Sgherzi
francesco.sgherzi@bsc.es
Barcelona Supercomputing Center
Universitat Politècnica de Catalunya,
Barcelona, Spain

Joshua Randall
joshua.randall@arm.com
Arm
Austin, Texas, USA

Douglas J. Joseph
doug.joseph@samsung.com
Samsung
Austin, Texas, USA

Miquel Moretó Planas
miquel.moreto@bsc.es
Barcelona Supercomputing Center
Universitat Politècnica de Catalunya,
Barcelona, Spain

Adrià Armejach
adria.armejach@bsc.es
Barcelona Supercomputing Center
Universitat Politècnica de Catalunya,
Barcelona, Spain

ABSTRACT

This paper proposes the Tensor Marshaling Unit (TMU), a near-core programmable dataflow engine for multicore architectures that accelerates tensor traversals and merging, the most critical operations of sparse tensor workloads running on today's computing infrastructures. The TMU leverages a novel multi-lane design that enables parallel tensor loading and merging, which naturally produces vector operands that are marshaled into the core for efficient SIMD computation. The TMU supports all the necessary primitives to be tensor-format and tensor-algebra complete. We evaluate the TMU on a simulated multicore system using a broad set of tensor algebra workloads, achieving 3.6×, 2.8×, and 4.9× speedups over memory-intensive, compute-intensive, and merge-intensive vectorized software implementations, respectively.

CCS CONCEPTS

• **Computer systems organization → Data flow architectures**; • **Hardware → Hardware accelerators**; • **Computing methodologies →** *Parallel programming languages.*

KEYWORDS

Dataflow accelerator, parallel tensor traversal, tensor merging, sparse tensor algebra, vectorization

ACM Reference Format:
Marco Siracusa, Víctor Soria-Pardos, Francesco Sgherzi, Joshua Randall, Douglas J. Joseph, Miquel Moretó Planas, and Adrià Armejach. 2023. A Tensor Marshaling Unit for Sparse Tensor Algebra on General-Purpose Processors. In *56th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO '23), October 28–November 01, 2023, Toronto, ON, Canada.* ACM, New York, NY, USA, 15 pages. https://doi.org/10.1145/3613424.3614284

1 INTRODUCTION

Recent advancements in multilinear algebra have opened up new avenues for solving a wide range of problems using tensor algebra methods. These methods are effective in various domains, including scientific computing [20, 30, 35], genomics [8, 21, 22, 51], graph processing [4, 7, 29], and both traditional [9, 25, 47, 53] and emerging [15, 23, 42, 48, 68] machine learning workloads. Typical operands of these applications are both (i) large sparse tensors, which store multilinear relationships among the entities involved in the problem, and (ii) smaller dense tensors, which store entities' features. Given the high sparsity of these relationships (i.e., most values of the multidimensional space are zeros), generally above 99% [56], sparse tensors are encoded in compressed formats only storing non-zero values and their positions [6, 12, 57].

Although these formats are essential to reduce sparse problems to tractable complexities, computation involving sparse tensors requires intensive traversal and merging operations to aggregate tensor operands [31]. On current CPUs, both traversal and merging have extensive *data-dependent control flow*, leading to frequent branch mispredictions and consequent pipeline flushes that limit performance [61]. Moreover, traversals typically loads non-contiguous data, leading to irregular memory access patterns that generate *long-latency memory accesses*, further hindering performance [60]. Overcoming these limitations by scaling-up components in current processors is not feasible [62], requiring different solutions [34].

This paper exploits the property that sparse tensor methods can be *decomposed* into three stages: (i) tensor traversal, (ii) merging, and (iii) computation. Tensor traversal and merging operations are generally implemented as a deep nested-loop structure that can be expressed as a dataflow formulation [26]. Hence, we propose to offload tensor traversal and merging to a dedicated near-core dataflow engine, called the *Tensor Marshaling Unit* (TMU), designed to marshal data into the core, which performs the computation.

The TMU can be programmed to perform traversal and merging operations of arbitrarily-complex tensor algebra expressions. Moreover, the TMU enables parallel data loading, exposing additional memory-level parallelism. In this way, a vector-friendly layout can be obtained naturally, i.e., elements loaded in parallel can be packed contiguously into vector operands; thereby marshaling data into the core that can be computed efficiently using Single-Instruction Multiple-Data (SIMD) instructions. The TMU also allows decoupling data loading/merging from computation, permitting both the TMU and the compute core to operate in parallel in a pipelined fashion. In contrast to standalone accelerators, the TMU leverages existing core components such as functional units and caches to compute and store partial results; providing additional flexibility to customize computation, accumulation, and writing of partial results.

This paper makes the following contributions:

- We propose the TMU, a specialized near-core dataflow engine to accelerate critical tensor traversals and merging operations through tensor format and tensor algebra complete primitives.
- The TMU seamlessly enables to load multiple tensors or a single tensor in parallel, to perform merging, and to marshal data in a vector-friendly layout for efficient decoupled core computation. No solution in the literature provides such properties.
- We evaluate the TMU using detailed multicore simulations on a wide range of tensor workloads processing large inputs. The TMU achieves 3.6×, 2.8×, and 4.9× speedup over vectorized software implementations of memory-intensive, compute-intensive, and merge-intensive workloads, respectively. Moreover, we prototype the TMU in RTL showing that it is cost-effective, adding 1.52% area per core.

2 BACKGROUND

We consider an *order-n* tensor an n-dimensional collection of data. A_{ijk} is the scalar element at position i, j, k of an order-3 tensor A. Einsum expressions leverage this index notation to efficiently describe tensor computation [31].

2.1 Einsum expressions

Einsum expressions describe tensor computation by specifying how input/output dimensions relate: (i) input dimensions with repeated indexes are combined *element-wise*, (ii) input dimensions whose index is omitted in the output tensor are *contracted*, and (iii) other dimensions are just copied to the output. For instance, Matrix-vector multiplication is written as $Z_i = A_{ij}B_j$. As dimension j does not appear in the output tensor, it needs to be contracted (i.e., multiplied element-wise and summed up). Similarly, matrix multiplication would be $Z_{ij} = A_{ik}B_{kj}$, with a contraction on k. Matrix addition, instead, would be $Z_{ij} = A_{ij} + B_{ij}$, meaning all elements of A and B are summed element-wise.

These expressions are implemented in software as a deep loop hierarchy, each loop traversing and combining tensor *fiber*s, which are one-dimensional views of a tensor (e.g., matrix rows/columns) [52]. The loop order, instead, is defined by an index *schedule* [31]. For instance, an inner-product implementation of matrix multiplication has a schedule set to ijk, whereas outer-product to kij, and dataflow to ikj [20]. However, the iteration boundaries of these loops depends on the format input tensors are stored with [13].

2.2 Tensor compression formats

Dense tensors store their fibers contiguously according to a given data layout (e.g., row or column major). In contrast, sparse tensors use compressed formats to only store non-zero values (*nnzs*) and their position. Figure 1 summarizes a number of common compression formats for matrices [13]. The Coordinate format (COO), in Figure 1a, explicitly stores the non-zero positions as n-dimensional coordinates, which are typically sorted by some multidimensional ordering (e.g., row/column major). For matrices with $\#nnzs > \#rows + 1$, the Compressed Sparse Row (CSR) format [6] (Figure 1b) improves COO storage by replacing the `row_idxs` with an array, the `row_ptrs`, that stores the starting position of each row into the value array. For matrices with $\#rows > 2 \times \#nonempty\_rows$, the Doubly-Compressed Sparse Row (DCSR) format [12] (Figure 1c) improves CSR storage by compressing empty rows of the `row_ptrs` array. Higher-dimensional tensors, instead, are generally stored as COO or in Compressed Sparse Fiber (CSF) [57], which is a generalization of DCSR to multiple dimensions, thereby compressing all tensor fibers.

Chou et al. [13] formalize a hierarchical abstraction to express these and other tensor formats using six *level formats*. For instance, with this abstraction, CSR can be defined by combining a *dense* and a *compressed* level, whereas DCSR requires two compressed levels. Instead, COO can be defined as a set of *singleton* levels, one for each tensor dimension, and CSF as a hierarchy of compressed levels. This way, one can build arbitrarily-complex formats to optimize performance and storage efficiency of a given algorithm [31].

2.3 Fiber traversal

Each compressed level format can be *traversed* (i.e. iterated and loaded) with a specific *level functions* such as:

(1) Dense traversal:

```
for(idx=0; idx<fbrSize; idx++)
  val = vals[idx];
```

(2) Compressed traversal:

```
for(p=ptr[i]; p<ptr[i+1]; p++)
  idx = idxs[p];
  val = vals[p];
```

(3) Coordinate singleton traversal:

```
for(p=0; p<numNnzs; p++)
  idx0 = idxs0[p];
  idx1 = idxs1[p];
  val = vals[p];
```

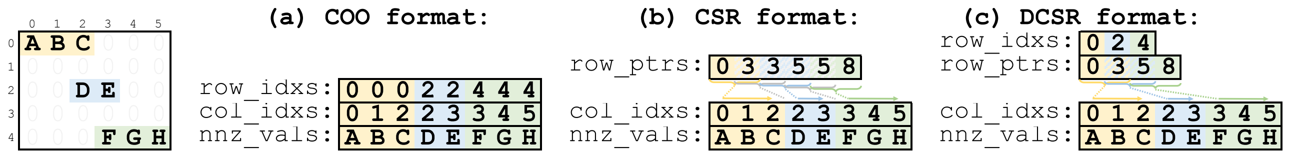

(a) COO format:
row_idxs: 0 0 0 2 2 4 4 4
col_idxs: 0 1 2 2 3 3 4 5
nnz_vals: A B C D E F G H
(b) CSR format:
row_ptrs: 0 3 3 5 5 8
col_idxs: 0 1 2 2 3 3 4 5
nnz_vals: A B C D E F G H
(c) DCSR format:
row_idxs: 0 2 4
row_ptrs: 0 3 5 8
col_idxs: 0 1 2 2 3 3 4 5
nnz_vals: A B C D E F G H

Figure 1: A sparse matrix and its representation in three different tensor compression formats.

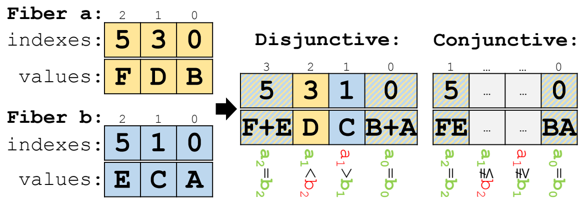


Figure 2: Disjunctive and conjunctive merging of two fibers.

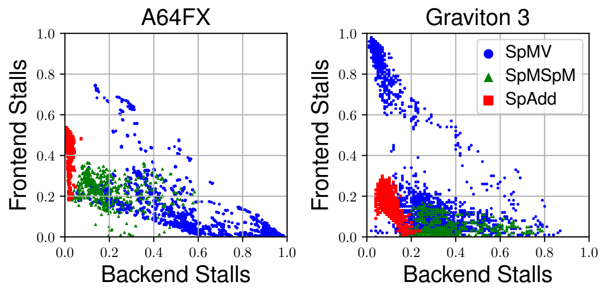


Figure 3: Normalized cycles spent stalling.

Hence, tensor traversal is implemented by nesting these functions into a deep loop hierarchy. To perform computation, multiple tensors are co-iterated and combined together, usually at the fiber level.

2.4 Fiber merging

While multiple dense fibers can be co-iterated with a simple for-loop structure, co-iterating compressed (or dense and compressed) fibers requires a *merging* operation [26]. As shown in Figure 2, *disjunctive* merging co-iterates multiple fibers and, for each iteration, assuming sorted coordinates, outputs and steps the fibers with the minimum coordinate [27], summing up elements with same indexes. *Conjunctive* merging works similarly but only outputs a value if all the output fibers have the same coordinate [24], multiplying their values. For instance, adding two CSR matrices requires disjunctive merging to join fibers (as $0 + x = x$), whereas element-wise multiplication requires conjunctive merging to intersect fibers (as $0 \cdot x = 0$) [31]. These operations are typically implemented with `while` loops and `if-then-else` constructs [31]. In practice, however, the intersection of a compressed and a dense fiber (see SpMV CSR in Figure 4) is usually implemented as a *scan-and-lookup* operation with constant complexity.

2.5 Computation

After merging, fiber elements are computed and written back to memory (e.g., lines 7 and 9 in Figure 4)). If the output is dense, data can be stored right away. However, if the output is compressed, the algorithm is generally implemented in two steps: a symbolic phase, which computes (or estimates) the size of the output data structure, and a numeric phase, which performs actual floating-point computation and writing [20]. In this context, another fundamental operation is tensor *reduction*, where different fibers are accumulated into a single one [35, 40]. Specifically, given a stream of coordinates-value pairs with possibly repeated coordinates, a reduction operation outputs a stream of pairs with unique coordinates where input pairs with equal coordinates have values accumulated.

3 MOTIVATION

As discussed in Section 2, tensor expressions are composed of fiber traversals, merging, and computation. To study the performance impact of each of these stages, we select three tensor kernels as representative proxies of each stage:

- Sparse Matrix-Vector Multiplication (SpMV) [64], which multiplies a CSR matrix and a dense vector and outputs a dense vector ($Z_i = A_{ij}B_j$). Since co-iteration of A_{*j} and B_j is generally implemented as a memory-intensive scan-and-lookup operation, SpMV has a tight loop structure, whose control flow introduces non-negligible overhead for highly-sparse matrices. Hence, we consider SpMV a representative proxy for the traversal stage.
- Gustavson's Sparse Matrix-Sparse Matrix Multiplication (SpMSpM) [20], which multiplies and outputs CSR matrices ($Z_{ij} = A_{ik}B_{kj}$ with (ikj) schedule). SpMSpM performs the same scan-and-lookup operation as SpMV but, instead of scalars, it looks up and loads whole rows of B, which are then reduced (accumulated) into a single row. For this reason, SpMSpM is a good proxy to study: (i) scan-and-lookup scaling to higher spatial locality and (ii) computation performance.
- Sparse Matrix Addition (SpAdd) [27], which adds and outputs CSR matrices ($Z_{ij} = A_{ij} + B_{ij}$) by repeatedly co-iterating and joining (disjunctive merge) matrix fibers with the same row-index i. Therefore, SpAdd is representative of merge-intensive kernels.

We profile these kernels both on an HPC processor, the Fujitsu A64FX [50], and a data-center processor, the AWS Graviton 3. While the A64FX has more memory bandwidth per core (1 TB/s for 48 cores vs. 300GB/s for 64 cores), Gravion 3 has cores with more out-of-order resources and larger caches. Figure 3 reports the portion of frontend (fetch) and backend (memory) cycle stalls (with respect to total cycles) when running the largest 100 matrices from the SuiteSparse Matrix Collection [16] on the aforementioned kernels.

Overall, we observe that sparse algebraic workloads have low CPU utilization [17, 18], as a large portion of cycles is spent either in frontend or backend stalls [60, 61]. In particular, we find that:

(1) The scan-and-lookup operation in SpMV performs better (fewer backend stalls) in architectures with larger caches, such as the Graviton 3, despite the lower per-core bandwidth. In fact, caches help filter long-latency memory accesses that saturate the limited ROB and MSHR resources of today's processors [41, 64].
(2) Despite better backend utilization, SpMV on Graviton 3 still suffers from high frontend stalls coming from compressed-fiber traversal. Data-dependent branching code is hard to predict and generates costly pipeline flushes [28].
(3) SpMSpM presents better backend utilization on the A64FX, which is also better with respect to SpMV. In contrast, on Graviton 3, SpMSpM has comparable backend utilization to SpMV. This suggests that (i) scan-and-lookup operations with higher spatial locality perform better in high-bandwidth architectures, and (ii) reduction operations do not drastically impact backend utilization since partial results generally fit in cache [40]. Computing these results, however, has better frontend utilization on more aggressive cores, like the ones in Graviton 3.
(4) SpAdd has low frontend utilization due to data-dependent branching code, especially for cores with limited out-of-order capabilities like the ones in the A64FX [27].

Hence, these results show that (i) current general-purpose processors generally have enough compute resources for the typical operational intensity of sparse algebraic workloads and (ii) existing cache hierarchies are quite efficient at capturing input locality (e.g., graph communities) and storing partial results. However, (i) data-dependent control flow from fiber traversal and merging [61] and (ii) irregular memory accesses from scan-and-lookup operations [60] prevent efficient system utilization, achieving just a fraction of the available peak performance [17, 18].

Scaling-up components in current processors is not enough to overcome these limitations [62]. Moreover, because of Moore's law coming short, we cannot expect significant improvements in the next years, requiring a more disruptive solution [34]. To this end, the following section introduces the TMU, which enables offloading and accelerating costly traversal and merging operations.

```
 1 void spmv(v_ty* x, const csr_ty* a, const v_ty* b) {
 2   v_ty sum = 0;
 3   for (i_ty i = 0; i < a->num_rows; i++) {
 4     p_ty p_beg = a->ptrs[i], p_end = a->ptrs[i+1];
 5     for (p_ty p = p_beg; p < p_end; p++) {
 6       v_ty vec_val = b[a->idxs[p]];
 7       v_ty nnz_val = a->vals[p];
 8       sum += vec_val * nnz_val; //inner-loop body (ri)
 9     }
10     x[i] = sum; sum = 0; //inner-loop tail (re)
11   }
12 }
```

Figure 4: SpMV decomposition in fiber traversals and computation. The outer-loop (lines 3-4 in yellow) traverses the dense CSR fiber of row pointers. The inner-loop traverses the compressed CSR fiber and dense vector with a scan-and-lookup operation (lines 5-7 in blue). Compute happens at the inner-loop body (line 8) and tail (line 10 in green.

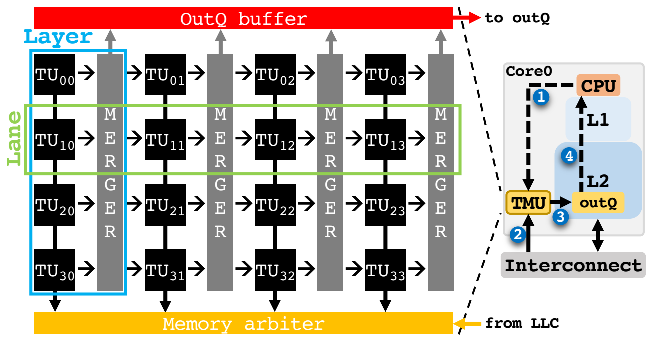


Figure 5: TMU overview, placement and operation.

4 TMU SYSTEM OVERVIEW

We leverage the format abstraction described in Section 2 to express tensor operations as *dataflow* programs composed of traversal, merging, and computation phases [26]. With this dataflow abstraction, we offload tensor traversal and merging to the TMU, while keeping computation into the core. Figure 4 shows this decomposition for SpMV.

As shown in Figure 5, the TMU can be ① programmed to ② traverse and merge tensor operands and ③ marshal the aggregated data to the core via a memory-mapped output queue (outQ) that the core can ④ process. TMU traversal, merging, and marshaling is programmed with dataflow primitives whereas the compute code is wrapped in callback functions the TMU triggers on traversal/merging events.

Traversal units, lanes and layers: As shown in Figure 5, the TMU is a matrix of Traversal Units (TUs) where each TU implements logic and storage to traverse a tensor fiber, that is, each of the `for` loops in Figure 4 (lines 3-4 and 5-7). Each of these `for` loops can be mapped to TUs of a TMU column, named *layer*. The `for` loop in lines 3-4 would map to the left-most layer, producing indexes that flow into the second layer, where the `for` loop in lines 5-7 would be mapped. Therefore, the TMU has a dataflow design that flows rightward. In addition, loop iterations can be parallelized using multiple rows, named *lanes*, which can also be used to load and merge multiple tensors.

Multi-lane parallelism: The values loaded by multiple lanes, either for parallel traversal or merging, are marshaled into vector operands. For example, each lane of the second layer that is traversing the `for` loop in lines 5-7 of Figure 4 loads one *vec_val* and one *nnz_val*, which are then marshaled into vector operands.

Callbacks and outQ processing: These operands are computed by the host core with the callbacks in Figure 6. These callbacks wrap the compute code of the body (*ri*) and tail (*re*) regions of the inner loop shown in Figure 4. The *ri* callback accumulates partial-results (into the `sum` variable) and is triggered by the TMU at every row iteration (inner-loop body). The *re* callback stores results at the end of every row traversal (inner-loop tail). To decouple TMU and core execution, the TMU sends to the core, through the outQ, a control/data stream defining the ordered sequence of callbacks and operands that the core should compute.

Table 1: TU traversal primitives. For each signature, `int` arguments are constant values and `stream` arguments are the values loaded by the parent loop, as explained later in this section.

Type	Description	Signature	Traversal
`Dense`	dense (or singleton) fiber scan	`DnsFbrT(int beg, int end, int stride=1)`	for(i=beg; i<end; i+=stride)
`Range`	compressed fiber lookup and scan	`RngFbrT(stream beg, stream end, int offset=0, int stride=1)`	for(i=beg.head()+offset; i<end.head(); i+=stride)
`Index`	dense fiber lookup and scan	`IdxFbrT(stream beg, int size, int offset=0, int stride=1)`	for(i=beg.head()+offset; i<beg.head()+size; i+=stride)

Table 2: Data streams for a TU.

Type	Description	Function
`mem`	loads `p`'s elements from memory at index `x`	`p[x]`
`ite`	generates TU iteration indexes ∈[beg,end)	`x:b->e`
`lin`	implements a linear transformation on `x`	`ax+b`
`map`	reads `x`-th element of a small map `a={v1,...,v16}`	`a[x]`
`ldr`	computes the address of the `x`-th element in `p`	`&p[x]`
`fwd`	forwards a leftward TU stream to the next layer	
`msk`	outputs a layer predicate for merging and lockstep operations	

Table 3: Inter-layer configurations.

Type	Description
`Single`	iterates a single lane
`BCast`	broadcasts a single lane to a parallel group
`Keep`	keeps one lane out of a parallel group
`DisjMrg`	joins lanes of a layer
`ConjMrg`	intersects lanes of a layer
`LockStep`	co-iterates lanes of a layer

4.1 Fiber traversal

As described in Section 2.3, all sparse and dense fibers traversals have the following `for` loop structure:

```
for(i=beg; i<end; i+=stride)
```

Each TU provides the logic and storage to iterate such a loop, with `beg` and `end` defined by the fiber traversal, and load fiber values into streams.

Fiber iteration: Table 1 shows the primitives we can use to program TUs to implement the traversals in Section 2.3. For instance, we can use a `Dense` primitive to implement the outer loop of SpMV in Figure 4, to iterate through matrix rows and load the row pointers. These pointers, loaded into streams, are then used to implement the inner-loop traversal, with a `Range` primitive, to load non-zero values and column indexes (compressed levels), which, in turn, are used to load dense vector values through scan-and-lookup. SpMM ($Z_{ij} = A_{ik}B_{kj}$), instead, works similarly but requires an additional inner loop, implemented with an `Indirect` primitive, to scan entire rows of the right-hand side matrix (B_k).

Data loading: Once the iteration space of a TU is defined, data loading is implemented through the *data streams* listed in Table 2. The `mem` stream loads data from a base address and a stream of indexes. To generate the index stream, TUs push their current iteration index (i.e., loop induction variable) into an `ite` stream, which can also be transformed with `lin` or `map` streams. Indirect accesses can be implemented by chaining `mem` streams. Hence, we can represent a fiber traversal with a *parent-children* dependency tree where the `ite` stream is the *root*. For instance, the SpMV scan-and-lookup operation in Figure 4 is implemented by instantiating two `mem` streams: a parent stream that loads the column indexes which are then used by its child stream to index into the dense vector.

4.2 Fiber merging and parallelization

We can use multiple TMU lanes to parallelize loop traversal or merge different tensors. Parallelization and merging can happen at any tensor level by setting TMU layers to one of the configurations shown in Table 3.

Parallelization is achieved by parallelizing loop iterations by using different lanes. Overall, parallelization allows (1) parallel loading, reducing the control overhead of traversing compressed data structures, and (2) parallel marshaling, enabling efficient vectorized compute in the host core. The bi-dimensional TMU design allows to parallelize loops at different levels. For instance, inner-loop parallelization allows to marshal adjacent tensor elements in the same vector operand (i.e. usual vectorization scheme). Outer-loop parallelization, instead, allows to marshal multiple fibers, slices, and arbitrary tensor dimensions in the same vector operand, enabling higher-dimensional parallelization schemes. In the SpMV example in Figure 4, where we parallelize at the inner-loop level, we should configure the first layer to load row pointers and `BCast` them to the next layer to load multiple elements of the same row in `LockStep` (with each lane loading at a different offset). To parallelize SpMV at the outer-loop level, concurrently loading multiple rows into different lanes, both layers should work in `LockStep`. Both parallization schemes aggregate data into vector operands that are marshaled into the core for computation. Boundary conditions can be handled by padding or marshaling the `msk` stream along with the operands, indicating the active elements in a multi-hot bit vector.

Fiber merging is achieved by loading different tensors in different lanes and merging them hierarchically with the mergers placed between layers. Merging operations are implemented by "sorting" the fiber indexes of all the active lanes in the layers, which is achieved by iteratively pulling fibers with minimum indexes. The position of these fibers can be identified with a multi-hot l-bit predicate, which is pushed in the `msk` stream of the layer. For the 2-lane merging example in Figure 2, the `msk` stream is `11, 01, 10, 11`, with first bit representing fiber A, second bit fiber B. These predicates are then used to aggregate vector operands to send to the core. In this way, for instance, we can implement SpKAdd [27], a summation of K sparse matrices, by mapping matrices in K different lanes and merging them with `DisjMrg` layers. Figure 7 shows the compute code of SpKAdd: the TMU traverses all the K matrices row by row

```
ri_callback(vfloat nnz_vals, vfloat vec_vals){
 vfloat mul_vals = vec_mul(nnz_vals, vec_vals);
 sum += vec_reduce(mul_vals);
}
re_callback(){
 x[i++] = sum; sum = 0;
}
```

Figure 6: SpMV callbacks.

```
ri_callback(vfloat nnz_els){
 *out_ptr++ = vec_reduce(nnz_els);
}
```

Figure 7: SpKAdd callback.

and, for each row, sends to the core all non-zero values with the same index, which are reduced with a vector operation. Running SpKAdd on the two rows in Figure 2 would produce the following vector operand stream: `AB C D EF`, with `AB` and `EF` being reduced in the core. If matrices are in CSR format, only the second compressed dimension requires merging. In contrast, if matrices are in DCSR format, both dimensions are compressed and both need to be merged. In this latter case, the merge happens hierarchically: only the active lanes from the first dimension, which are identified by the `msk` predicate of the first layer, are merged in the second layer.

4.3 Data marshaling and computation

Once we mapped the loop structure of a tensor expression into TMU layers, the TMU streams the aggregated data to the core for computation. Core compute is enabled by wrapping into callback functions the compute code within the *head* (H), *body* (B), and *tail* (T) regions of traversal/merging loops. These callback functions have a unique callback ID and a list of scalar, vector, or predicate operands produced by the TMU. In the SpMV example in Figure 4, we wrap the code in the inner-loop body into a `ri` callback, which multiplies and accumulates the matrix and vector values provided by the TMU, and the code in the inner-loop tail into a `re` callback, to store the accumulated results.

Each layer of the TMU is programmed to trigger these callbacks upon traversal/merging events such as the *begin*, *iteration*, and *end* of a traversal or merging. In the SpMV example in Figure 4, we register the `ri` callback to the *iteration* of the inner-loop layer and the `re` callback to the *end* of the inner-loop layer. For the `ri` callback, we also register the list of operands consisting of matrix and vector values. Callback registration can be done with the following callback:

```
add_callback(event, callback_id, args_list)
```

While running, the TMU pushes the callback IDs and vector operands of each registered event into the current outQ chunk. When the chunk is full, the core starts reading callback IDs and executes the proper HBT callback to compute the data operands. Meanwhile, the TMU populates another outQ chunk, overlapping data loading and computation.

```
 1 //Load and broadcast CSR row pointers
 2 row_fbrt = DnsFbrT(0, num_rows);
 3 row_ptbs = row_fbrt.add_mem_str(a->ptrs);
 4 row_ptes = row_fbrt.add_mem_str(a->ptrs+1);
 5 row_grpt = BCast(row_fbrt);
 6 //Load two row elements (and vec vals) in lockstep
 7 //TUx1 have step=2, TU01 has offset=0, TU11 has offset=1
 8 col_fbrt0 = RngFbrT(row_ptbs, row_ptes, 0, 2);
 9 col_idxs0 = col_fbrt0.add_mem_str(a->idxs);
10 nnz_vals0 = col_fbrt0.add_mem_str(a->vals);
11 vec_vals0 = col_fbrt0.add_mem_str(b, col_idxs0);
12 col_fbrt1 = RngFbrT(row_ptbs, row_ptes, 1, 2);
13 col_idxs1 = col_fbrt1.add_mem_str(a->idxs);
14 nnz_vals1 = col_fbrt1.add_mem_str(a->vals);
15 vec_vals1 = col_fbrt1.add_mem_str(b, col_idxs1);
16 col_grpt = LockStep(col_fbrt0, col_fbrt1);
17 nnz_vals = col_grpt.add_vec_str(nnz_vals0, nnz_vals1);
18 vec_vals = col_grpt.add_vec_str(vec_vals0, vec_vals1);
19 //Set the row-end and row-ite callbacks and operands
20 col_grpt.add_callback(GITE, ri, {nnz_vals, vec_vals});
21 col_grpt.add_callback(GEND, re, {});
```

Figure 8: TMU code for SpMV column-parallel.

4.4 Programmability and generalization

Figure 8 shows the full TMU configuration code for the SpMV example in Figure 4. The top block defines the dense traversal (*DnsFbrT*) for the CSR matrix row pointers. The middle block (second layer) co-iterates in *LockStep* two columns in parallel (*RngFbrT*), marshaling vector operands to the core (lines 17 and 18). The bottom block registers the *callbacks* happening at each row iteration (*ri*), specifying the associated data, and at the end of a row (*re*). With the increasing adoption of Domain-Specific Languages (DSLs) [1, 26, 31, 44], TMU primitives and compute code can be generated by extending DSL compilers like Custard/SAM [26] for TMU primitives and TACO [31] for compute code. Alternatively, because of the advances in compiler analysis [54] and transformation [55] techniques for high-level synthesis, TMU primitives and compute code [33] could be automatically generated out of standard high-level languages (e.g. C/C++). However, we leave this as future work.

Table 4 has an extensive list of tensor algebra algorithms showing how they map to the TMU. The loop schedule encodes the different parallelization schemes by underlining the parallel loops generating vector operands. Additionally, for each loop we state: (i) traversal type, (ii) instantiated data streams, (iii) layer operation mode, and (iv) associated callbacks. To prove functional completeness of TMU primitives with respect to tensor algebra, we observe that our primitives implement all of the minimal functionalities for tensor algebra formulated by Hsu et al. [26].

5 TMU OPERATION

This section details the implementation and operation of TMU components, concluding with a step-by-step example.

5.1 Traversal Unit design

TUs implement the logic to (i) iterate tensor fibers, (ii) generate a binary control sequence to track the iteration status, and (iii) populate data streams. To iterate a tensor fiber, each TU implements a finite-state machine (FSM) looping through the `fbeg`, `fite`, and `fend` states. The `fbeg` state initializes the iteration boundaries, which can either be constant values or read from a leftward TU, stalling

Table 4: Mapping different sparse tensor algebra kernels to TMU hardware. Notation Px means the loop at level x is parallel.

Algorithm	Einsum expression	Sparse formats	Schedule (SIMD)	Traversal type			Data streams						Group Traversal						Callbacks		
				Dns	Idx	Rng	lin	map	mem	ldr	fwd	msk	Single	BCast	LockStep	CnjMrg	DsjMrg	Keep	H	B	T
SpMV P0	$Z_i = A_{ij}B_j$	A=CSR	$\underline{ij}$	i		j			ij						ij					j	
SpMV P1	$Z_i = A_{ij}B_j$	A=CSR	$i\underline{j}$	i		j			ij					i	j					j	j
SpMSpV	$Z_i = A_{ij}B_j$	A,B=CSR	ij	ij		j			ij					i		j				j	
SpMM P0	$Z_{ij} = A_{ik}B_{kj}$	A=CSR	$\underline{ikj}$	ij	j	k			ikj						ikj				j	j	
SpMM P1	$Z_{ij} = A_{ik}B_{kj}$	A=CSR	$i\underline{kj}$	ij	j	k			ikj					i	kj				j	j	k
SpMM P2	$Z_{ij} = A_{ik}B_{kj}$	A=CSR	$ik\underline{j}$	ij	j	k			ikj				i	k	j				j	j	
SpMSpM P0	$Z_{ij} = A_{ik}B_{kj}$	A,B,X=CSR	$\underline{ikj}$	i		kj			ikj						ikj					j	k
SpMSpM P2	$Z_{ij} = A_{ik}B_{kj}$	A,B,X=CSR	$ik\underline{j}$	i		kj			ikj				i	k	j				j	j	k
SpKAdd	$Z_{ij} = \sum_k A^k_{ij}$	A^k,X=DCSR	$\underline{kij}$	i		j			ij			j					ij			j	
PageRank	$Z_i = A_{ij}X_jY_i$	A=CSR	$i\underline{j}$	i		j			ij					i	j					j	j
TriangleCount	$c = L_{ik}L^T_{kj}L_{ij}$	L=CSR	ikj	i		kj			ikj		k				ik	j				j	
MTTKRP P1	$Z_{ij} = A_{ikl}B_{kj}C_{lj}$	A=COO	$i\underline{klj}$	ij	klj		i		$iklj$					i	klj				j	j	
MTTKRP P2	$Z_{ij} = A_{ikl}B_{kj}C_{lj}$	A=COO	$ikl\underline{j}$	$iklj$	j		i	kl	$iklj$	i			i	kl	j				kl	j	
SpTC	$Z_{ij} = A_{ikl}B_{lkj}$	A,B=CSF	$iklj$	i		jkl			$ijkl$					i		jkl		j		j	i
SpTTV	$Z_{ij} = A_{ijk}B_k$	A=CSF	$ij\underline{k}$	ik		jk			ijk					k	k					k	k
SpTTM	$Z_{ijk} = A_{ijl}B_{lk}$	A=CSF	$ijl\underline{k}$	ik	l	jl			$ijlk$					l	k					k	k

the current TU execution if the leftward TU has not produced new valid data yet. If the streams, which are implemented as circular queues, are not full, the `fite` state pushes (i) a `0` token into the binary control sequence, (ii) the current iteration index into the `ite` stream, and (iii) a new element into each other stream (which is generated according to the stream type). Finally, when the fiber traversal has no more elements to iterate, the `fend` state pushes a 1 token into the binary control sequence and goes back to the `fbeg` state.

All data streams within the same TU are of equal size and controlled simultaneously with a single push/pull command. Hence, the value of the i-th element of a queue is computed starting from the i-th element of its parent queue, as defined in Section 4.1.

5.2 Traversal Group design

A Traversal Group (TG) implements the logic to merge and co-iterate TUs in a layer. Similarly to TUs, TGs also implement an FSM to loop through gbeg, `gite`, and gend states, generating predicates and a control sequence used later on to trigger callbacks. TG FSMs compute their control sequence by combining (i) the predicate of the previous layer, if any, and (ii) the control sequences of all the TUs in the layer, implementing a hierarchical evaluation. In particular, we consider a lane to be *active* only if the corresponding bit of the predicate coming out of the previous layer is set to true. TGs only process a `gite` state if all the active lanes have valid data in their queue heads.

In case of disjunctive merging, the `gite` state (i) computes the output predicate by setting to 1 the active lanes with minimum indexes, (ii) consumes them, and (iii) pushes a `0` token into the binary control sequence. When all active TUs have no more elements to merge disjunctively, the gend state pushes a 1 token into the binary control sequence.

In case of conjunctive merging, the `gite` state (i) computes the output predicate setting to one of the active lanes with minimum indexes, (ii) consumes them, and (iii) pushes a `0` token into the binary control sequence only if all active lanes have minimum indexes (all-true predicate). When any active TU has no more elements to merge conjunctively, the gend state pushes a 1 token into the binary control sequence.

Finally, in case of lockstep co-iteration, the `gite` state (i) computes the output predicate setting to one of the active lanes not done iterating, (ii) consumes their heads, and (iii) pushes a `0` token into the binary control sequence. When all active TUs have no more elements to co-iterate, the gend state pushes a 1 token into the binary control sequence.

5.3 Output Queue construction

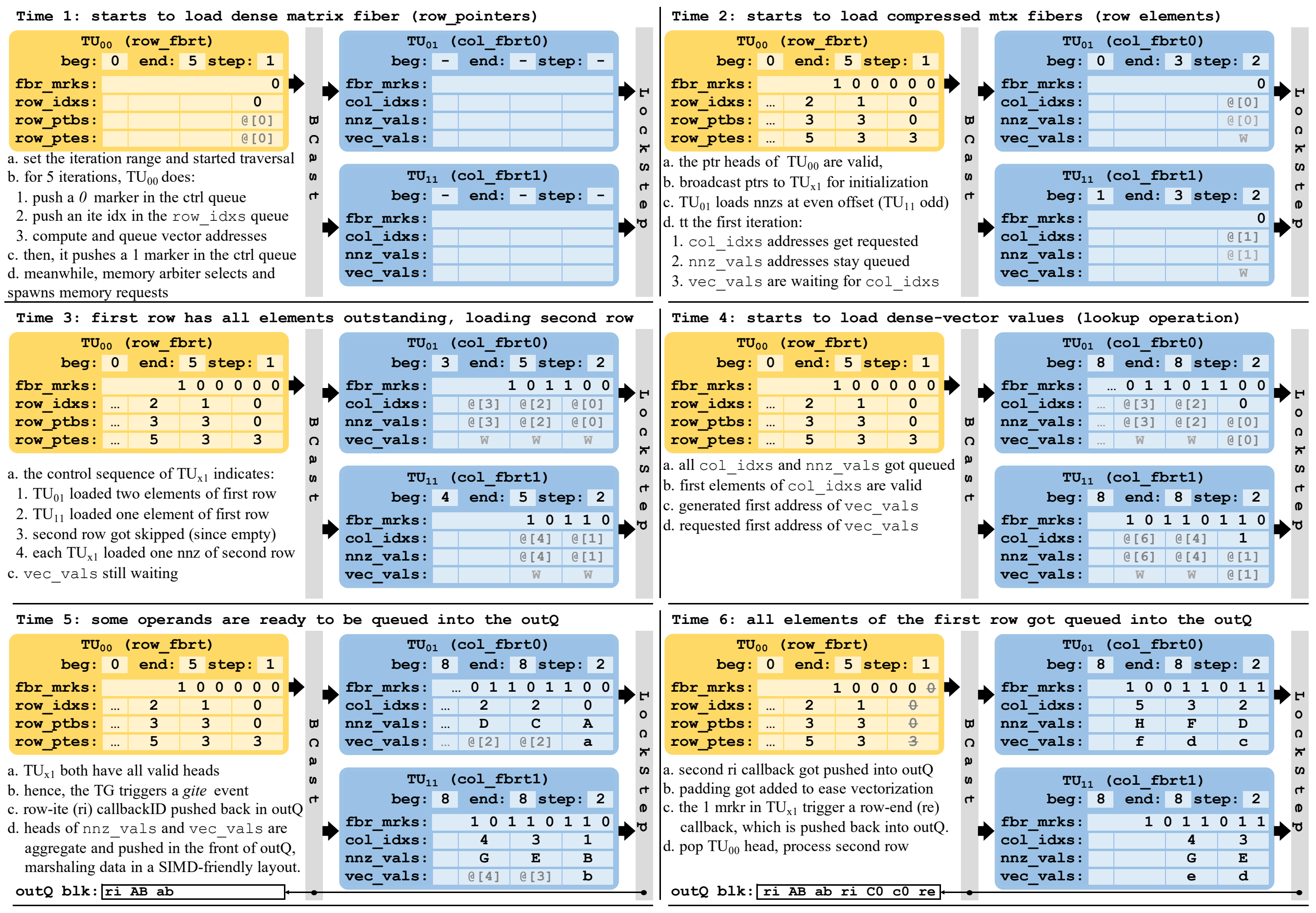
The predicates and control sequences generated from each TG are then used to push callback IDs and operands into the outQ. Similarly to traversal and merging, outQ construction is implemented as an FSM running in each TG. However, while traversal and merging phases can be fully decoupled (i.e., each TU/TG iteration can start as soon as it has valid inputs), outQ generation needs to be serialized across TGs to preserve the order in which callbacks and operands are processed by the core. Hence, besides the obeg, `oite`, and oend states, outQ generation also requires ow4p and ow4n states to signal a TG is waiting for the previous or next TG to push data into the outQ. When a TG is in a gbeg, `gite`, or gend state, it checks whether there is any callback associated to that state and, if so, it pushes its ID and operands into the outQ buffer. For performance reasons, both the outQ and outQ buffers are double-buffered.

5.4 Memory arbiter

The TMU sends out memory requests at the cacheline granularity. For each cycle, the TMU hierarchically selects the next cacheline address to request. Requests from the leftmost layers (outer loops) are prioritized. TUs within the same layer are selected round-robin. Streams within a TU are selected in configuration order. Requests within the same queue are selected in order.

5.5 TU queue sizing

All TUs of a layer instantiate, at configuration time, the same amount of streams with the same size. However, since nested loops are mapped from left to right, the rightmost layers load and merge


Time 1: starts to load dense matrix fiber (row_pointers)
Time 2: starts to load compressed mtx fibers (row elements)
Time 3: first row has all elements outstanding, loading second row
Time 4: starts to load dense-vector values (lookup operation)
Time 5: some operands are ready to be queued into the outQ
Time 6: all elements of the first row got queued into the outQ
TU00 (row_fbrt)
TU01 (col_fbrt0)
TU11 (col_fbrt1)
beg: end: step:
fbr_mrks:
row_idxs:
row_ptbs:
row_ptes:
col_idxs:
nnz_vals:
vec_vals:
BCast
Lockstep
outQ blk: ri AB ab
outQ blk: ri AB ab ri C0 c0 re

Time 1:
a. set the iteration range and started traversal
b. for 5 iterations, TU$_{00}$ does:
1. push a *0* marker in the ctrl queue
2. push an ite idx in the row_idxs queue
3. compute and queue vector addresses
c. then, it pushes a 1 marker in the ctrl queue
d. meanwhile, memory arbiter selects and spawns memory requests

Time 2:
a. the ptr heads of TU$_{00}$ are valid,
b. broadcast ptrs to TU$_{x1}$ for initialization
c. TU$_{01}$ loads nnzs at even offset (TU$_{11}$ odd)
d. tt the first iteration:
1. col_idxs addresses get requested
2. nnz_vals addresses stay queued
3. vec_vals are waiting for col_idxs

Time 3:
a. the control sequence of TU$_{x1}$ indicates:
1. TU$_{01}$ loaded two elements of first row
2. TU$_{11}$ loaded one element of first row
3. second row got skipped (since empty)
4. each TU$_{x1}$ loaded one nnz of second row
c. vec_vals still waiting

Time 4:
a. all col_idxs and nnz_vals got queued
b. first elements of col_idxs are valid
c. generated first address of vec_vals
d. requested first address of vec_vals

Time 5:
a. TU$_{x1}$ both have all valid heads
b. hence, the TG triggers a *gite* event
c. row-ite (ri) callbackID pushed back in outQ
d. heads of nnz_vals and vec_vals are aggregate and pushed in the front of outQ, marshaling data in a SIMD-friendly layout.

Time 6:
a. second ri callback got pushed into outQ
b. padding got added to ease vectorization
c. the 1 mrkr in TU$_{x1}$ trigger a row-end (re) callback, which is pushed back into outQ.
d. pop TU$_{00}$ head, process second row

Figure 9: Example performing SpMV with inner-loop vectorization (see Table 2, P1 version) over the first row of the CSR matrix in Figure 1.

more data than the leftmost ones, leading to different storage requirements. To provide flexibility, all TUs within a lane share the same storage and queues are allocated at configuration time using this shared per-lane storage. This permits shorter queues on leftmost TUs while making full utilization of the available storage, even if some TUs within a lane are not used at all. Queues are sized with an analytical model which allocates space to layers according to the amount of data to load, which can be statically estimated from the number of nnzs per fiber of the tensor.

5.6 System integration

Placement: As shown in Figure 5 each core in a multicore system features a TMU. The TMU traverses fibers by loading data from the LLC, and marshals the data into the outQ that is written into the core's private L2 cache. Data from fiber traversals is unlikely to experience reuse from private caches, and by reading from the LLC we take advantage of the larger MSHR count (enabling more MLP). Each outQ is core-private, therefore injecting it into the L2 cache enables faster compute throughput.

Memory subsystem integration: Like prior general-purpose processor engines, the TMU operates with virtual addresses and uses the host core's address translation hardware (MMU) [2, 32, 39, 66, 67]. In particular, it queries the L2 TLB, and if a page fault occurs, it interrupts the core so the OS can handle the page fault. Once the missing translation is available, the MMU signals the TMU to retry the memory access.

The TMU operates decoupled from the host core, issuing coherent read-only memory requests (fiber traversals) that do not affect coherence or consistency. TMU produces the outQ that is written (write-only) into the private L2 cache of the host core. While the outQ data may be evicted into shared cache levels, this data is not shared across cores. Therefore, there is no shared read-write data between the TMU and the host core or other cores in the system.
Context switching and exceptions: TMU architectural state must be saved and restored when a thread is context-switched. When the OS deschedules a thread that uses the TMU, it quiesces the TMU, saves its context, and restores it when the thread is rescheduled. The minimum context state that needs to be saved is the initial TMU configuration (e.g., queue types and sizes, beg and end iteration

boundaries), the head of each TU ite stream, and some control registers such as the base outQ address and current writing offset. The memory-mapped outQ is private per thread.

5.7 Step-by-step example

Figure 9 contains a step-by-step example for SpMV (Figure 4) where the TMU has been programmed using the code in Figure 8. The example loads the CSR matrix shown in Figure 1 and assumes a two-lane design.

6 EXPERIMENTAL METHODOLOGY

Simulation Infrastructure: We evaluate the TMU using the gem5 v20.1.0.0 simulator [37]. We simulate a multicore system consisting of 8 Neoverse-N1-like out-of-order cores, three levels of cache, 4 HBM2e memory channels, and a mesh-based network-on-chip modeling the AMBA 5 CHI protocol. Each core features a TMU engine that faithfully models the described architecture. The simulated system runs Ubuntu 20.04 with Linux kernel 5.4.65. Table 5 summarizes the architectural parameters.

Workloads: We evaluate our proposal over several sparse linear and tensor algebra kernels and real-world applications with different characteristics. We employ high-performance multi-threaded software baselines which we either generated by the well-known TACO compiler [31] or extracted from existing reference libraries [35, 47]. We vectorized all baselines with Arm SVE and utilized vector gather and scatter instructions where appropriate. In particular, we evaluate the following kernels from Table 4:

- **SpMV CSR** [64]: Software baseline from TACO vectorized with Arm SVE. The TMU implements inner-loop vectorization (P1).
- **SpMSpM CSR** [20]: Implements $Z = AA^T$. Software baseline from TACO vectorized with Arm SVE. The TMU implements inner-loop vectorization (P2).
- **SpKAdd DCSR** [27]: Software baseline from TACO (k=8) vectorized with Arm SVE. To preserve domain characteristics, input matrices (A^x with $x = 1 \ldots k$) are generated by cyclically distributing rows of the original matrix (A) to each input matrix (i.e., i-th row of input $A^x_i = A_{i*k+x}$).
- **MTTKRP COO** [47]: Performs a Matricized COO Tensor Times Khatri-Rao Product with the permutation optimization proposed by Phipps et al. [47]. The TMU implements parallelism at the mode (P1) and rank (P2) levels.
- **SpTC CSF** [35]: Performs a tensor contraction of two CSF tensors with the expressions evaluated by Liu et al. [35]. We evaluate the symbolic phase to limit simulation time.

Moreover, we evaluate the following real-world applications:

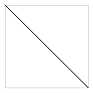
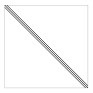
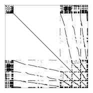
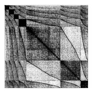
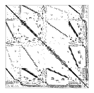
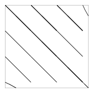
- **PR**: Performs the PageRank algorithm as implemented in the GAP Benchmark suite [7] (Jacobi-style method), optimized with Arm SVE.
- **TC**: Counts the triangles in a graph as in the fused GraphBLAS version [11] based on masked SpMSpM [38].
- **CP-ALS** [47]: Performs a canonical polyadic decomposition of a COO tensor with an alternating least squares algorithm, from the GenTen framework [46].

We run these applications with the settings suggested by the authors. Table 6 describes the inputs we used for all linear and tensor algebra applications, from SuiteSparse [16] and FROSTT [56], respectively.

Table 5: Simulated architectural parameters.

Cores	8 Neoverse-N1-like out-of-order cores at 2.4GHz
SVE width	512 bits
Reorder buffer	224 entries
Load/Store queues	96 entries, 96 entries
Private L1 I&D	64 KiB/core, 4-way, 2 cycle data access, 32 MSHRs
L1I prefetcher	Tagged
L1D prefetcher	Stride, degree 2
Private L2	512 KiB/core, 8-way, 8 cycle data access, 64 MSHRs
L2 prefetcher	Best Offset Prefetcher
Shared L3 (LLC)	Mostly exclusive, 8 slices of 1MiB, 16 ways, 12 cycles data access, 128 MSHRs
Coherence protocol	MOESI-like AMBA 5 CHI specification
Network topology	4 × 4 2D mesh, 1 cycle routers, 1 cycle links
Memory	4 HBM2e channels, FR-FCFS, 37.5GB/s per channel
TMU	2KB per-lane storage, 8 lanes, 4 TGs with mergers 128 outstanding requests

Table 6: Inputs

Id	**Matrix [16]**	**nnzs**	**rows**	**nnzs/row**	**Domain**
M1	af_0_k101	17.6M	504K	~35	structural
M2	atmosmodm	10.3M	1.5M	~7	fluid dynamic
M3	Freescale1	17.1M	3.4M	~5	circuit simulation
M4	gb_osm	13.3M	7.7M	~2	graph
M5	halfb	12.4M	225K	~55	structural
M6	test1	9.4M	393K	~24	semiconductor
Id	**Tensor [56]**	**nnzs**	**Dimensions**		**Domain**
T1	Chicago-crime	5M	6K x 24 x 77 x 32		count
T2	LBNL-network	2M	2K x 4K x 2K x 4K x 1M		network
T3	NIPS pubs	3M	3K x 3K x 14K x 17		text
T4	Uber pickups	3M	183 x 24 x 1K x 2K		map
M1	**M2**	**M3**	**M4**	**M5**	**M6**
					

RTL implementation and area analysis: We implement the TMU in System Verilog using the parameters listed in Table 5 and synthesize the design using the GlobalFoundries 22nm FD-SOI technology node. We have used the Cadence Genus tool v19.11 for the logical synthesis and the Cadence Innovus tool v19.11 for the place-and-route. The area cost of the TMU is 0.0704mm^2, where each lane occupies just 0.0080mm^2. The TMU requires 1.52% of the area of a Neoverse N1 core [45], the one modeled in our evaluation, when scaled to the same technology node [59].

7 EXPERIMENTAL RESULTS

In this section we first evaluate TMU performance with respect to software baselines. Then, we provide a sensitivity analysis on TMU storage and SVE vector length. Finally, we compare to existing single-lane engine solutions.

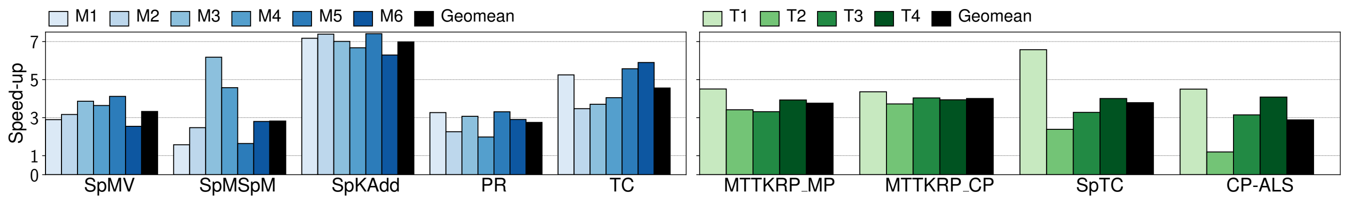


Figure 10: TMU speedups for linear (left) and tensor (right) algebra workloads with respect to baseline.

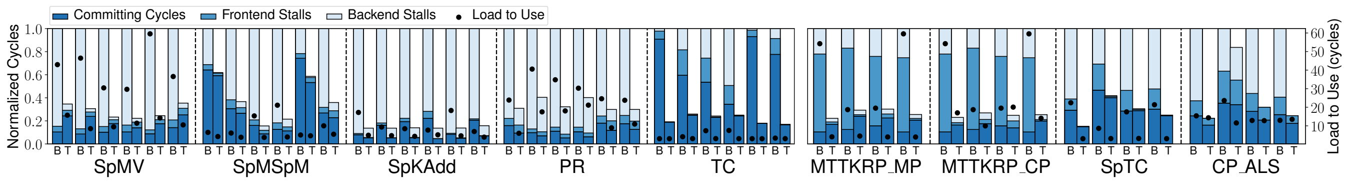
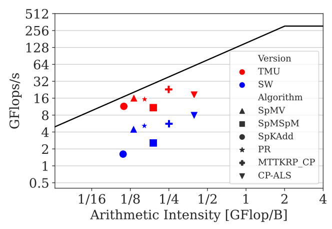
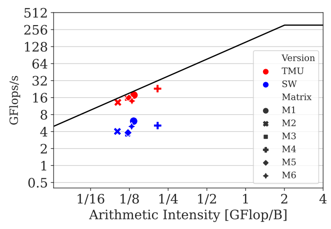


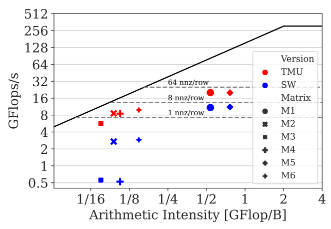
Figure 11: Normalized cycles breakdown: (i) committing, (ii) frontend stalls, and (iii) backend stalls for TMU (T) and baseline (B). The dots indicate average load-to-use latency (right y-axis).



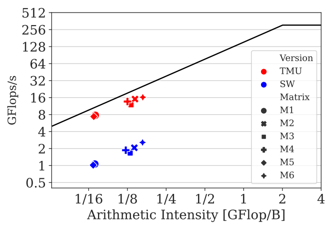
(a) Geomean



(b) SpMV



(c) SpMSpM



(d) SpKAdd

Figure 12: Roofline models for: (a) evaluated workloads (geomean inputs), (b) SpMV, (c) SpMSpM, and (c) SpKAdd

7.1 Performance analysis

Figure 10 reports the performance speedups achieved by the TMU over the software baselines detailed in Section 6. Overall, the proposed system achieves geomean performance speedups of 3.58×, 2.82×, and 4.94× on memory-intensive (SpMV, PR, MTTKRP, CP-ALS), compute-intensive (SpMSpM), and merge-intensive (SpKAdd, TC, SpTC) workloads, respectively. Figure 11 shows, for each evaluated workload and input, the normalized cycles spent (left y-axis) (i) committing at least one instruction, (ii) on frontend (fetch) stalls, or (iii) on backend (commit) stalls. While on the right y-axis, we show average load-to-use latency as seen by the cores.

SpMV and PR present similar performance regimes with geomean speedups of 3.32× and 2.74×, respectively. PR attains slightly lower performance since the weight update is not accelerated by the TMU. Figure 11 shows the TMU drastically reduces backend stalls, the main bottleneck of these memory-intensive workloads. In addition, we observe a sharp reduction in load-to-use latency, e.g., from 67 to 23 cycles for M1 in SpMV. A similar behavior can be observed on the memory-intensive tensor workloads (MTTKRP_MP, MTTKRP_CP, and CP-ALS), which obtain geomean speedups of 3.76×, 4.01×, and 2.88×, respectively. Overall, we observe that the TMU effectively hides long-latency misses on memory-intensive workloads.

A TMU-enabled system achieves from 1.58× to 6.18× speedups on SpMSpM (2.82× geomean). SpMSpM is a compute-intensive workload, as shown by the large portion of committing cycles in Figure 11. However, the time spent on compute is input dependent, i.e., matrices with a higher nnzs per row count perform more compute. For input matrices with a larger portion of committing cycles (i.e., M1 and M5), the TMU is unable to provide large performance speedups due to Amdhal's law but still reduces backend stalls to a minimum nonetheless. In contrast, for matrices that have a lower portion of committing cycles (i.e., M3 and M4) the TMU delivers 6.18× and 4.58× speedup, respectively.

On merge-intensive workloads such as SpKAdd, TC, and SpTC, the TMU achieves 6.98×, 4.56×, and 3.79×, respectively. SpKAdd presents significant speedups due to two main reasons. First, the TMU enables parallel loading of all eight matrices in a dataflow manner, unlocking memory level parallelism that significantly reduces backend stalls. Second, merging is performed efficiently by the TMU, almost eliminating frontend stalls and reducing committing cycles (4× reduction on M4), since the core has less work to do. TC relies heavily on merging, experiencing a large portion of frontend stalls and committing cycles. In this case, the TMU is again able to reduce almost completely frontend stalls and drastically reduce the amount of compute to perform by the core related to merging operations, greatly reducing committing cycles. A similar

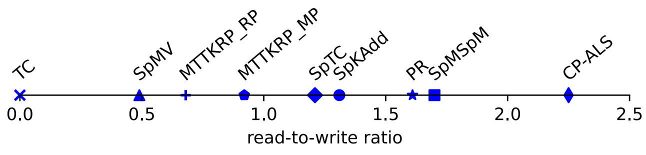


Figure 13: Read-to-write ratio for evaluated workloads.

behaviour can be observed in SpTC. Overall, workloads that perform merging operations greatly benefit from the TMU's ability to co-iterate multiple fibers, merge them in place, and produce a SIMD-friendly outQ for the core to process.

System utilization: To explain the improvements the TMU brings in terms of system utilization, we present Roofline models [65] in Figure 12. Roofline models relate arithmetic intensity (AI) (x-axis), measured by the ratio of total floating-point operations to total data movement, and performance in giga-flops (y-axis). The diagonal line draws the peak memory bandwidth, while the horizontal line is the processor's peak performance.

Figure 12a shows the data points for the evaluated workloads with the geomean for all inputs, except TC (integer arithmetic) and SpTC (symbolic phase) that do not perform floating-point computation. As can be seen, baseline SVE software versions are unable to use a substantial amount of memory bandwidth, attaining poor system utilization and compute throughput. TMU-accelerated workloads are close to the memory bandwidth roof, and SpMV is almost saturating peak bandwidth, achieving significantly higher compute throughput. SpMSpM is unable to use as much bandwidth as the other workloads due to its compute-intensive nature.

The rest of the Roofline models focus on SpMV, SpMSpM, and SpKAdd, showing data for all inputs. For SpMV (Figure 12b), which is memory-intensive, the software version is not able to use the available memory bandwidth due to data-dependencies on long-latency memory accesses, quickly filling out-of-order core resources, stalling its execution. The dataflow design of the TMU unlocks additional MLP for offloaded traversals, increasing memory bandwidth utilization by almost 4×, leading to much better system-level performance. A similar behavior can be observed in SpKAdd for the software baseline versions (Figure 12d), where the additional pressure that merging puts on core resources leads to even lower performance. The TMU again enables almost close to peak bandwidth utilization, which combined with its merging support significantly boosts overall performance.

Finally, for SpMSpM we also plot three additional dashed ceilings (Figure 12c). We compute these ceilings by processing synthetic matrices, each one storing a fixed number of non-zeros per row ($n = \{1, 8, 64\}$) located at column indexes $\{0..n-1\}$, proxying performance on ideal spatio-temporal locality. The data points starting from the right correspond to inputs M5 and M1, respectively. These two matrices have higher nnzs per row values (see Table 6) and perform better on the software baseline with respect to other inputs due to higher AI. TMU-accelerated executions improve performance and bandwidth utilization, but the former is constrained by the inherent *nnzs per row* compute ceilings.

The core and the TMU work in a decoupled pipelined manner. Therefore, we introduce the *read-to-write ratio*, which is defined as the ratio between the time it takes the core to process (read) an outQ block versus the time it takes the TMU to produce (write) it, averaged over all blocks. Figure 13 shows this ratio for the evaluated workloads. A value below one implies the core is faster than the engine. We observe this happens for TC, since merging is offloaded to the TMU, and for SpMV and MTTKRP, where the compute is very regular due to the SIMD-friendly data layout produced by the TMU. SpKAdd and SpTC have ratios close one, which indicates a balanced execution. SpMSpM, PR and CP-ALS have higher ratios due to more and less regular compute, indicating that the bottleneck is on the core's side.

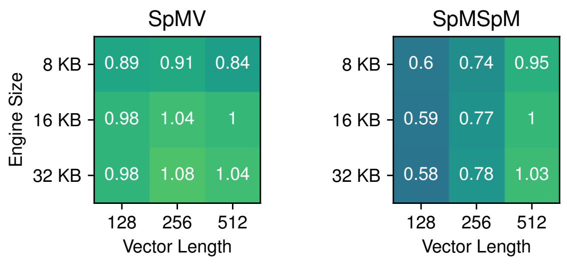


Figure 14: Design space exploration varying engine storage size and SVE vector length.

Summary: Our evaluation demonstrates that the TMU successfully accelerates tensor traversals and merging operations, thereby improving overall system utilization. Moreover, performing computation on the core enables additional flexibility to seamlessly express and accelerate workloads that implement kernel fusion (TC) and that require evaluating partial results (CP-ALS).

7.2 Sensitivity analysis

Figure 14 shows a sensitivity analysis varying the total engine size and SVE vector length, which is tied to the number of TMU lanes. That is, a system with 512-bit SVE has an 8-lane TMU design, while a system with 256-bit SVE has a 4-lane TMU design. The heatmap shows normalized speedup with respect to the evaluated 16KB 512-bit SVE configuration. We observe that SpMV is more sensitive to engine storage, as larger storage enables more MLP, hiding memory latency, which is the main bottleneck in SpMV. SVE length does not have a large impact, since SpMV has a read-to-write ratio of 0.5. In SpMSpM we observe the opposite behaviour: wider SVE length has a large performance impact, since the bottleneck is on the core side (1.68 read-to-write ratio).

7.3 Comparison with the state-of-the-art

Existing traversal engines integrated into multicore systems are conceived for graph analytics [39, 66]. For this reason, they feature architectures similar to a single TMU lane, but with fewer primitives (i.e., stream types) and no support for parallel data loading or merging, limiting their applicability to simple linear algebra kernels. Two prominent examples are HATS [39] and SpZip [66], the former exploits locality in graph communities and the latter implements de(compression) techniques. Both optimizations are orthogonal to the TMU operation. Therefore, we evaluate a *Single-Lane* TMU with the same storage as the multi-lane *TMU* employed throughout the

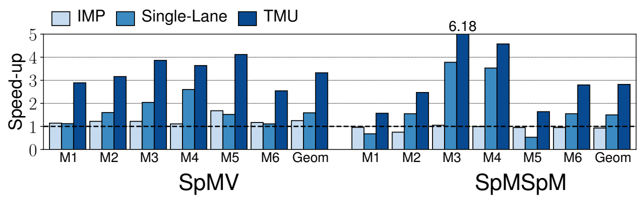


Figure 15: Speedup of IMP, Single-Lane and TMU with respect to baseline.

evaluation. Figure 15 shows that *Single-Lane* achieves 1.59× and 1.50× geomean speedup on SpMV and SpMSpM, while *TMU* gets 3.32× and 2.82×, respectively. This indicates that traversal engines for graph analytics would benefit from parallel data loading as well.

We also evaluate the Indirect Memory Prefetcher (IMP) [67], configured as recommended by the paper authors, including the use of virtual address to prefetch across memory page boundaries. IMP performs well on SpMV, with an average speedup of 1.25×, but fails to deliver performance improvements on SpMSpM due to thrashing of partial results. Overall, *Single-Lane* and *TMU* obtain better performance as they unlock more MLP which always fetches useful data.

8 RELATED WORK

As previously discussed, the two main performance bottlenecks of sparse tensor algebra workloads on multicore processors are (i) data-dependent control flow and (ii) long-latency memory accesses.

Prefetchers and near-memory accelerators: A number of programmable prefetchers have been proposed to hide the access latency of indirect accesses [3, 60, 67]. Similarly, data layout transformation mechanisms rearrange sparse data into a dense representation to improve locality and make use better the memory hierarchy [5, 36, 70]. However, all these proposals are limited in the access patterns they can handle and do not tackle the data-dependent control flow overheads.

Vector gather-scatter units: Some efforts to optimize vector units for non-unit stride memory accesses require memory systems that do not employ conventional cache-line based approaches, such as cache-less solutions [63] or single-word cache-lines found in Cray SV1 systems [10]. Tarantula [19] is a vector extension that gracefully integrates into a traditional memory hierarchy, providing high bandwidth for non-unit stride accesses and efficient support for gather/scatter instructions. Nevertheless, these proposals mainly focus on vectorizing scan-and-lookup operations at the inner-loop level, which only occur in a fraction of the tensor methods we discussed. Conversely, the TMU benefits from: (i) parallel traversal of the entire loop nest by leveraging layers; (ii) dataflow operation that always stays on the right execution path, avoiding costly miss-speculation events, and (iii) efficient merging in hardware without core intervention.

Traversal engines: HATS [39] and SpZip [66] are specialized fetchers that perform graph traversals. The former focuses on exploiting graph locality to reduce data movement, while the latter focuses on (de-)compression techniques to reduce memory traffic. These systems are conceived to work well on graph analytics and linear algebra workloads. Therefore, they do not have support for the primitives that are necessary for tensor algebra, nor implement parallel data loading, merging, and vectorized computation.

Table 7: Qualitative state-of-the-art comparison.

Features	Prog. prefetchers [3, 60, 67]	Gather-scatter [10, 19, 63]	Traversal engines [39, 66]	Fixed-function DSAs [43, 58, 69, 71]	Prog. DSAs [14, 24, 49]	TMU
Improves bandwidth usage	✓	✓	✓	✓	✓	✓
HW-accelerated traversal			✓	✓	✓	✓
HW-accelerated merging				✓	✓	✓
Tensor algebra complete	✓	✓			✓	✓
Storage format complete	✓	✓			✓	✓
General-purpose compute	✓	✓	✓			✓
SIMD-friendly compute		✓		✓	✓	✓

Domain-Specific Architectures (DSAs): Architectures such as GAMMA [69] and others [43, 58, 71] implement efficient fixed-function designs for sparse matrix-matrix multiplication, which is a fundamental kernel many tensor algebra operations can be reduced to. However, these solutions provide a low degree of flexibility and no opportunities for kernel fusion [9]. In contrast, architectures like Capstan [49] and others [14, 24] can be programmed to execute a wider set of Einsum expressions [26]. However, despite their wider applicability, we believe that (i) computation on specialized hardware is constrained to design choices such as a reduced set of operations and data types, and (ii) computing on specialized accelerators requires moving tensors in and out of the accelerator memory before and after computation. In real-world applications such as CP-ALS tensor decomposition [47], where partial results need to be evaluated at each iteration, this approach may not be convenient.

Summary: Therefore, we advocate that offloading critical operations such as tensor traversal and merging to specialized units while keeping tensor computation in the core provides a valuable trade-off between performance and flexibility. Moreover, by computing and storing partial results in the core, we leverage vector units and caches which provides more flexibility than DSAs for applications requiring complex control logic. By performing parallel tensor loading and marshaling, we can achieve better performance than near-core traversal engines in the literature. Table 7 summarizes a qualitative state-of-the-art comparison.

9 CONCLUSIONS

In this paper we propose the TMU, a programmable dataflow engine for multicore architectures that enables offloading costly traversal and merging operations of tensor algebra expressions. The TMU stands out from prior work by supporting all the necessary primitives to be tensor format and algebra complete and by proposing a novel multi-lane design to expose abundant memory level parallelism through parallel data loading, which naturally produces vector operands to be efficiently computed by the core. To achieve high performance, the TMU is decoupled from the computing core, hiding memory access latencies. Performing tensor computation into the core enables additional flexibility to support applications that require complex control logic. We show that a multicore system with TMU hardware achieves large performance gains due to better system utilization on a wide set of tensor algebra workloads.

ACKNOWLEDGMENTS

This work has been partially supported by the Spanish Ministry of Science and Innovation MCIN/AEI/10.13039/501100011033 (contract PID2019-107255GB-C21), the Generalitat of Catalunya (contract 2021-SGR-00763), the Arm-BSC Center of Excellence, the European HiPEAC Network of Excellence, and the European Processor Initiative (EPI), which is part of the European Union's Horizon 2020 research and innovation program under grant agreement No. 826647. M. Siracusa has been supported through an FI fellowship [2022FI_B 00969] and V. Soria-Pardos through an FPU fellowship [FPU20-02132]. A. Armejach is a Serra Hunter Fellow.

REFERENCES

[1] Martín Abadi, Ashish Agarwal, Paul Barham, Eugene Brevdo, Zhifeng Chen, Craig Citro, Greg S. Corrado, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Ian Goodfellow, Andrew Harp, Geoffrey Irving, Michael Isard, Yangqing Jia, Rafal Jozefowicz, Lukasz Kaiser, Manjunath Kudlur, Josh Levenberg, Dandelion Mané, Rajat Monga, Sherry Moore, Derek Murray, Chris Olah, Mike Schuster, Jonathon Shlens, Benoit Steiner, Ilya Sutskever, Kunal Talwar, Paul Tucker, Vincent Vanhoucke, Vijay Vasudevan, Fernanda Viégas, Oriol Vinyals, Pete Warden, Martin Wattenberg, Martin Wicke, Yuan Yu, and Xiaoqiang Zheng. 2015. TensorFlow: Large-Scale Machine Learning on Heterogeneous Systems. https://www.tensorflow.org/ Software available from tensorflow.org.

[2] Sam Ainsworth and Timothy M. Jones. 2016. Graph Prefetching Using Data Structure Knowledge. In *Proceedings of the 2016 International Conference on Supercomputing, ICS 2016, Istanbul, Turkey, June 1-3, 2016*, Ozcan Ozturk, Kemal Ebcioglu, Mahmut T. Kandemir, and Onur Mutlu (Eds.). ACM, 39:1–39:11. https://doi.org/10.1145/2925426.2926254

[3] Sam Ainsworth and Timothy M. Jones. 2018. An Event-Triggered Programmable Prefetcher for Irregular Workloads. *SIGPLAN Not.* 53, 2 (mar 2018), 578–592. https://doi.org/10.1145/3296957.3173189

[4] Ariful Azad and Aydin Buluç. 2019. LACC: A Linear-Algebraic Algorithm for Finding Connected Components in Distributed Memory. In *2019 IEEE International Parallel and Distributed Processing Symposium, IPDPS 2019, Rio de Janeiro, Brazil, May 20-24, 2019.* IEEE, 2–12. https://doi.org/10.1109/IPDPS.2019.00012

[5] Adrián Barredo, Adrià Armejach, Jonathan Beard, and Miquel Moreto. 2021. PLANAR: A Programmable Accelerator for near-Memory Data Rearrangement. In *Proceedings of the ACM International Conference on Supercomputing* (Virtual Event, USA) *(ICS '21).* Association for Computing Machinery, New York, NY, USA, 164–176. https://doi.org/10.1145/3447818.3460368

[6] Richard Barrett, Michael Berry, Tony F. Chan, James Demmel, June Donato, Jack Dongarra, Victor Eijkhout, Roldan Pozo, Charles Romine, and Henk van der Vorst. 1994. *Templates for the Solution of Linear Systems: Building Blocks for Iterative Methods.* Society for Industrial and Applied Mathematics. https://doi.org/10.1137/1.9781611971538 arXiv:https://epubs.siam.org/doi/pdf/10.1137/1.9781611971538

[7] Scott Beamer, Krste Asanović, and David Patterson. 2015. The GAP Benchmark Suite.

[8] Maciej Besta, R. Kanakagiri, H. Mustafa, M. Karasikov, G. Ratsch, T. Hoefler, and E. Solomonik. 2020. Communication-Efficient Jaccard similarity for High-Performance Distributed Genome Comparisons. In *2020 IEEE International Parallel and Distributed Processing Symposium (IPDPS).* IEEE Computer Society, Los Alamitos, CA, USA, 1122–1132. https://doi.org/10.1109/IPDPS47924.2020.00118

[9] V. Bharadwaj, A. Buluc, and J. Demmel. 2022. Distributed-Memory Sparse Kernels for Machine Learning. In *2022 IEEE International Parallel and Distributed Processing Symposium (IPDPS).* IEEE Computer Society, Los Alamitos, CA, USA, 47–58. https://doi.org/10.1109/IPDPS53621.2022.00014

[10] M. Brandt, J. Brooks, M. Cahir, T. Hewitt, E. LopezPineda, and D. Sandness. [n. d.]. The Benchmarker's Guide for CRAY SV1 Systems. https://parallel.ru/sites/default/files/ftp/computers/cray/sv1_bmguide.pdf.

[11] Benjamin Brock, Scott McMillan, Aydın Buluç, Timothy Mattson, and José Moreira. 2022. GraphBLAS C++ Specification. https://github.com/GraphBLAS/graphblas-api-cpp.

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

[13] 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

[14] Vidushi Dadu, Jian Weng, Sihao Liu, and Tony Nowatzki. 2019. Towards general purpose acceleration by exploiting common data-dependence forms. In *Proceedings of the 52nd Annual IEEE/ACM International Symposium on Microarchitecture.* 924–939.

[15] Shail Dave, Riyadh Baghdadi, Tony Nowatzki, Sasikanth Avancha, Aviral Shrivastava, and Baoxin Li. 2021. Hardware Acceleration of Sparse and Irregular Tensor Computations of ML Models: A Survey and Insights. *Proc. IEEE* 109, 10 (2021), 1706–1752. https://doi.org/10.1109/JPROC.2021.3098483

[16] Timothy A. Davis and Yifan Hu. 2011. The University of Florida Sparse Matrix Collection. *ACM Trans. Math. Softw.* 38, 1, Article 1 (dec 2011), 25 pages. https://doi.org/10.1145/2049662.2049663

[17] Jack Dongarra and Michael A Heroux. 2013. Toward a new metric for ranking high performance computing systems. *Sandia Report, SAND2013-4744* 312 (2013), 150.

[18] Jack Dongarra, Piotr Luszczek, and M Heroux. 2013. HPCG technical specification. *Sandia National Laboratories, Sandia Report SAND2013-8752* (2013).

[19] R. Espasa, F. Ardanaz, J. Emer, S. Felix, J. Gago, R. Gramunt, I. Hernandez, T. Juan, G. Lowney, M. Mattina, and A. Seznec. 2002. Tarantula: a vector extension to the alpha architecture. In *Proceedings 29th Annual International Symposium on Computer Architecture.* 281–292. https://doi.org/10.1109/ISCA.2002.1003586

[20] Jianhua Gao, Weixing Ji, Fangli Chang, Shiyu Han, Bingxin Wei, Zeming Liu, and Yizhuo Wang. 2023. A Systematic Survey of General Sparse Matrix-Matrix Multiplication. *ACM Comput. Surv.* 55, 12, Article 244 (mar 2023), 36 pages. https://doi.org/10.1145/3571157

[21] Giulia Guidi, Marquita Ellis, Daniel Rokhsar, Katherine Yelick, and Aydın Buluç. 2020. BELLA: Berkeley Efficient Long-Read to Long-Read Aligner and Overlapper. *bioRxiv* (2020). https://doi.org/10.1101/464420 arXiv:https://www.biorxiv.org/content/early/2020/03/23/464420.full.pdf

[22] Giulia Guidi, Oguz Selvitopi, Marquita Ellis, Leonid Oliker, Katherine Yelick, and Aydin Buluc. 2020. Parallel String Graph Construction and Transitive Reduction for De Novo Genome Assembly. In *2020 IEEE International Parallel and Distributed Processing Symposium.*

[23] U. Gupta, C. Wu, X. Wang, M. Naumov, B. Reagen, D. Brooks, B. Cottel, K. Hazelwood, M. Hempstead, B. Jia, H. S. Lee, A. Malevich, D. Mudigere, M. Smelyanskiy, L. Xiong, and X. Zhang. 2020. The Architectural Implications of Facebook's DNN-Based Personalized Recommendation. In *2020 IEEE International Symposium on High Performance Computer Architecture (HPCA).* IEEE Computer Society, Los Alamitos, CA, USA, 488–501. https://doi.org/10.1109/HPCA47549.2020.00047

[24] Kartik Hegde, Hadi Asghari-Moghaddam, Michael Pellauer, Neal Crago, Aamer Jaleel, Edgar Solomonik, Joel Emer, and Christopher W. Fletcher. 2019. ExTensor: An Accelerator for Sparse Tensor Algebra. In *Proceedings of the 52nd Annual IEEE/ACM International Symposium on Microarchitecture* (Columbus, OH, USA) *(MICRO '52).* Association for Computing Machinery, New York, NY, USA, 319–333. https://doi.org/10.1145/3352460.3358275

[25] Torsten Hoefler, Dan Alistarh, Tal Ben-Nun, Nikoli Dryden, and Alexandra Peste. 2021. Sparsity in Deep Learning: Pruning and Growth for Efficient Inference and Training in Neural Networks. *J. Mach. Learn. Res.* 22, 1, Article 241 (jan 2021), 124 pages.

[26] Olivia Hsu, Maxwell Strange, Ritvik Sharma, Jaeyeon Won, Kunle Olukotun, Joel S. Emer, Mark A. Horowitz, and Fredrik Kjølstad. 2023. The Sparse Abstract Machine. In *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 3* (Vancouver, BC, Canada) *(ASPLOS 2023).* Association for Computing Machinery, New York, NY, USA, 710–726. https://doi.org/10.1145/3582016.3582051

[27] M. Hussain, G. Abhishek, A. Buluc, and A. Azad. 2022. Parallel Algorithms for Adding a Collection of Sparse Matrices. In *2022 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW).* IEEE Computer Society, Los Alamitos, CA, USA, 285–294. https://doi.org/10.1109/IPDPSW55747.2022.00058

[28] Konstantinos Kanellopoulos, Nandita Vijaykumar, Christina Giannoula, Roknoddin Azizi, Skanda Koppula, Nika Mansouri Ghiasi, Taha Shahroodi, Juan Gomez Luna, and Onur Mutlu. 2019. SMASH: Co-Designing Software Compression and Hardware-Accelerated Indexing for Efficient Sparse Matrix Operations. In *Proceedings of the 52nd Annual IEEE/ACM International Symposium on Microarchitecture* (Columbus, OH, USA) *(MICRO '52).* Association for Computing Machinery, New York, NY, USA, 600–614. https://doi.org/10.1145/3352460.3358286

[29] Jeremy Kepner and John Gilbert. 2011. *Graph Algorithms in the Language of Linear Algebra.* Society for Industrial and Applied Mathematics. https://doi.org/10.1137/1.9780898719918 arXiv:https://epubs.siam.org/doi/pdf/10.1137/1.9780898719918

[30] Jinsung Kim, Aravind Sukumaran-Rajam, Changwan Hong, Ajay Panyala, Rohit Kumar Srivastava, Sriram Krishnamoorthy, and P. Sadayappan. 2018. Optimizing Tensor Contractions in CCSD(T) for Efficient Execution on GPUs. In *Proceedings of the 2018 International Conference on Supercomputing* (Beijing, China) *(ICS '18).* Association for Computing Machinery, New York, NY, USA, 96–106. https://doi.org/10.1145/3205289.3205296

[31] Fredrik Kjolstad, Shoaib Kamil, Stephen Chou, David Lugato, and Saman Amarasinghe. 2017. The Tensor Algebra Compiler. *Proc. ACM Program. Lang.* 1, OOPSLA, Article 77 (oct 2017), 29 pages. https://doi.org/10.1145/3133901

[32] Onur Kocberber, Boris Grot, Javier Picorel, Babak Falsafi, Kevin Lim, and Parthasarathy Ranganathan. 2013. Meet the walkers accelerating index traversals for in-memory databases. In *2013 46th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO).* 468–479.

[33] Chris Lattner and Vikram Adve. 2004. LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation. In *Proceedings of the International Symposium on Code Generation and Optimization: Feedback-Directed and Runtime Optimization* (Palo Alto, California) *(CGO '04)*. IEEE Computer Society, USA, 75.

[34] Charles E. Leiserson, Neil C. Thompson, Joel S. Emer, Bradley C. Kuszmaul, Butler W. Lampson, Daniel Sanchez, and Tao B. Schardl. 2020. There's plenty of room at the Top: What will drive computer performance after Moore's law? *Science* 368, 6495 (2020), eaam9744. https://doi.org/10.1126/science.aam9744 arXiv:https://www.science.org/doi/pdf/10.1126/science.aam9744

[35] Jiawen Liu, Jie Ren, Roberto Gioiosa, Dong Li, and Jiajia Li. 2021. Sparta: High-Performance, Element-Wise Sparse Tensor Contraction on Heterogeneous Memory. In *Proceedings of the 26th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming* (Virtual Event, Republic of Korea) *(PPoPP '21)*. Association for Computing Machinery, New York, NY, USA, 318–333. https://doi.org/10.1145/3437801.3441581

[36] Scott Lloyd and Maya Gokhale. 2015. In-Memory Data Rearrangement for Irregular, Data-Intensive Computing. *Computer* 48, 8 (2015), 18–25. https://doi.org/10.1109/MC.2015.230

[37] J. Lowe-Power, A. Mutaal Ahmad, A. Akram, M. Alian, R. Amslinger, M. Andreozzi, A. Armejach, N. Asmussen, B. Beckmann, S. Bharadwaj, G. Black, G. Bloom, B. R. Bruce, D. Rodrigues Carvalho, J. Castrillon, L. Chen, N. Derumigny, S. Diestelhorst, W. Elsasser, C. Escuin, M. Fariborz, A. Farmahini-Farahani, P. Fotouhi, R. Gambord, J. Gandhi, D. Gope, T. Grass, A. Gutierrez, B. Hanindhito, A. Hansson, S. Haria, A. Harris, T. Hayes, A. Herrera, M. Horsnell, S. A. R. Jafri, R. Jagtap, H. Jang, R. Jeyapaul, T. M. Jones, M. Jung, S. Kannoth, H. Khaleghzadeh, Y. Kodama, T. Krishna, T. Marinelli, C. Menard, A. Mondelli, M. Moreto, T. Mück, O. Naji, K. Nathella, H. Nguyen, N. Nikoleris, L. E. Olson, M. Orr, B. Pham, P. Prieto, T. Reddy, A. Roelke, M. Samani, A. Sandberg, J. Setoain, B. Shingarov, M. D. Sinclair, T. Ta, R. Thakur, G. Travaglini, M. Upton, N. Vaish, I. Vougioukas, W. Wang, Z. Wang, N. Wehn, C. Weis, D. A. Wood, H. Yoon, and É. F. Zulian. 2020. The gem5 Simulator: Version 20.0+. arXiv:2007.03152 [cs.AR]

[38] Srundefinedan Milaković, Oguz Selvitopi, Israt Nisa, Zoran Budimlić, and Aydin Buluç. 2022. Parallel Algorithms for Masked Sparse Matrix-Matrix Products. In *Proceedings of the 27th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming* (Seoul, Republic of Korea) *(PPoPP '22)*. Association for Computing Machinery, New York, NY, USA, 453–454. https://doi.org/10.1145/3503221.3508430

[39] Anurag Mukkara, Nathan Beckmann, Maleen Abeydeera, Xiaosong Ma, and Daniel Sanchez. 2018. Exploiting Locality in Graph Analytics through Hardware-Accelerated Traversal Scheduling. In *2018 51st Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. 1–14. https://doi.org/10.1109/MICRO.2018.00010

[40] Yusuke Nagasaka, Satoshi Matsuoka, Ariful Azad, and Aydın Buluç. 2018. High-Performance Sparse Matrix-Matrix Products on Intel KNL and Multicore Architectures. In *Workshop Proceedings of the 47th International Conference on Parallel Processing* (Eugene, OR, USA) *(ICPP Workshops '18)*. Association for Computing Machinery, New York, NY, USA, Article 34, 10 pages. https://doi.org/10.1145/3229710.3229720

[41] Tan Nguyen, Colin MacLean, Marco Siracusa, Douglas Doerfler, Nicholas J. Wright, and Samuel Williams. 2022. FPGA-based HPC accelerators: An evaluation on performance and energy efficiency. *Concurrency and Computation: Practice and Experience* 34, 20 (2022), e6570. https://doi.org/10.1002/cpe.6570 arXiv:https://onlinelibrary.wiley.com/doi/pdf/10.1002/cpe.6570

[42] NVIDIA. 2023. Best Practices for Building and Deploying Recommender Systems. https://docs.nvidia.com/deeplearning/performance/recsys-best-practices/index.html

[43] Subhankar Pal, Jonathan Beaumont, Dong-Hyeon Park, Aporva Amarnath, Siying Feng, Chaitali Chakrabarti, Hun-Seok Kim, David Blaauw, Trevor Mudge, and Ronald Dreslinski. 2018. OuterSPACE: An outer product based sparse matrix multiplication accelerator. In *2018 IEEE International Symposium on High Performance Computer Architecture (HPCA)*. IEEE, 724–736.

[44] Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, Alban Desmaison, Andreas Kopf, Edward Yang, Zachary DeVito, Martin Raison, Alykhan Tejani, Sasank Chilamkurthy, Benoit Steiner, Lu Fang, Junjie Bai, and Soumith Chintala. 2019. PyTorch: An Imperative Style, High-Performance Deep Learning Library. In *Advances in Neural Information Processing Systems 32*. Curran Associates, Inc., 8024–8035. http://papers.neurips.cc/paper/9015-pytorch-an-imperative-style-high-performance-deep-learning-library.pdf

[45] Andrea Pellegrini, Nigel Stephens, Magnus Bruce, Yasuo Ishii, Joseph Pusdesris, Abhishek Raja, Chris Abernathy, Jinson Koppanalil, Tushar Ringe, Ashok Tummala, Jamshed Jalal, Mark Werkheiser, and Anitha Kona. 2020. The Arm Neoverse N1 Platform: Building Blocks for the Next-Gen Cloud-to-Edge Infrastructure SoC. *IEEE Micro* 40, 2 (2020), 53–62. https://doi.org/10.1109/MM.2020.2972222

[46] et al. Phipps, Eric. [n. d.]. Genten: Software for Generalized Tensor Decompositions by Sandia National Laboratories. https://gitlab.com/tensors/genten/.

[47] Eric T. Phipps and Tamara G. Kolda. 2019. Software for Sparse Tensor Decomposition on Emerging Computing Architectures. *SIAM Journal on Scientific Computing* 41, 3 (2019), C269–C290. https://doi.org/10.1137/18M1210691 arXiv:https://doi.org/10.1137/18M1210691

[48] Stephan Rabanser, Oleksandr Shchur, and Stephan Günnemann. 2017. Introduction to Tensor Decompositions and their Applications in Machine Learning. *ArXiv* abs/1711.10781 (2017).

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

[50] Mitsuhisa Sato, Yutaka Ishikawa, Hirofumi Tomita, Yuetsu Kodama, Tetsuya Odajima, Miwako Tsuji, Hisashi Yashiro, Masaki Aoki, Naoyuki Shida, Ikuo Miyoshi, Kouichi Hirai, Atsushi Furuya, Akira Asato, Kuniki Morita, and Toshiyuki Shimizu. 2020. Co-Design for A64FX Manycore Processor and "Fugaku". In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis* (Atlanta, Georgia) *(SC '20)*. IEEE Press, Article 47, 15 pages.

[51] Oguz Selvitopi, S. Ekanayake, G. Guidi, G. A. Pavlopoulos, A. Azad, and A. Buluc. 2020. Distributed Many-to-Many Protein Sequence Alignment using Sparse Matrices. In *2020 SC20: International Conference for High Performance Computing, Networking, Storage and Analysis (SC)*. IEEE Computer Society, Los Alamitos, CA, USA, 1–14. https://doi.org/10.1109/SC41405.2020.00079

[52] Ryan Senanayake, Changwan Hong, Ziheng Wang, Amalee Wilson, Stephen Chou, Shoaib Kamil, Saman Amarasinghe, and Fredrik Kjolstad. 2020. A Sparse Iteration Space Transformation Framework for Sparse Tensor Algebra. *Proc. ACM Program. Lang.* 4, OOPSLA, Article 158 (Nov. 2020), 30 pages. https://doi.org/10.1145/3428226

[53] Nicholas D. Sidiropoulos, Lieven De Lathauwer, Xiao Fu, Kejun Huang, Evangelos E. Papalexakis, and Christos Faloutsos. 2017. Tensor Decomposition for Signal Processing and Machine Learning. *IEEE Transactions on Signal Processing* 65, 13 (2017), 3551–3582. https://doi.org/10.1109/TSP.2017.2690524

[54] Marco Siracusa, Emanuele Del Sozzo, Marco Rabozzi, Lorenzo Di Tucci, Samuel Williams, Donatella Sciuto, and Marco Domenico Santambrogio. 2022. A Comprehensive Methodology to Optimize FPGA Designs via the Roofline Model. *IEEE Trans. Comput.* 71, 8 (2022), 1903–1915. https://doi.org/10.1109/TC.2021.3111761

[55] Marco Siracusa and Fabrizio Ferrandi. 2020. Tensor Optimization for High-Level Synthesis Design Flows. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 39, 11 (2020), 4217–4228. https://doi.org/10.1109/TCAD.2020.3012318

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

[57] 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* (Austin, Texas) *(IA<sup>3</sup> '15)*. Association for Computing Machinery, New York, NY, USA, Article 5, 7 pages. https://doi.org/10.1145/2833179.2833183

[58] Nitish Srivastava, Hanchen Jin, Jie Liu, David Albonesi, and Zhiru Zhang. 2020. MatRaptor: A sparse-sparse matrix multiplication accelerator based on row-wise product. In *2020 53rd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. IEEE, 766–780.

[59] Aaron Stillmaker and Bevan Baas. 2017. Scaling equations for the accurate prediction of CMOS device performance from 180nm to 7nm. *Integration* 58 (2017), 74–81. https://doi.org/10.1016/j.vlsi.2017.02.002

[60] Nishil Talati, Kyle May, Armand Behroozi, Yichen Yang, Kuba Kaszyk, Christos Vasiladiotis, Tarunesh Verma, Lu Li, Brandon Nguyen, Jiawen Sun, John Magnus Morton, Agreen Ahmadi, Todd Austin, Michael O'Boyle, Scott Mahlke, Trevor Mudge, and Ronald Dreslinski. 2021. Prodigy: Improving the Memory Latency of Data-Indirect Irregular Workloads Using Hardware-Software Co-Design. In *2021 IEEE International Symposium on High-Performance Computer Architecture (HPCA)*. 654–667. https://doi.org/10.1109/HPCA51647.2021.00061

[61] Nishil Talati, Haojie Ye, Yichen Yang, Leul Belayneh, Kuan-Yu Chen, David Blaauw, Trevor Mudge, and Ronald Dreslinski. 2022. NDMiner: Accelerating Graph Pattern Mining Using near Data Processing. In *Proceedings of the 49th Annual International Symposium on Computer Architecture* (New York, New York) *(ISCA '22)*. Association for Computing Machinery, New York, NY, USA, 146–159. https://doi.org/10.1145/3470496.3527437

[62] Neil C. Thompson and Svenja Spanuth. 2021. The Decline of Computers as a General Purpose Technology. *Commun. ACM* 64, 3 (feb 2021), 64–72. https://doi.org/10.1145/3430936

[63] J. Wawrzynek, K. Asanovic, B. Kingsbury, D. Johnson, J. Beck, and N. Morgan. 1996. Spert-II: a vector microprocessor system. *Computer* 29, 3 (1996), 79–86. https://doi.org/10.1109/2.485896

[64] Samuel Williams, Leonid Oliker, Richard Vuduc, John Shalf, Katherine Yelick, and James Demmel. 2007. Optimization of sparse matrix-vector multiplication on emerging multicore platforms. In *SC '07: Proceedings of the 2007 ACM/IEEE*

Conference on Supercomputing. 1–12. https://doi.org/10.1145/1362622.1362674

[65] Samuel Williams, Andrew Waterman, and David Patterson. 2009. Roofline: An Insightful Visual Performance Model for Multicore Architectures. *Commun. ACM* 52, 4 (apr 2009), 65–76. https://doi.org/10.1145/1498765.1498785

[66] Yifan Yang, Joel S. Emer, and Daniel Sanchez. 2021. SpZip: Architectural Support for Effective Data Compression In Irregular Applications. In *2021 ACM/IEEE 48th Annual International Symposium on Computer Architecture (ISCA)*. 1069–1082. https://doi.org/10.1109/ISCA52012.2021.00087

[67] Xiangyao Yu, Christopher J. Hughes, Nadathur Satish, and Srinivas Devadas. 2015. IMP: Indirect memory prefetcher. In *2015 48th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. 178–190. https://doi.org/10.1145/2830772.2830807

[68] Manzil Zaheer, Guru Guruganesh, Avinava Dubey, Joshua Ainslie, Chris Alberti, Santiago Ontanon, Philip Pham, Anirudh Ravula, Qifan Wang, Li Yang, and Amr Ahmed. 2020. Big Bird: Transformers for Longer Sequences. In *Proceedings of the 34th International Conference on Neural Information Processing Systems* (Vancouver, BC, Canada) *(NIPS'20)*. Curran Associates Inc., Red Hook, NY, USA, Article 1450, 15 pages.

[69] 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*. 687–701.

[70] Lixin Zhang, Zhen Fang, M. Parker, B.K. Mathew, L. Schaelicke, J.B. Carter, W.C. Hsieh, and S.A. McKee. 2001. The Impulse memory controller. *IEEE Trans. Comput.* 50, 11 (2001), 1117–1132. https://doi.org/10.1109/12.966490

[71] Zhekai Zhang, Hanrui Wang, Song Han, and William J Dally. 2020. SpArch: Efficient architecture for sparse matrix multiplication. In *2020 IEEE International Symposium on High Performance Computer Architecture (HPCA)*. IEEE, 261–274.