A Sparse Iteration Space Transformation Framework for Sparse Tensor Algebra

RYAN SENANAYAKE, Reservoir Labs
CHANGWAN HONG, MIT CSAIL
ZIHENG WANG, MIT CSAIL
AMALEE WILSON, Stanford University
STEPHEN CHOU, MIT CSAIL
SHOAIB KAMIL, Adobe Research
SAMAN AMARASINGHE, MIT CSAIL
FREDRIK KJOLSTAD, Stanford University

We address the problem of optimizing sparse tensor algebra in a compiler and show how to define standard loop transformations—split, collapse, and reorder—on sparse iteration spaces. The key idea is to track the transformation functions that map the original iteration space to derived iteration spaces. These functions are needed by the code generator to emit code that maps coordinates between iteration spaces at runtime, since the coordinates in the sparse data structures remain in the original iteration space. The result is a new iteration order that still iterates over only the subset of nonzero coordinates defined by the data structures. We further demonstrate that derived iteration spaces can tile both the universe of coordinates and the subset of nonzero coordinates: the former is analogous to tiling dense iteration spaces, while the latter tiles sparse iteration spaces into statically load-balanced blocks of nonzeros. Tiling the space of nonzeros lets the generated code efficiently exploit heterogeneous compute resources such as threads, vector units, and GPUs.

We implement these concepts and an associated scheduling API in the open-source TACO system. The scheduling API can be used by performance engineers or it can be the target of an automatic scheduling system. We outline one heuristic autoscheduling system, but many other systems are both possible and enabled by our API. Using the scheduling API, we show how to optimize sparse and mixed sparse-dense tensor algebra expressions on both CPUs and GPUs. Our results show that the sparse transformations are sufficient to generate code with competitive performance to hand-optimized implementations from the literature, while generalizing to all of the tensor algebra.

CCS Concepts: • **Software and its engineering → Source code generation**; **Domain specific languages**; • **Mathematics of computing → Mathematical software performance**;

Additional Key Words and Phrases: Sparse Tensor Algebra, Sparse Iteration Spaces, Optimizing Transformations

1 INTRODUCTION

Sparse tensor algebra compilers, unlike their dense counterparts, lack an iteration space transformation framework. Dense tensor algebra compilers, such as TCE [Auer et al. 2006], Halide [Ragan-Kelley et al. 2012], TVM [Chen et al. 2018], and TC [Vasilache et al. 2018], build on many decades of research on affine loop transformations [Allen and Cocke 1972; Wolfe 1982], leading to sophisticated models like the polyhedral model [Feautrier 1988; Lamport 1974]. Despite progress over the last three decades [Bik and Wijshoff 1993; Kotlyar et al. 1997; Strout et al. 2018], we lack a unifying sparse iteration space framework that can model all of the sparse tensor algebra.

Authors' addresses: Ryan Senanayake, Reservoir Labs, senanayake@reservoir.com; Changwan Hong, MIT CSAIL, changwan@mit.edu; Ziheng Wang, MIT CSAIL, ziheng@mit.edu; Amalee Wilson, Stanford University, amalee@cs.stanford.edu; Stephen Chou, MIT CSAIL, changwan@mit.edu; Shoaib Kamil, Adobe Research, kamil@adobe.com; Saman Amarasinghe, MIT CSAIL, saman@csail.mit.edu; Fredrik Kjolstad, Stanford University, kjolstad@cs.stanford.edu.

2020. XXXX-XXXX/2020/10-ART $15.00
https://doi.org/

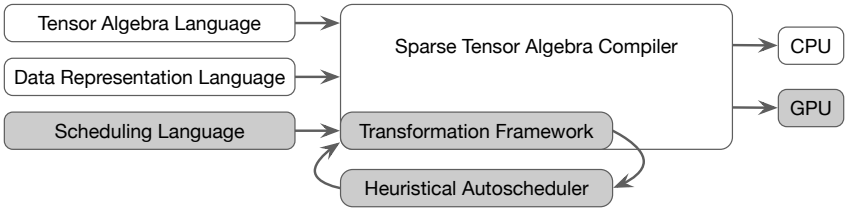


Fig. 1. Overview of the sparse tensor algebra compiler that highlights the contributions of this paper.

Without an iteration space transformation framework, sparse tensor algebra compilers [Chou et al. 2018; Kjolstad et al. 2017b] leave crucial optimizations on the table, such as tiling, parallelization, vectorization, and static load-balancing techniques to make effective use of GPUs. Furthermore, sparse tensor algebra expressions often contain both dense and sparse subexpressions, where some operands are stored in sparse data structures and some in dense arrays. Examples include sparse matrix-vector multiplication with a dense vector (SpMV) and the matricized tensor times Khatri-Rao product with dense matrices (MTTKRP). Without a general transformation framework, sparse tensor algebra compilers cannot optimize the dense loops in mixed sparse and dense expressions.

In addition to the general challenges of sparse code generation [Chou et al. 2018; Kjolstad et al. 2017b], the primary challenge of a sparse transformation framework is that the access expressions that index into arrays are not always affine expressions of the loop indices. Instead, they may be computed from coordinates and positions that are loaded from compressed data structures. Thus, the compiler cannot merely rewrite the access expressions to affine functions of the new loop indices, as in a dense transformation framework. Nor will we allow it to rewrite the coordinates in the compressed data structures.[1] Thus, the compiler must keep a log of the transformation functions, which we call a provenance graph, so that it can generate code to map between coordinates in the original iteration space (stored in the data structures) and coordinates in the rewritten/derived iteration space (reflected by the loop indices). In addition, two secondary challenges are how to support loop collapsing, where the resulting loop must iterate over the Cartesian combination of the data structures the original loops iterated over, and how to control the locations of the boundaries between tiles for load-balanced execution.

In this paper, we propose a unified sparse iteration space transformation framework for the dense and sparse iteration spaces that come from sparse tensor algebra. The transformations are applied to a high-level intermediate representation (IR) before the compiler generates sparse imperative code and therefore side-steps the need for sophisticated analysis. The transformations let us split (strip-mine), reorder, collapse, vectorize, and parallelize both dense and sparse loops, subject to straightforward preconditions. Furthermore, the transformations can split sparse loops both in the full iteration space corresponding to that of a dense loop nest (a coordinate space) and in the subset given by the nonzeros in one of the compressed data structures (a position space). The latter results in load-balanced execution. Our specific contributions are:

(1) Generalization of the split and collapse transformations to sparse iteration spaces;
(2) Provenance graphs that store the functions that map derived iteration spaces to the original iteration space, so that the compiler can generate code to map coordinates between these spaces at run-time;
(3) Position iteration spaces to complement coordinate iteration spaces for tiling; and
(4) A generalization of scheduling languages to sparse iteration spaces.

As depicted in Figure 1, we have implemented these ideas in the open-source TACO sparse tensor algebra compiler [Kjolstad 2020; Kjolstad et al. 2017b], which can generate code for any sparse tensor

[1] See Chou et al. [2020] for a treatment on how to implicitly tile an iteration space by instead transforming the data structures.

```
a(i) = B(i,j) * c(j);
```

(a) Tensor Algebra Expression


```
Tensor<double> a({m},  {Dense});
Tensor<double> B({m,n},{Dense,Compressed});
Tensor<double> c({n},  {Dense});
```

(b) CSR data structure and data representations

```
for (int i = 0; i < m; i++) {
  for (int p = B_pos[i]; p < B_pos[i+1]; p++) {
    int j = B_crd[p];
    a[i] += B[p] * c[j];
  }
}
```

(c) Unscheduled SpMV

```
y.split(i, i1, i2, 32);
 .parallelize(i1, CPUThread, Atomics);
```

```
#pragma omp parallel for schedule(runtime, 1)
for (int i1 = 0; i1 < CEIL(m,32); i1++) {
  for (int i2 = 0; i2 < 32; i2++) {
    int i = i1*32 + i2;
    if (i >= m) break;
    for (int p = B_pos[i]; p < B_pos[i+1]; p++) {
      int j = B_crd[p];
      a[i] += B[p] * c[j];
    }
  }
}
```

(d) CPU-scheduled SpMV

```
IndexVar f, p, p1, p2, block, warp, thread;
IndexVar thread_nz, thread_nz_pre;
TensorVar t(Type(Float64, {thread_nz}), Dense);
y.collapse(i, j, f)
 .pos(f, p, B(i,j))
 .split(p, block, p1, NNZ_PER_THREAD * BLOCK_SIZE)
 .split(p1, warp, p2, NNZ_PER_THREAD * WARP_SIZE)
 .split(p2, thread, thread_nz, NNZ_PER_THREAD)
 .reorder({block, warp, thread, thread_nz})
 .precompute(texpr, thread_nz, thread_nz_pre, t)
 .unroll(thread_nz_pre, NNZ_PER_THREAD)
 .parallelize(block,  GPUBlock, IgnoreRaces)
 .parallelize(warp,   GPUWarp, IgnoreRaces)
 .parallelize(thread, GPUThread, Atomics);
```

```
__global__ void gpuKernel(...) {
  ...
  int p_start = block * 8192 + warp * 512 + lane * 16;
  int i = binarySearch(B_pos, block_row_starts[block],
          block_row_starts[block+1], p_start);
  double t[16];

  #pragma unroll
  for (int thread_nz = 0; thread_nz < 16; thread_nz++) {
    int p = p_start + thread_nz;
    if (p >= B_nnz) break;

    int j = B_crd[p];
    t[thread_nz] = B[p] * c[j];
  }

  double row_sum = 0.0;

  for (int thread_nz = 0; thread_nz < 16; thread_nz++) {
    int p = p_start + thread_nz;
    if (p >= B_nnz) break;
    while (p == B_pos[i+1]) i++;

    row_sum += t[thread_nz];
    if (p+1 == B_pos[i+1]) {
      atomicAdd(&a[i], row_sum);
      row_sum = 0.0;
    }
  }
  atomicAddWarp<double>(a, i, row_sum);
}
```

(e) GPU-scheduled SpMV

Fig. 2. With this work, the sparse tensor algebra compiler separates algorithm, data representation, and schedule. Figure (a) shows the tensor algebra expression and Figure (b) shows the data representation descriptions. These and the schedule can be changed independent of each other. Figures (c)–(e) shows three different schedules for the given tensor algebra expression and data representation: the default schedule (c), a schedule optimized for CPUs (d), and a schedule optimized for NVIDIA GPUs (e).

algebra expression. We have contributed our changes back to the TACO repository (https://github.com/tensor-compiler/taco). The scheduling language is thus available in the TACO library, the command-line tool [Kjolstad et al. 2017a], and the web code generation tool (http://tensor-compiler.org/codegen.html). We expose the transformations as a scheduling API analogous to the Halide scheduling language [Ragan-Kelley et al. 2012], but unlike Halide the transformations apply to both sparse and dense loops from sparse and dense tensor algebra expressions. Figure 2 shows three example schedules for the sparse matrix-vector multiplication with a CSR matrix. We also describe an automatic heuristic scheduler that automatically finds good schedules that the programmer can override or adapt. Finally, we demonstrate that these transformations can be used to generate code with performance competitive with hand-optimized libraries, as well as code that efficiently uses GPUs, and code implementing static load-balancing for sparse computations.

2 EXAMPLE

In this section, we provide a simple end-to-end example that demonstrates our sparse iteration space transformation and code generation framework. Our framework generalizes to tensor algebra where the tensors have any number of modes (dimensions), but for simplicity we use the well-known example of matrix-vector multiplication $a = Bc$.

Code for matrix-vector multiplication is straightforward and easy to optimize when all operands are dense because of the affine loop bounds and dense array accesses. But when the matrix is sparse, as shown in Figure 2c, the loops become more complicated, because sparse matrices are typically stored in *compressed* data structures that remove zeros. Our example uses the compressed sparse rows (CSR) format, and we show an example in Figure 2b. (The compiler framework, however, generalizes to other formats [Chou et al. 2018].) Like Kjolstad et al. [2017b], we express sparse tensor formats as a composition of per-level (i.e., per-mode) formats, such that each level represents a tensor mode that provides the capability to access the coordinates in a subsequent tensor mode that belong to a given subslice in the represented mode. This lets us conceptualize sparse tensors as coordinate tree hierarchies and define iteration over these trees, as shown in Figure 3. For example, the CSR format is defined as a dense level of rows and a compressed level of columns. The combination of complicated iteration and indirect storage, common to sparse tensor formats, makes subsequent loop transformations challenging, even for a simple expression like matrix-vector multiplication.

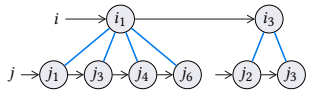


Fig. 3. Coordinate tree hierarchy.

There are two primary ways to parallelize a sparse matrix-vector multiplication (SpMV) with a CSR matrix. The rows of the matrix can be divided among the threads (as shown in Figure 2d), which has the benefit that all threads can independently compute their respective components of the output vector. Alternatively, the nonzero elements of the matrix can be divided. This requires synchronization when a row is shared across multiple threads, but is load balanced and thus leads to significant performance improvement over the row-split strategy when the matrix has a skewed distribution of nonzeros per row. Such an uneven distribution can cause threads assigned to denser rows to become a bottleneck for the computation, which is especially relevant on GPUs where thousands of threads may wait for a single thread to finish.

For complex computations that involve a format whose outer dimension is also compressed or where there are multiple sparse data structures, there are many more profitable ways to parallelize. Our framework lets you specify how multiple sparse data structures should be partitioned, while maintaining correctness and minimizing overhead. Details are provided in Section 3.

We will now walk through how to write the SpMV schedule in Figure 4a, which instructs our compiler to partition the nonzero elements of the matrix evenly across parallel threads. Figure 4b shows the generated code for this schedule and highlights the extensions to our prior work on sparse tensor algebra code generation [Kjolstad et al. 2017b]. This compound transformation consists of three basic transformations, as shown in Figure 5. First, the loops are collapsed so that a single loop iterates over both dimensions of the computation. Second, the collapsed loop is split (strip-mined) into two nested loops, such that the inner loop iterates over a constant number of nonzeros of the matrix. Third and finally, the outer loop is parallelized over CPU threads with atomic instructions to serialize updates. As discussed further in Section 4, the split transformation is preceded by a pos transformation that specifies that subsequent splits should divide the sparse matrix nonzeros into equal parts. That is, we split the iteration space with respect to the position space of the matrix.

Figure 5 shows how these transformations transform the iteration over the coordinate trees (middle) and the code (bottom). The top row of the figure shows how the transformations transform the iteration graph IR that is used by the sparse tensor algebra compiler [Kjolstad et al. 2017b],

```
// Data representations
Tensor<double> a({m},  {Dense});
Tensor<double> B({m,n},{Dense,Compressed});
Tensor<double> c({n},  {Dense});

// Tensor algebra
a(i) = B(i,j) * c(j);

// Schedule
a.collapse(i, j, f)
 .pos(f, p, B(i,j))
 .split(p, p0, p1, Down, 16)
 .parallelize(p0, CPUThread, Atomics);

std::cout << code(a) << std::endl;
```

(a) User-provided C++ code.

```
#pragma omp parallel for schedule(runtime, 1)
for (int p0 = 0; p0 < CEIL(B2_pos[B1_dim],16); p0++) {
  int i = binary_search(B2_pos, 0, B1_dim, p0*16);
  double t = 0.0;
  for (int p1 = 0; p1 < 16; p1++) {
    int p = p0 * 16 + p1;
    if (p >= B2_pos[B1_dim]) break;
    int j = B2_crd[p] % B2_dim;
    while (p == B2_pos[(i+1)]) i++;

    t += B[p] * c[j];
    if (p+1 == B2_pos[(i+1)]) {
      #pragma omp atomic
      a[i] += t;
      t = 0.0;
    }
  }
  #pragma omp atomic
  a[i] += t;
}
```

(b) Generated parallel SpMV C code.

Fig. 4. User-provided C++ code (a) used to generate SpMV C code (b) that distributes nonzeros evenly among parallel threads. The generated code is colored to reflect different components of code generation that we add to allow for generation of code from transformed iteration spaces. Index variable recovery is shown in red, derived loop bounds in green, and iteration guards in blue. The user may also insert values into B and c and print a, which triggers automatic compilation and execution behind the scenes.

which we have generalized in this work as described in Section 3. Iteration graphs represent iteration over sparse iteration spaces in terms of iteration over the coordinate trees of the operands. Nodes represent dimensions of the iteration space and the vertical ordering of the nodes represent the lexicographical ordering of the iteration. Each path in an iteration graph represents iteration over the levels of one tensor coordinate tree, and together they represent the iteration over the intersection or union (i.e., co-iteration) of coordinates from several tensor coordinate trees. More background is provided in Section 3. Old versions of the iteration graph are maintained within a provenance graph data structure, which horizontally spans the top row of the figure. The parallelize transformation is then applied on the final iteration graph shown in the figure to generate the parallel code that appears in Figure 4b. This partitioning strategy is the basis for our schedule for the SpMV optimized for NVIDIA V100 GPUs in Figure 2e, which we evaluate in Section 7.

During code generation the final iteration graph is traversed top-down to generate code to iterate over the corresponding sparse data structures at each node.[2] Traversals of the provenance graph are used to produce different pieces of the generated code. Figure 4b highlights these components in different colors: green for loop bounds, red for computing index variables that no longer appear in the iteration graph but are needed to index into data structures, and blue for code to guard the iteration from visiting out-of-bounds points[3].

We choose SpMV as a simple example, but our transformations generalize to any tensor algebra expression. For example, implementations with higher-dimensional tensors, multiple sparse data structures, and more complicated optimization and tiling strategies can be generated by our technique, whereas hand-writing such implementations become increasingly difficult as more complex data structures are combined. It is also noteworthy that while in this example, there is a one-to-one mapping from index variables to for-loops in the generated code, this is not true in the general case of code that includes computing intersections and unions over multiple sparse data

[2]See Section 5 for a description of the extensions introduced in this work to the code generation algorithm described by Kjolstad et al. [2017b], and see Kjolstad [2020, Chapter 5] for the complete code generation algorithm.

[3]To enhance readability, we show the iteration guards inside the loop nest, but in our implementation we clone the loops and apply the iteration guard outside to determine which loop nest to enter.

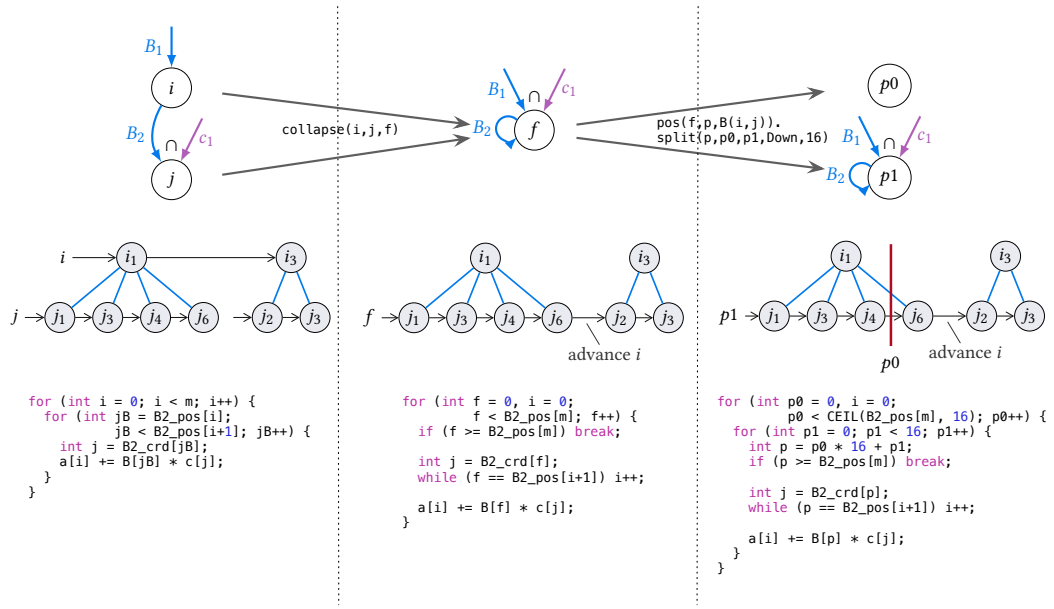


Fig. 5. An end-to-end example depicting how each transformation on a SpMV expression modifies from top-to-bottom: the iteration of the coordinate hierarchy of the sparse matrix, the iteration graph, and the generated code. The provenance graph data structure horizontally spans the figure such that the provenance graph after each transformation consists of the subgraph contained within that transformation's column and the columns to the left of it.

structures. In the next section, we will describe derived sparse iteration spaces, more generally, which will motivate our set of transformations, described in Section 4, as well as the iteration graph and provenance graph abstractions that are used during code generation (Section 5).

3 DERIVED SPARSE ITERATION SPACES

The iteration space of the loops that iterate over tensors in a tensor algebra expression can be described as a hyperrectangular grid of points, by taking the Cartesian product of the iteration domain of each index variable in the expression. Figure 6b shows the iteration space of a dense matrix-vector multiplication. Because the iteration space is dense, the grid contains every point. A sparse iteration space, on the other hand, is a grid with holes, as shown in Figure 7b. The holes are empty grid locations that should be skipped when iterating over the space. Locations that should be visited are determined by sparse tensor data structures that only store the coordinates of nonzero-valued components, shown abstractly as a coordinate tree in Figure 7a[4]. Missing coordinates in these data structures correspond to tensor components whose values are zero. In sparse tensor algebra, we can avoid iterating over the zeros because $a + 0 = a$ and $a \cdot 0 = 0$.

The sparse iteration space of a sparse tensor algebra expression is sparse because it iterates over a subset of the points. This subset is described as intersections (stemming from multiplications) and unions (stemming from additions) of the data structures of the tensor operands. In many expressions, however, the data structures are not element-wise combined as in $A_{ij} = B_{ij} + C_{ij}$, but instead different tensors are indexed by different subsets of the index variables. A simple example is

[4]The abstract coordinate trees we show are replaced by concrete data structures during code generation [Chou et al. 2018].

matrix-vector multiplication $a_i = \sum_j B_{ij} c_j$, where B is indexed by i and j while c is indexed only by j. A slightly more involved example is matrix multiplication $A_{ij} = \sum_k B_{ik} C_{kj}$, where B and C are indexed by different subsets of the three index variables. Because the indexing may not be perfectly aligned, the intersections and unions must be expressed per tensor dimension, where the tensor dimensions correspond to levels in the coordinate trees. Figure 8 (left) shows the coordinate trees of B and c in a (sparse) matrix-vector multiplication. In the sparse iteration space of the expression, the first dimension iterates over the i coordinates of B. The second dimension, however, must iterate over the intersection of the j coordinates in B and in c.

The transformations in this section transform iteration spaces by transforming an intermediate representation called iteration graphs. We call the transformed iteration spaces derived iteration spaces.

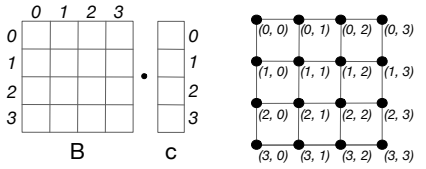
Iteration Graphs

The TACO sparse tensor algebra compiler represents sparse iteration spaces with an intermediate representation called iteration graphs from which its code generation algorithm produces sparse code [Kjolstad 2020; Kjolstad et al. 2017b]. Nodes in iteration graphs represent dimensions of the iteration space, while the vertical ordering of nodes represents the lexicographical order of the dimensions. The paths in iteration graphs symbolically represent the coordinate hierarchies. Figure 8 (middle) first shows the iteration graphs for iterating over B and c individually. The paths in these two iteration graphs symbolically represent all the paths in B's and c's coordinate hierarchy respectively. Next, the individual iteration graphs are merged to form an iteration graph for iterating over the multiplication of B and c. Notice how the j variable becomes the intersection of the second dimension of B and c.

The transformations we discuss in this paper transform sparse iteration spaces by transforming iteration graphs. For example, the two most important transformations, split and collapse, substitute two and one index variables for one and two derived index variables respectively. We call the resulting iteration graphs derived iteration graphs, where some of the index variables are derived from index variables of the original iteration graph.

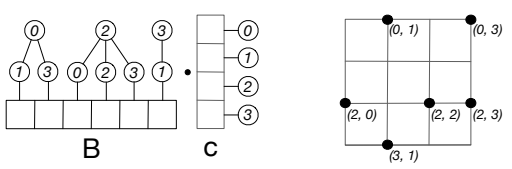
Provenance Graphs

A sparse iteration space transformation framework must keep a log of the transformations that produce derived iteration spaces. This is in contrast to dense iteration space transformation frameworks such as the unimodular transformation framework [Banerjee 1990] and the polyhedral model [Lamport 1974]. When splitting and collapsing dimensions in a dense transformation framework, the array access expressions can be rewritten to use the new loop index variables. It is not possible to



(a) Data structures (b) Iteration space

Fig. 6. The data structures and iteration space of dense matrix-vector multiplication (GEMV).



(a) Data structures (b) Iteration space

Fig. 7. The data structures and iteration space of sparse matrix-vector multiplication (SpMV).

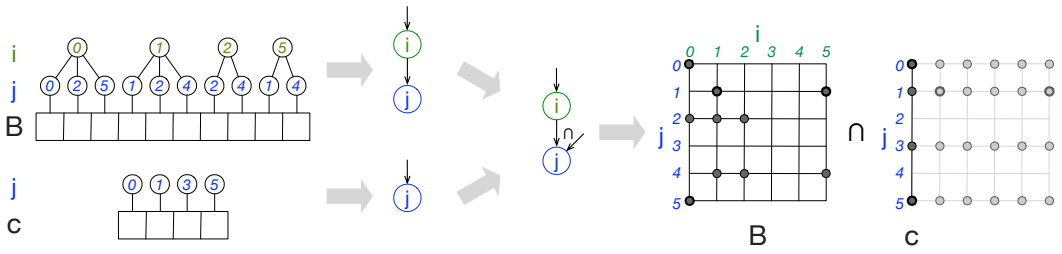


Fig. 8. Iteration graph example for an SpMSpV expression $a_i = \sum_j B_{ij} c_j$. The coordinate hierarchies (left) of matrix B and vector c are first summarized as iteration graphs (center left). These are then merged (center right) as described by the expression and, since the operation is a multiplication, the iteration graph operation is an intersection. The iteration graph describes the intersected iteration space of the coordinate trees (right).

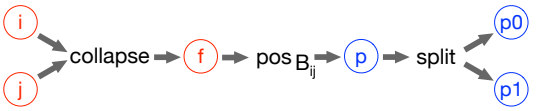


Fig. 9. The index variable provenance graph after parallelizing SpMV in the CSR matrix's position space maps derived index variables to the index variables they derived from. Blue variables are in position space.

rewrite the array access expressions in sparse computations, although they are also affine, because they depend on coordinates stored in the compressed tensor data structures. Because we do not permit the compiler to rewrite the tensor data structures in response to scheduling commands[5], it must keep a log of the transformation functions that produced derived index variables. Keeping track of these functions allows the compiler to emit code that maps between coordinates in the derived space (iterated over by the loop index variables) and coordinates in the original space (stored in the compressed data structures).

Provenance graphs are the intermediate representation which tracks transformations that produce a derived iteration graph. Section 5 describes the extension to the TACO code generation algorithm that enables producing sparse code from derived iteration graphs. The resulting code generation algorithm, described in full in Kjolstad [2020, Chapter 5], maps coordinates between derived iteration spaces and the original iteration spaces. Figure 9 contains the provenance graph for the transformations in our end-to-end example in Figure 5. As transformations are applied, the compiler builds a provenance graph that is then used during code generation.

Provenance Graph Functions

The split function strip-mines a computation by turning a single index variable (i.e., loop) into two nested derived index variables whose combined iteration domain (the Cartesian combination of their domains) has the same cardinality as the domain of the original index variable. Moreover, values of the combined iteration domain of the derived index variables map one-to-one to the

[5]An important design point is that we do not permit the compiler to implicitly rewrite data structures in response to scheduling commands, because it is useful to support code transformations independently of data structure transformations. By separating these concerns, our compiler can produce tiled and parallelized code that does not *require* potentially expensive data structure transformations. There are, of course, situations when users instead want to block data by rewriting data structures. In our design, blocked data structures are expressed in the data representation language [Chou et al. 2018; Kjolstad et al. 2017b] and data transformations can be represented as assignments [Chou et al. 2020], potentially facilitated by the precompute transformations [Kjolstad et al. 2019]

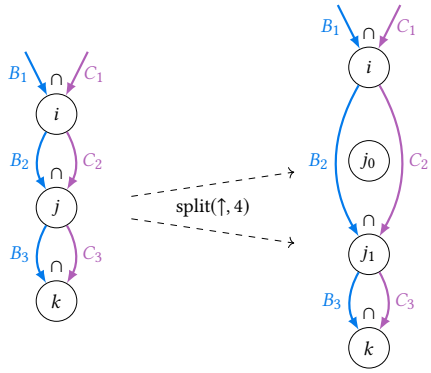


Fig. 10. The split function that is recorded in the provenance graphs splits one index variable into two nested index variables. The iteration space of the index variable that is split is the Cartesian combination of the resulting variables.

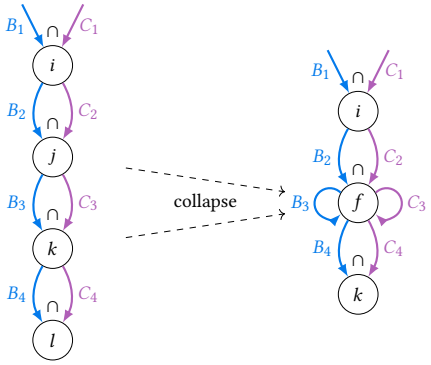


Fig. 11. The collapse function that is recorded in the provenance graphs collapses two nested index variables into a single index variable. The iteration space of the resulting variable is the Cartesian combination of the variables that are collapsed.

original index variables through an affine function. One of the derived index variables after the split has a constant-sized domain and the split function's size argument determines the size; the size of the domain of the other index variable is determined so that together they have the same number of coordinates as the original index variable. The split function takes two arguments: the size of the split and a direction. The size determines the size of the constant-size loop, while the direction—down or up—determines whether the inner or outer loop should be of the given size. Figure 10 shows an up-split with a size of 4, meaning the outer index variable j_0 has a domain whose cardinality is 4. Note also that the data structures are co-iterated over by the inner index variable (and resulting loop). Thus, there are four blocks and each block iterates over the subset of coordinates contained in one-fourth of the entire iteration domain.

We distinguish between two different types of iteration spaces that the split transformation may strip-mine: coordinate spaces and position spaces. A coordinate space is the sparse iteration space given by the domain of an index variable, while a position space is the subset of coordinates that are (contiguously) stored in one of the data structures. A split in the coordinate space evenly divides the sparse space, while a split in the position space of one of the operand evenly divides its nonzero coordinates. Figure 12 shows the two types of split. On the left is a sparse iteration space that arises from iterating over a sparse vector, and on the right is the coordinates stored in that same vector. The black points in the iteration space are the points that are visited because they are stored in the coordinate list. The purple dotted lines show that a split in the coordinate space evenly divides the sparse iteration space, while unevenly dividing the coordinates. Conversely, the red line shows that a split in the position space evenly divides the coordinates, while unevenly dividing the sparse iteration space.

The collapse function flattens two index variables (i.e., loops) into one index variable whose domain is the Cartesian combination of the domains of the original index variables. As shown in Figure 5, the effect of a collapse transformation is to iterate over the bottom of a coordinate hierarchy and, for each iteration, to recover the index variable values of the coordinate tree levels corresponding to the collapsed index variables. Figure 11 shows a collapse function applied to the middle nodes in an iteration graph with four levels. Note how the edges incoming on both collapsed nodes are incoming on the new node. This represents the iteration over their Cartesian combination.

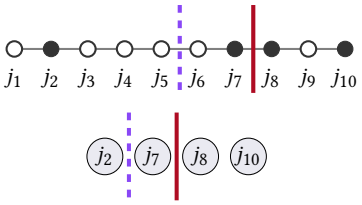


Fig. 12. The iteration space of iterating over a sparse vector (top) and the its nonzero coordinates (bottom). The stippled purple coordinate split divides the iteration space evenly, while the red position split divides the nonzero coordinates evenly.

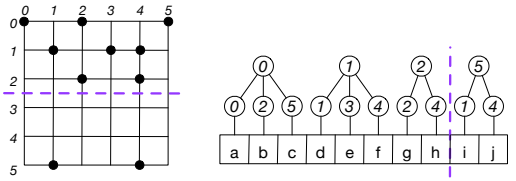


Fig. 13. The iteration space of iterating over a sparse matrix (left) and its nonzero coordinates (right). A coordinate split in the row dimension evenly divides the iteration space, but unevenly divides the rows with nonzeros as well as the nonzeros themselves.

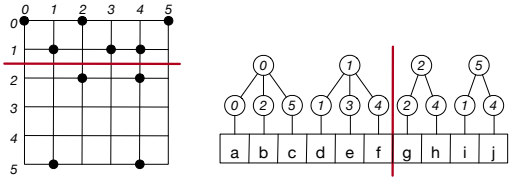


Fig. 14. The iteration space of iterating over a sparse matrix (left) and its nonzero coordinates (right). A position split in the row dimension evenly divides the rows with nonzeros, but unevenly divides the iteration space as well as the nonzeros themselves.

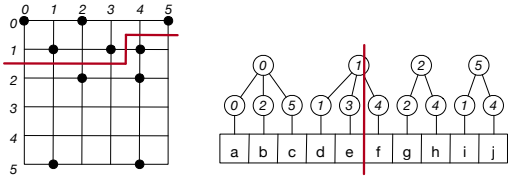


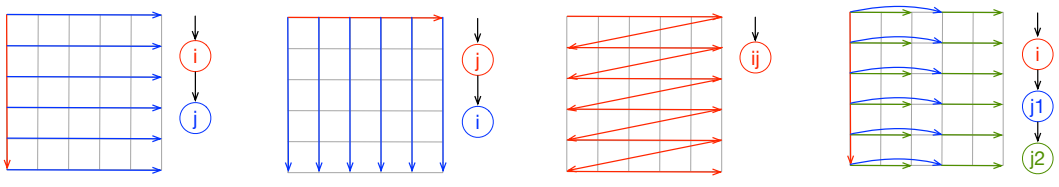
Fig. 15. The iteration space of iterating over a sparse matrix (left) and its nonzero coordinates (right). A position split in the collapsed row and column dimension evenly divides the nonzeros, but unevenly divides the iteration space.

Tiling

The pos, split, and collapse functions can be combined to express different types of tiled iteration spaces. Figures 13–15 shows three different ways to cut a two-dimensional iteration space into two pieces along the horizontal axis. Figure 13 splits the row dimension in the coordinate space, meaning the rows are divided into equal tiles. The coordinate split is the sparse analogue of the traditional dense strip-mine transformation. Figure 14 and Figure 15 both split in the position space: Figure 14 in the position space of the row dimension, and Figure 15 in the position space of the collapsed row and column dimensions. Position splits evenly divide the nonzero coordinates of one of the tensors that is iterated over in the sparse iteration space. Figure 14 evenly divides the rows with nonzeros, while Figure 15 evenly divides the nonzeros themselves.

4 TRANSFORMATIONS

The transformations in this paper—`pos`, `coord`, `collapse`, `split`, `reorder`, `precompute`, `unroll`, `bound`, and `parallelize`—provide a comprehensive loop transformation framework for controlling iteration order through sparse iteration spaces. The transformations let us control the order of computation, so that we can optimize data access locality and parallelism. All transformations apply to index variables in either the coordinate space or the position space, and the `coord` and `pos` transformations transition index variables between these spaces. Figure 16 shows the effect of the `reorder`, `collapse`, and `split` transformations on an i, j iteration space in terms of the original iteration space. Although they are here shown separately, transformations are typically used together, with some adding or removing iteration space dimensions that other transformations reorder or tag for parallel execution.



(a) Original (b) `reorder(i,j)` (c) `collapse(i,j,ij)` (d) `split(j,j1,j2)`

Fig. 16. An original row-major nested iteration of a two-dimensional coordinate iteration space is shown alongside various transformed nested iterations caused by different iteration space transformations.

The key to our approach is that the transformations operate on the iteration graph intermediate representation before sparse code is generated. The iteration graph representation makes it possible to reason about sparse iteration spaces algebraically without the need for sophisticated dependency and control flow analysis of sparse code, which may contain while loops, conditionals and indirect accesses. The dependency, control flow, and sparsity information that are implicit in the iteration graph representation are incorporated when iteration graphs are lowered to code, as described in the next section.

If a valid iteration graph is provided to any transformation and if the preconditions for this transformation pass, then the definitions of our transformations guarantee that the correctness of the transformed iteration graph will yield the same results for all possible inputs. Preconditions for all transformations are provided below. This is because all transformations are defined such that every point in the original iteration space appears exactly once in the transformed iteration space. `Collapse` and `split` are affine functions and therefore bijective, `reorder` and `parallelize` only change the order of iteration without modifying the iteration space, and the correctness of the `precompute` transformation is given in [Kjolstad et al. 2019]. All other transformations do not modify the iteration space. The below preconditions verify that the changes to the iteration order preserve correctness, based on the associativity, commutativity, and distributivity properties of tensor algebra. Finally, we only augment the existing TACO code generation machinery [Kjolstad et al. 2017b] with the ability to work with transformed iteration spaces and pass all existing test cases as well as new ones we added to test our scheduling APIs and GPU backend (1500+ tests total).

We expose the sparse transformation primitives as a scheduling API in TACO. The scheduling language is independent of both the algorithmic language (used to specify computations) and the format language (used to specify tensor data structures). This lets users schedule tensor computations independently of data structure choice, while ensuring correctness for the overall algorithm, and further enables efficient execution on different hardware without changing the algorithm. The scheduling language is a contribution of this paper, so we show the API for every supported transformation. However, two of these transformations, `reorder` and `precompute`, were previously described by Kjolstad et al. [2019]. We include them here for completeness because we included them in our scheduling language, allowing them to be composed of other transformations. We now present the API and describe each transformation:

Pos and Coord. The `pos(IndexVar i, IndexVar p, Access a)` transformation takes an index variable `i` in the coordinate space and replaces it with a new derived index variable `p` that operates over the same iteration range, but with respect to one input `a`'s position space. The `coord (IndexVar p, IndexVar i)` transformation, on the other hand, takes an index variable

in position space `p` and replaces it with a new derived index variable `i` that iterates over the corresponding iteration range in the coordinate iteration space.

Preconditions. The index variable `i` supplied to the `pos` transformation must be in coordinate space. Position spaces refer to a specific tensor's nonzero components. Therefore, the `pos` transformation also takes a tensor input `a` that should be used. This input must appear in the computation expression and also be indexed by the coordinate index variable `i`. Some sparse tensor formats (such as CSR) can only be iterated efficiently in certain orders. For example, some matrix formats can be iterated in (i,j), but not (j,i) ordering. If the coordinate index variable `i` is derived by collapsing multiple variables, then the in-order list of collapsed variables must appear as a sub-list in a valid index variable ordering supported by the tensor format. This precondition is necessary to allow for the corresponding levels of the tensor format to be used as an efficient map from position to coordinate space as described in Section 3. The index variable `p` supplied to the `coord` transformation must be in position space.

Collapse. The `collapse(IndexVar i, IndexVar j, IndexVar f)` transformation collapses two index variables (i and j) that are iterated in-order (directly nested in the iteration graph). It results in a new collapsed index variable `f` that iterates over the Cartesian product of the coordinates of the collapsed index variables. This transformation by itself does not change iteration order but facilitates other transformations such as iterating over the position space of several variables and distributing a multi-dimensional loop nest across threads on a GPU.

Preconditions. The `collapse` transformation takes in two index variables (`i` and `j`). The second index variable `j` must be directly nested under the first index variable `i` in the iteration graph. In addition, the outer index variable `i` must be in coordinate space. This allows us to isolate preconditions related to position spaces to the `pos` transformation. Multi-dimensional position spaces are still possible, by first collapsing multiple dimensions and then applying the `pos` transformation. Collapsing an outer coordinate variable with an inner position variable is still possible and will similarly maintain the iteration order. The `collapse` transformation does not require any reduction-related preconditions as the order of the iteration is maintained through this transformation and reductions can be computed in the same order.

Split. The `split(IndexVar i, IndexVar i1, IndexVar i2, Direction d, size_t s)` transformation splits an index variable `i` into two nested index variables (`i1` and `i2`), where the size of one of the index variables is set to a constant value `s`. The size of the other index variable `i1` is the size of the original index variable `i` divided by the size of the constant-sized index variable, so the product of the new index variable sizes equals the size of the original index variable. There are two alternative versions of split, chosen among by specifying a direction `d`: split up for a constant-sized outer loop and split down for a constant-sized inner loop. Note that in the generated code, when the size of the constant-sized index variable does not perfectly divide the original index variable, a *tail strategy* is employed such as emitting a variable-sized loop that handles remaining iterations.

Preconditions. The constant factor `s` must be a positive non-zero integer.

Precompute and Reorder. The `precompute` transformation allows us to leverage scratchpad memories, and `reorder(IndexVar i1, IndexVar i2)` swaps two directly nested index variables (`i1` and `i2`) in an iteration graph to reorder computations and increase locality. Refer to the work of Kjolstad et al. [2019] for a full description and preconditions.

Preconditions. The precondition of a `reorder` transformation is that it must not move any operation outside or inside of a reduction when the operation does not distribute over the reduction. Otherwise, this will alter the contents of a reduction and change the value of the result. For example, $\forall_i c(i) = b(i) + \sum_j A(i,j)$ can not be reordered to $\forall_j \forall_i c(i) += b(i) + A(i,j)$ because the addition

of $b(i)$ does not distribute over the reduction. In addition, we check that the result of the reorder transformation does not cause tensors to be iterated out of order; certain sparse data formats can only be accessed in a given mode ordering and we verify that this ordering is preserved after the reorder.

Bound and Unroll. The bound and `unroll` transformations apply to only one index variable and tag it with information that instructs the code generation machinery of how to lower it to code. The `bound(IndexVar i, BoundType type, size_t bound)` transformation fixes the range of an index variable `i`, which lets the code generator insert constants into the code and enables other transformations that require fixed size loops, such as vectorization. By passing a different `type`, bound also allows specifying constraints on the end points of an index variable range as well as if the size of a range is divisible by a constant bound. The `unroll(IndexVar i, size_t unrollFactor)` transformation tags an index variable `i` to result in unrolling a loop `unrollFactor` times.

Preconditions. All provided constraints to a bound transformation must hold for inputs of the generated code. The `unrollFactor` must be a positive non-zero integer.

Parallelize. The `parallelize(IndexVar i, ParallelUnit pu, OutputRaceStrategy rs)` transformation tags an index variable `i` for parallel execution. The transformation takes as an argument the type of parallel hardware `pu` to execute on. The set of parallel hardware is extensible, and our code generation algorithm supports SIMD vector units, CPU threads, GPU thread blocks, GPU warps, and individual GPU threads. Parallelizing the iteration over an index variable changes the iteration order of the loop and therefore requires all reductions inside the iteration space represented by the subtree rooted at this index variable to be associative. Furthermore, if the computation uses a reduction strategy that does not preserve the order, such as atomic instructions, then the reductions must also be commutative.

Preconditions. The compiler checks preconditions before each transformation is applied, but the `parallelize` transformation has the most complex preconditions. Once a `parallelize` transformation is used, no other transformations may be applied on the iteration graph as other transformations assume serial code. There are also hardware-specific rules. On a GPU, for example, a CUDA warp has a fixed number of threads. These rules are checked in the hardware-specific backend, rather than before the transformation. Preconditions related to coiteration apply for all hardware: an index variable that indexes into multiple sparse data structures cannot be parallelized as this iteration space will produce a while loop. This loop can instead be parallelized by first strip-mining it with the `split` transformation to create a parallel for loop with a serial nested while loop. Also, expressions that have an output in a format that does not support random insert (such as CSR) cannot be parallelized. Parallelizing these expressions would require creating multiple copies of a data structure and then merging them, which is left as future work. Note that there is a special case where the output's sparsity pattern is the same as the entire input iteration space, such as sampled dense-dense matrix multiplication (SDDMM), tensor times vector (TTV), and tensor times matrix (TTM) expressions for certain combinations of tensor formats. We detect and support this case, and our compiler can thus generate code for these important kernels.

There are also preconditions related to data races during reductions. The `parallelize` transformation allows for supplying a strategy `rs` to handle these data races. The `NoRaces` strategy has the precondition that there can be no reductions in the computation. The `IgnoreRaces` strategy has the precondition that for the given inputs the code generator can assume that no data races will occur. For all other strategies other than `Atomics`, there is the precondition that the racing reduction must be over the index variable being parallelized.

5 CODE GENERATION

From the transformed iteration spaces, we can generate code for either CPUs or GPUs, including CUDA GPU code with warp and thread-level parallelism and CPU code that exploits SIMD vectorization and OpenMP parallelization. To support derived iteration space code generation, we extended the sparse tensor algebra code generation algorithm of Kjolstad et al. [2017b]. Details of these modifications can be found in Senanayake [2020] and a complete code generation algorithm can be found in Kjolstad [2020]. The extensions introduced in Senanayake [2020] allow for the following new capabilities:

(1) recover coordinates in the original iteration space from the transformed iteration space,
(2) determine derived index variable bounds from non-derived index variable bounds,
(3) generate iteration guards to prevent invalid data accesses,
(4) coiterate over tiled and collapsed iteration spaces, and
(5) generate parallel code for GPUs and vectorized parallel code for CPUs.

For context, we provide an overview of the code generation algorithm of Kjolstad et al. [2017b]. For more information, we refer the reader to their paper. Their code generator operates on iteration graphs and generates nested loops to iterate over each index variable. For each index variable, it generates one or more loops to either iterate over a full dimension (a dense loop) or to coiterate over levels of one or more coordinate hierarchy data structures. Coiteration code is generated using a construct called a merge lattice that enumerates the intersections that must be covered to iterate over the sparse domain of the dimension. This may result in a single for loop, a single while loop, or multiple while loops.

5.1 Coordinate Recovery

When iterating over a transformed iteration space, it is necessary to recover index variables in the original iteration space: the original index variables appear in the coordinate hierarchy data structures of tensors, whereas the derived index variables represent the dimensions in the transformed iteration space and guide code generation. Generated loops iterate over the coordinate ranges defined by these derived index variables, but tensors must be accessed by the coordinates in the original iteration space.

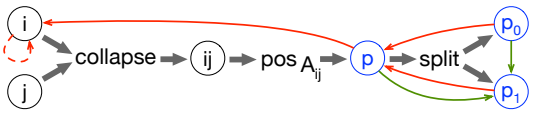


Fig. 17. An index variable provenance graph annotated with arrows that depict different ways that an unknown index variable's coordinates can be recovered from known index variables. Red arrows depict original coordinate recovery and green arrows derived coordinate recovery.

The code generator must therefore emit code to map between these iteration spaces. We call this mapping coordinate recovery. It may be necessary to recover original or derived coordinates. Recovering original coordinates is required when they are used to index into tensor data structures. And derived coordinates must be computed when a coordinate in the original iteration space is loaded from a coordinate hierarchy, but a coordinate in the derived space is needed to determine iteration guard exit conditions.

The code generator defines two functions on the provenance graph to map coordinates between original and derived index variables (Figure 18). These are:

recover_original which computes the coordinate of an index variable from its derived index variables in a provenance graph (red arrows in Figure 17), and
recover_derived which computes the coordinate of an index variable from the variable it derives from and its siblings (green arrows in Figure 17).

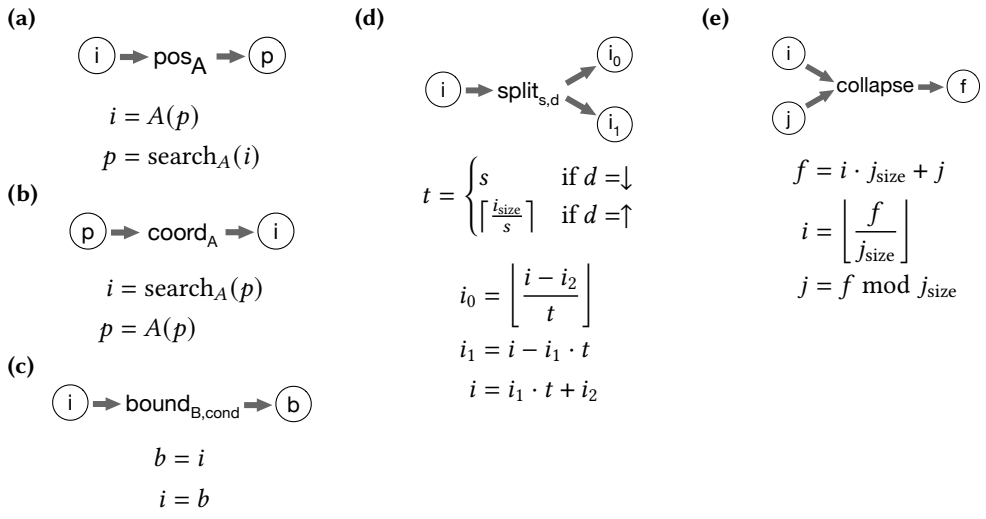


Fig. 18. We show how we can recover the value of any unknown variable if the values for the other variables in the relationship are known. These relationships are defined per transformation and exposed as **recover_original** and **recover_derived** functions. Some recovery functions require additional transformation context. **(a-b)** require the tensor that can be indexed or searched to map between the given position space and coordinate space. **(d)** requires the split factor (s) and the split direction (d).

Coordinate recovery may require an expensive search, so we define a coordinate tracking optimization that computes the next coordinate faster than computing an arbitrary coordinate. Coordinate tracking code implements iteration through recovered coordinates and has two parts: an initialization that finds the first coordinate and tracking that advances it. The initialization is done with the following code generation function on provenance graphs:

recover_track which computes the next coordinate (red stippled arrow in Figure 17).

The SpMV example in Section 2 uses the tracking optimization to track the row coordinate. It starts by finding the first row coordinate using a binary search, and it then simply advances to the next row coordinate when it finds the end of a row segment, using a while loop to move past empty segments.

5.2 Derived Bounds Propagation

To determine the iteration domain of each derived index variable, we propagate bounds through the index variable provenance graph (Figure 9). We have defined propagation rules for each transformation (Figure 19) and calculating the iteration domain of the derived index variables involves applying the propagation rules to each arrow in turn, from the original index variables to the derived index variables.

5.3 Iteration Guards

Unlike the coordinate tree iteration pattern generated for default-scheduled computations, tiled iteration and position space iteration cannot simply perform a computation at every coiterated point. When a loop is tiled with a tile size that does not evenly divide the total number of iterations, additional iterations may occur outside the bounds of the tile or the data structure. Such iterations can lead to incorrectly computed results or cause the program to crash due to a segmentation fault.

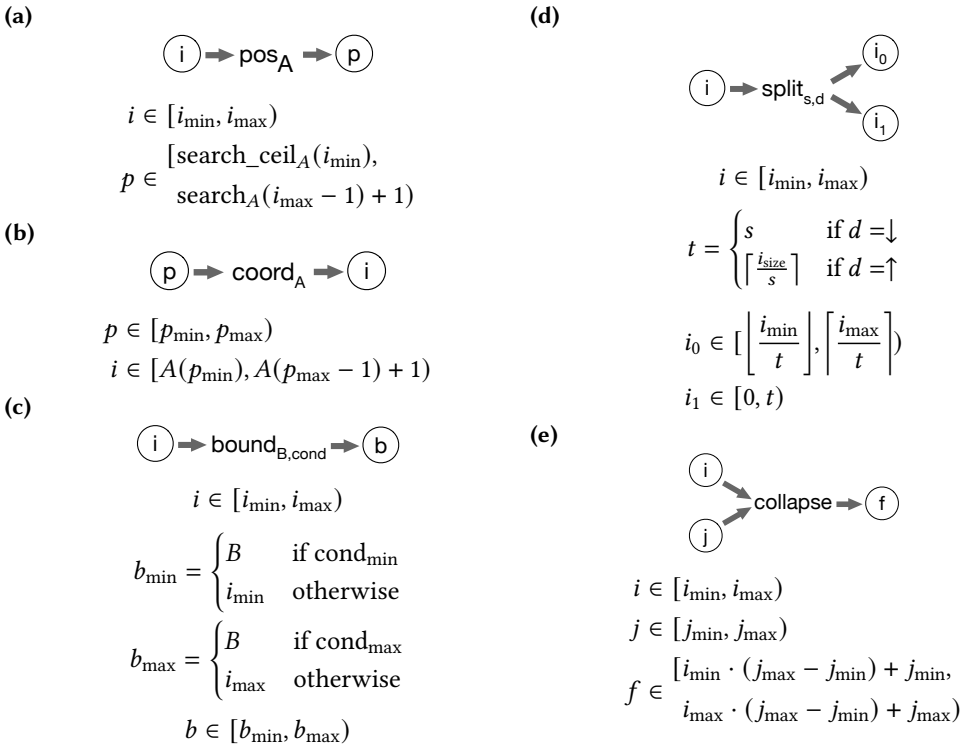


Fig. 19. The provided bound propagation functions show how the bounds of derived index variables are computed from the bounds of the original index variables provided to the transformation. Some bound propagation functions require additional transformation context. **(a-b)** require the tensor that can be indexed or searched to map between the given position space and coordinate space. **(c)** requires the provided bound (B) and the condition (cond) of when the bound should apply. **(d)** requires the split factor (s) and the split direction (d).

Furthermore, during position space iteration, it is necessary to determine when subtrees of the coordinate tree have been fully iterated to prevent reducing values into the wrong location.

To prevent incorrect behavior, we generate iteration guards. Tiled iteration guards cause an early exit of the loop and guard against invalid data access. Position space iteration guards are generated depending on the format of the coordinate tree being iterated. They are used to determine when a subtree has ended and properly handle updating upper-level coordinate values, writing out results, and resetting temporaries. The compiler generates these guards using the same process as generating loop bounds for a normal coordinate tree iteration. The process for generating writes is also straightforward: results are written only when the guard condition evaluates true.

To determine if an index variable can become out-of-bounds, we start with the fully derived variables and work backwards through the provenance graph to determine this quality for all index variables. Fully derived variables are guaranteed to be in-bounds as their bounds appear in the bounds of loops generated in Section 5.2. However, the parents of a `split` transformation lose this quality and are marked as possibly being out-of-bounds. If the bound transformation supplies a constraint that guarantees that the parent of a `split` transformation has a range that is evenly divided by the tile size, then they are also guaranteed to be in-bounds. The parent of a `pos` or `coord`

transformation is guaranteed to be in-bounds, even if it has a child that is out-of-bounds. This is because a guard will always be inserted to ensure that a data structure is not indexed or searched with an out-of-bounds value, and therefore the value being read from a data structure must be valid and in-bounds. All other transformations will propagate the guarantee or lack-of-guarantee of being in-bounds from child index variables to parents.

5.4 Merge Lattice Construction

To coiterate over tiled and collapsed iteration spaces, we extend the merge lattice construction algorithm described in [Kjolstad et al. 2017b]. A full description and examples are provided in Kjolstad [2020]. Tiled iteration graphs contain two types of nodes: those that iterate over an integer range and those that coiterate over data structures. The new merge lattice construction algorithm uses a provenance graph to determine the type of each node, rather than assuming that every node in an iteration graph performs a full coiteration of all data structures defined on the given dimension. If an iteration graph node iterates over an integer range, it produces the single-point merge lattice that results in a simple for loop over the given index variable. When constructing a merge lattice for the coiteration over data structures, the algorithm adds an additional intersecting data structure during construction, which is defined on values within the segment and zero otherwise. This data structure is used only to bound the iteration and is not itself coiterated.

When coiterating over a collapsed dimension, the new merge lattice construction algorithm constructs new iterators for the collapsed dimension. These new iterators are equivalent to the iterators for the deepest nested dimension, with the exception that they are not nested within the iterator of the outer dimension. These collapsed iterators iterate through an entire level of a tensor rather than one segment of the level, but they skip the outer dimension in this chain. The values for the skipped iterators are recovered using the position space iteration strategy. This construction allows us to merge data structures on collapsed dimensions just as we would on a single dimension.

5.5 Parallel and GPU Code Generation

Parallelization and vectorization are applied to the high-level iteration graph IR, so can be assured without heavy analysis. A parallelization strategy is tagged onto an index variable and the code generator generates parallel constructs from it, whether SIMD vector instructions, a parallel OpenMP loop, or a GPU thread array. The parallelization command can easily be extended with other parallelization strategies, and in this context parallel code generators are easy to write: they mechanically translate from the iteration graph and need not perform target-specific optimizations.

Parallel code generators must also generate code that safely manages parallel reductions. We have implemented two strategies: the first strategy detects data races, by inspecting whether a reduction is dominated by an index variable that is summed over, and inserts an atomic instruction at the appropriate location. The second strategy separates the loop into worker and reduction loops that communicate through a workspace [Kjolstad et al. 2019] (e.g., a dense vector stored as a dense array). The threads in the parallel loop reduce into separate parts of the workspace, and when they finish, the second loop reduces across the workspace, either sequentially or in parallel; this strategy mimics the Halide `rfactor` construct [Suriana et al. 2017]. We have implemented this strategy on CPUs using SIMD instructions, and on GPUs with CUDA warp-level reduction primitives. It is also possible to control the reduction strategy at each level of parallelism to optimize for different levels of parallel hardware. For example, on a GPU we can choose a loop separation strategy within a warp and atomics across warps.

6 HEURISTICS-BASED AUTOSCHEDULING

The TACO scheduling language can lead to tens of thousands of semantically-correct schedules even for the simplest expressions, as with Halide [Adams et al. 2019; Ragan-Kelley et al. 2013]. In this section, we describe an automated system based on heuristics to prune this massive search space given a particular tensor algebra expression. Our system does not produce a single best schedule for an input expression; instead, it generates a list of promising *schedule templates*. Templates consist of a sequence of scheduling commands without fixed split sizes or unroll factors. These templates provide different strategies for a user to consider or a system to search. The optimal set of parameters for a template can be determined separately.

For computations that will be executed a large number of times, exhaustively searching over a large set of parameter settings for all templates may be possible, but for a small number of executions, users may opt to pick a single template with user-provided parameters. To make searching through these templates tractable, we focus on restricting the size of this search space to a small number of templates with a high likelihood of achieving good performance.

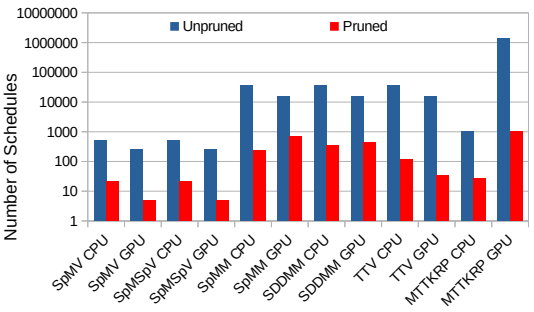


Fig. 20. The number of schedule templates in the search space before and after applying our pruning heuristics on six sparse tensor algebra expressions on both CPU and GPU that are also evaluated in Section 7. The unpruned search space size does not include multi-level tiling strategies, which can create an unbounded number of schedules. Given that evaluating a schedule template for various parameter choices can take on the order of a minute, this pruning is necessary for quickly tuning many expressions.

Our strategy prunes three kinds of schedule templates from the search space. Invalid schedule templates that do not represent valid transformation strategies are eliminated using the preconditions built into the scheduling language. Redundant schedule templates, which result in the same transformations as some other schedule template can be removed by establishing equivalences between different sequences of scheduling commands. For example, a schedule with two reorders of the same pair of variables is equivalent to the same schedule without these reorders. Finally, inefficient schedule templates are additionally filtered using heuristics based on empirical observations and knowledge of the hardware backend.

We divide the generation of a schedule template into two stages, similar to the phased approach that the Halide auto-scheduler uses to prune the space explored by tree search [Adams et al. 2019]. We first consider useful partition strategies of the iteration space, using the *split*, *pos*, and *collapse* transformations. For each of these strategies, we then consider in the second stage possible reordering and parallelization strategies. Currently, the auto-scheduler does not automatically apply the `precompute` transformation, but users can manually supply precomputed index expressions according to the heuristics described in Kjolstad et al. [2019].

We can constrain the total number of generated schedule templates by reducing the outputs at each of the two stages to remove invalid, redundant, and inefficient schedule templates. Similar to the Halide auto-scheduler [Adams et al. 2019], we constrain the search space to a large set of candidate schedules, but our heuristics for inefficiency prevent us from exploring all valid schedules. We now describe each of the stages, using SpMM, $C_{ik} = \sum_j A_{ij} B_{jk}$, as a running example.

6.1 Partition

The first stage aims to remove redundant and invalid partitions of the iteration space caused by combinations of `collapse`, `split`, and `pos`. We also prune combinations that almost always perform worse than a schedule template that is already included in the search space. Decisions made at this stage define the loops of the final program, though the order of the loops can still be permuted in the second stage. The auto-scheduler first determines the search space of collapsed variables. A sparse tensor algebra expression typically consists of both dense and compressed index variables. The auto-scheduler disallows collapsing two dimensions, if the inner dimension is dense. From experience, this is only beneficial for matrices with small outer dimensions that provide insufficient parallelism or when operating on certain blocked sparse formats. In the SpMM example, these heuristics lead us to only explore `collapse(i,j)` and the empty schedule.

After deciding what variables to collapse, we are left with a list of index variables that can now be partitoned individually. By default, we exclude schedules that split an index variable twice, in effect disabling the consideration of all multi-level tiling strategies. Multi-level tiling strategies tend to be less effective on most sparse tensor algebra problems due to the overhead of restarting iteration. However, the user can choose to enable multiple levels of splits at the cost of increasing the number of generated schedule templates. We also always split collapsed dimensions to avoid redundancy with non-collapsed schedule templates. We do not explore splits on compressed dimensions in coordinate space as the additional search overhead required compared to a position space split tends to lead to worse performance except for cases where different sparse data structures have very different and skewed distributions of nonzeros over coordinate space. This means that we also never explore schedule templates with the `coord` transformation. In our SpMM example, if the auto-scheduler applies `collapse(i,j,f)`, then applying `split` offers two choices: split f in position space and k in coordinate space and split only f in position space.

6.2 Reordering and Parallelization

We next determine the iteration order over the index variables derived from partitioning and decide which index variables to parallelize. Iteration order and parallelization are closely intertwined and thus the auto-scheduler considers them jointly. We consider all valid topological orderings of the index variables based on the constraints defined by efficient iteration orders for sparse data structures. This can lead to an exponential relationship between the number of index variables and the number of explored schedules. In practice, however, most index variables are heavily constrained unless a multi-level tiling strategy is explored. In addition to iteration order, we also consider variables to parallelize over threads and vector units on CPU and over blocks, warps, and threads on GPU.

On CPUs, we use the following heuristics to reduce the number of generated schedule templates:

(1) Allow only the outermost index variable to be parallelized, which cannot be a collapsed variable. This is due to the limited parallelism needed on CPUs.
(2) Disallow redundant schedule templates that `split`, which does not change the iteration order by itself, without a subsequent `reorder` or `parallelize`.
(3) Disallow schedule templates that iterate over a sparse tensor out-of-order. Many common sparse tensor formats, such as CSR, only support efficient iteration for certain dimension-orderings. To avoid asymptotic slowdowns, we consider only valid topological orderings of these iteration dependencies. Additionally, if a given partitioning allows iterating over a dense tensor in order of its layout in memory, then we exclude schedules that iterate out-of-order.

On GPUs, we use constraints 2 and 3 from above, as well as three additional restrictions:

(1) Disallow all schedule templates that are not load balanced, as on GPU, it is very beneficial for all parallel units to do the same amount of work due to the large number of parallel threads. We only consider schedule templates that assign an equal amount of work to all threads.
(2) Disallow schedule templates that parallelize a non-constant-sized index variable over threads or warps as these schedules are invalid by the preconditions of the GPU backend.
(3) Allow only schedules with efficient global memory accesses. GPU threads must iterate over data with a stride of the warp size to efficiently utilize DRAM bandwidth. If a thread instead iterates over a contiguous piece of data, its size must be statically known and small enough to fit the limited L1 cache size.

Our restrictions successfully cut down the number of schedule templates that the auto-scheduler considers. Figure 20 shows the reduction in schedule templates for the six different sparse tensor algebra problems we use in our evaluation in the next section. In most cases, we achieve two to three orders of magnitude reduction in the size of the search space of schedule templates. Notably, for MTTKRP on GPU, we reduce the search space size from more than one million to around one thousand. Assuming evaluating all the parameter choices for a schedule template takes one minute, an exhaustive search over one million schedule templates would have taken around two years while now the search can finish in around half a day.

We use the schedules generated by our auto-scheduler as a starting point in constructing schedules for the comparative performance evaluation in Section 7.2.

7 EVALUATION

We carry out experiments to compare the performance of code generated by our technique against state-of-the-art library implementations of a wide variety of sparse linear and tensor algebra kernels. We show how the transformations that our scheduling language exposes are general and can be combined in different ways to optimize the performance of different kernels. We then carry out additional studies to highlight situations where the best schedules differ depending on the situation. We show, for example, that the best schedules for CPU and GPU differ for the same computations, and that the best GPU SpMV schedule depends on whether the computation is load-balanced.

7.1 Methodology

We implement the transformation framework as an extension to the TACO compiler, which is freely available under the MIT license. To evaluate it, we compare the performance of code that has been optimized using the introduced transformations on CPU to Intel MKL 2020 [Intel 2012], Eigen 3.3.7 [Guennebaud et al. 2010], SPLATT 1.1.1 [Smith et al. 2015], and the original TACO system (commit 331188). On GPUs, we compare to cuSPARSE v9.0 [NVIDIA V10.1.243 2019], the Merge-Based SpMV implementation of Merrill and Garland [2016], the ASpT CSR SDDMM implementation of Hong et al. [2019], and hand-optimized GPU MTTKRP kernels (B-CSF, COO, HCSR, and HYB) presented by Nisa et al. [2019].

For the comparative studies, we use all real-valued matrices from the SuiteSparse sparse matrix repository [Davis and Hu 2011], tensors from the Formidable Repository of Open Sparse Tensors and Tools (FROSTT) [Smith et al. 2017], and tensors constructed from the 1998 DARPA Intrusion Detection Evaluation Dataset and the Freebase dataset [Jeon et al. 2015]. We exclude matrices and tensors that do not fit in memory on the machine used to run the experiments. We generate random dense tensors and compressed vectors as required by the expression. Compressed vectors are generated with a sparsity of 10% and dimensions that are not fixed by the size of the dataset tensor are set to 128. To store sparse matrices, we use the standard compressed sparse row (CSR) format for SpMV, SpMM, and SDDMM and, for work-efficient execution, the compressed sparse

column (CSC) format for SpMSpV. To store sparse tensors, we use the compressed sparse fiber (CSF) format [Smith and Karypis 2015].

All CPU experiments are run on a dual-socket, 12-core with 24 threads, 2.5 GHz Intel Xeon E5-2680 v3 machine with 30 MB of L3 cache per socket and 128 GB of main memory, running Ubuntu 18.04.3 LTS. On CPU, we compile code that our technique generates using Intel icpc 19.1.0.166 with `-O3`, `-DNDEBUG`, `-march=native`, `-mtune=native`,`-ffast-math`, and `-fopenmp`. We run CPU experiments with a cold cache 25 times and report median execution times recorded with `std::chrono::high_resolution_clock`.

All GPU experiments are run on an NVIDIA V100 GPU, which has 80 streaming multiprocessors (SM) with 32 GB of global memory, 6 MB of L2 cache and 128 KB of L1 cache per SM, and a bandwidth of 897 GB/s. We compile the generated code with NVIDIA nvcc 9.0.176 with `-O3`, `-gencode arch=compute_70,code=sm_70`, and `-use_fast_math`. We run GPU experiments 25 times and report median execution times recorded with CUDA events. We exclude the time of copying and allocating memory and freshly copy all data to the device for each trial.

7.2 Comparative Performance

We measure and compare the performance of code that our technique generates against hand-implemented versions in other libraries for six sparse linear and tensor algebra kernels:

SpMV: $y_i = \sum_j \boldsymbol{A}_{ij} x_j$ (A.1 and A.2)
SpMSpV: $y_i = \sum_j \boldsymbol{A}_{ij} \boldsymbol{x}_j$ (A.3 and A.4)
SpMM: $Y_{ij} = \sum_j \boldsymbol{A}_{ik} X_{kj}$ (A.5 and A.6)
SDDMM: $\boldsymbol{A}_{ij} = \boldsymbol{B}_{ij} \sum_k C_{ik} D_{kj}$ (A.7 and A.8)
TTV: $\boldsymbol{A}_{ij} = \sum_k \boldsymbol{B}_{ijk} c_k$ (A.9 and A.10)
MTTKRP: $A_{ij} = \sum_{kl} \boldsymbol{B}_{ikl} C_{kj} D_{lj}$ (A.11–A.14)

Operands and results highlighted in bold are sparse, while all others are dense. For MTTKRP CPU, we also evaluate kernels for order-4 and order-5 tensors, these expressions are in A.12 and A.13. For all kernels other than SpMSpV, we start with a schedule template discovered by the heuristic-based autoscheduler described in Section 6. For SpMM CPU, we modify a `reorder` command and add a `parallelize` command to vectorize a loop. For MTTKRP GPU, we modify a `reorder` command to enforce a better memory access pattern. For SpMSpV GPU, we elect to use an alternative kernel that is not strictly load-balanced like the generated schedule templates. The schedules we use to generate each kernel for CPU and GPU are identified in parentheses and can be found in the referenced appendices.

Figures 21–32 show the results of our experiments for each kernel and test matrix/tensor. For each kernel, we additionally report scheduled TACO's geometric mean speedup across all test matrices/tensors relative to the fastest equivalent alternative implementation (either hand-optimized or generated by original TACO) where one exists:

	SpMV	SpMSpV	SpMM	SDDMM	TTV	MTTKRP
CPU	1.03×	2.45×	0.99×	1.02×	1.93×	0.96×
GPU	1.00×	n/a	0.59×	0.65×	n/a	0.90×

Our technique generates code that has comparable performance with hand-optimized libraries for a wide range of sparse tensor algebra expressions with varied characteristics. For SPMM on GPUs, cuSPARSE is able to outperform scheduled TACO by better utilizing shared memory, which scheduled TACO currently has limited support for. Even though scheduled TACO exhibits worse average performance for GPU SpMM, it is nevertheless competitive with state-of-the-art. On the whole, these results demonstrate the generality of our technique; its performance not limited to any specific kernel.

For each CPU kernel, Figures 21–32 also show results for original TACO, which uses only the default schedule, in addition to scheduled TACO, which for each matrix uses the faster of either

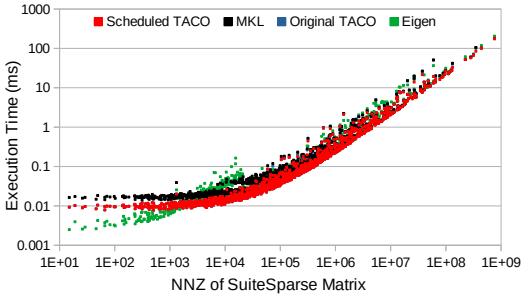


Fig. 21. Performance of CPU SpMV.

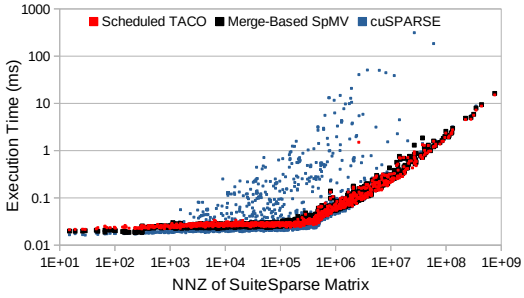


Fig. 22. Performance of GPU SpMV.

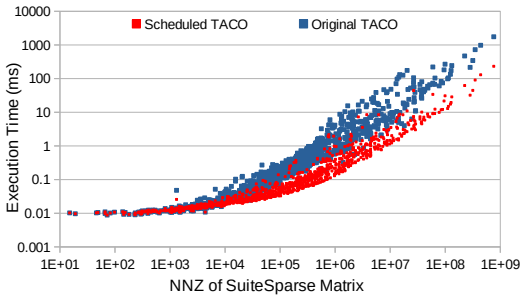


Fig. 23. Performance of CPU SpMSpV.

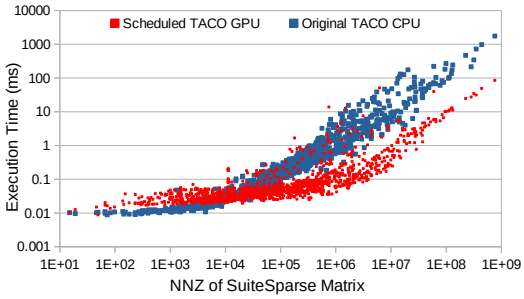


Fig. 24. Performance of GPU SpMSpV.

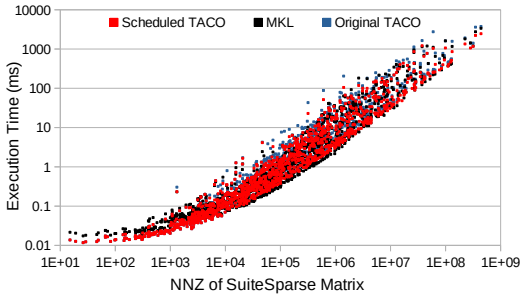


Fig. 25. Performance of CPU SpMM.

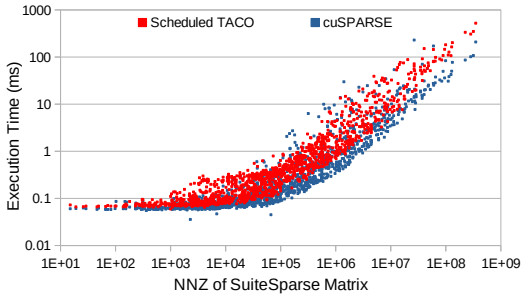


Fig. 26. Performance of GPU SpMM.

the default schedule or the previously-referenced custom schedule The flexibility of our scheduling language makes it easy to selectively optimize kernels based on the inputs, which is essential as the default schedule generally performs well for small matrices and tensors. For instance, the SpMM CPU schedule provided in Appendix A.5 outperforms the default for 66% of all test matrices and is faster by 29.2% on average for those typically larger matrices, but is slower for the remaining, typically smaller matrices. Nevertheless, by using the custom schedule for only the aforementioned 66% of matrices, scheduled TACO is able to achieve a mean speedup of 16.7% over original TACO across all test matrices. Furthermore, for larger matrices (with more than a million nonzeros) and tensors, GPU kernels generated by scheduled TACO all significantly outperform equivalent CPU kernels generated by original TACO, which does not support GPU code generation. For smaller matrices though, the GPU kernels do not have enough work to fully utilize available GPU resources, and thus their equivalent CPU kernels actually offer better performance. This demonstrates the benefit of being able to selectively generate efficient code for different platforms.

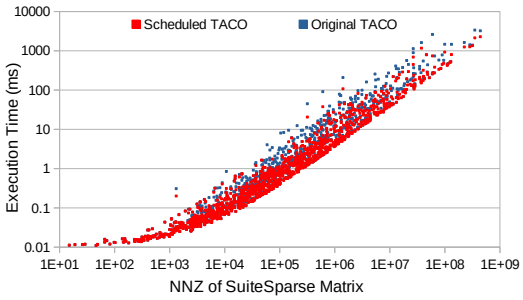


Fig. 27. Performance of CPU SDDMM.

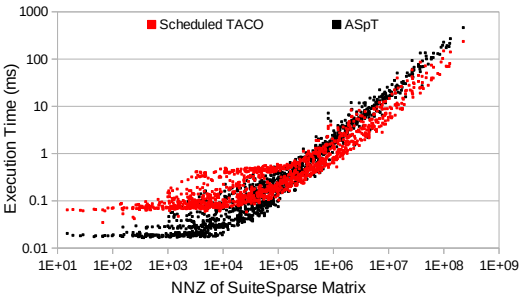


Fig. 28. Performance of GPU SDDMM.

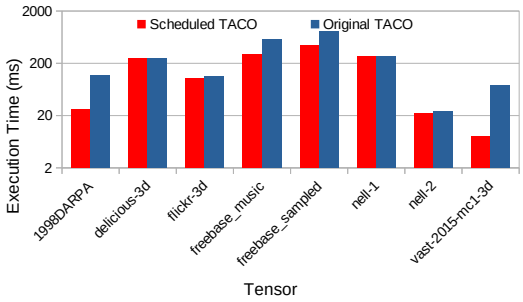


Fig. 29. Performance of CPU TTV.

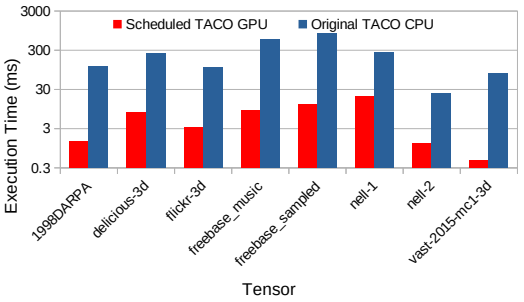


Fig. 30. Performance of GPU TTV.

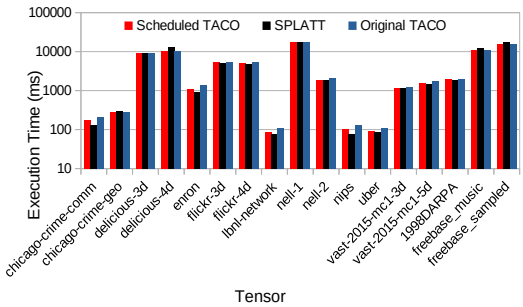


Fig. 31. Performance of CPU MTTKRP.

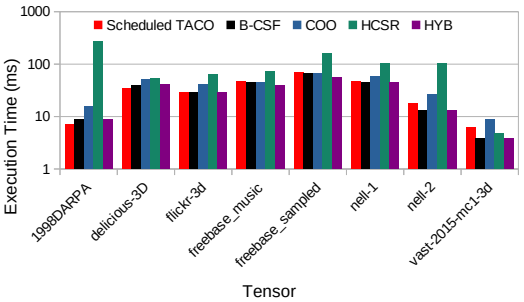


Fig. 32. Performance of GPU MTTKRP.

7.3 Scheduling for GPUs

Good GPU schedules are different from good CPU schedules, and it is important to have transformations that let us order operations to fit the machine at hand. GPUs are sensitive to load-balanced execution, and they typically require more involved schedules to ensure operations are done in the right order. For instance, the best parallel CPU SpMV schedule compiled to and executed on a GPU performs 6.9× worse than the warp-per-row GPU schedule on a matrix with four million randomly allocated nonzeros. This illustrates the necessity of our transformations to produce useful GPU code.

The best CPU schedule for the SpMV operation when the matrix is load-balanced is a simple strip-mining of the outer dense loop to create parallel blocks, followed by parallelizing the outer loop. The resulting code assigns a set of rows to each CPU thread executing in parallel. The analogous schedule can be a disaster. Since threads in a warp execute separate rows, they cannot coalesce memory loads. This leads to poor effective memory bandwidth and poor cache utilization, thus

resulting in poor performance for the memory-bound SpMV kernel. Furthermore, if there are a different number of nonzeros on the rows executed by different threads in a warp, then they will experience lower parallelism.

By contrast, more optimized GPU schedules are more carefully tiled. The warp-per-row schedule assigns an equal number of nonzero elements of each row of the sparse matrix to each thread and uses warp-level synchronization primitives to efficiently reduce these partial sums. The optimized SpMV schedule that we use in Figure 22 tiles the position space of the sparse matrix across threads. Tiling in position space maximizes parallelism, as it enables the kernel to utilize most GPU threads. The result, as shown in Figure 22, is that, unlike cuSPARSE, scheduled TACO's performance is not sensitive to the structure of the sparse matrix; the execution time and the number of nonzeros in the sparse matrix are highly correlated. We also use a temporary to allow unrolling the loop that performs loads and then later use atomic instructions to store the results in the output. This provides better memory access patterns and increased instruction-level parallelism, but makes hand-writing such a kernel difficult. On a matrix with four million randomly allocated nonzeros, this increased instruction-level parallelism provides a 36% speedup for our optimized schedule over the same schedule without the temporary or loop unrolling. The schedules with and without the unrolling optimization are shown, respectively, in Appendices A.2 and A.17.

Load-balanced execution is also important for the other GPU kernels. For instance, as shown in Figure 32, the performance of the comparison baselines except COO varies a lot depending on the input tensor; their load-balancing strategies suffer from load-imbalance when a nonzero distribution is very skewed. COO, on the other hand, achieves good performance across different inputs but incurs additional data footprint.

However, the same schedule is not always optimal across different inputs. For example, while tiling in position space resolves the load-balancing issue, since it also allows the same entry of the output to be accessed by multiple threads, atomic operations are needed to avoid race condition. Furthermore, tiling incurs overhead from additional GPU kernel launches, which can dominate the execution time when the workload is not large enough to amortize the overhead [Pai and Pingali 2016]. Thus, for matrices that are relatively small or that have nonzeros evenly distributed, minimizing the number of kernels launched and avoiding atomic operations is more critical for performance. This highlights the need for a scheduling language such as ours, which makes it easy to selectively apply optimizations based on attributes of the inputs.

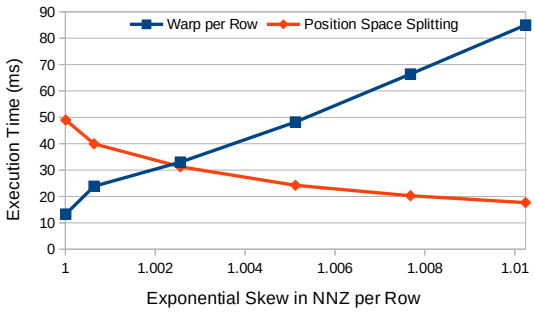


Fig. 33. Execution time of warp-per-row and load-balanced SpMV on matrices fixed number of nonzeros with increasingly skewed distribution. The number of nonzeros in row i is given by the formula $k * c^i$. c appears on the x-axis of the graph (when $c = 1$, all rows have the same number of nonzeros). For every c, we choose k so that the total number of nonzeros is always 400 million. Nonzeros are uniformly random within a row and the rows were shuffled. The schedules that generated the kernels are in Appendices A.15 and A.16.

7.4 Scheduling for Load Balance

This study shows that the best GPU schedules differ for load-balanced and load-imbalanced computations. The SpMV computation demonstrates the issue, as it is sensitive to a skewed distribution of nonzeros in the matrix. The challenge, however, generalizes to any expression with a sparse tensor.

The previous section outlined an effective warp-per-row GPU SpMV schedule where threads in a warp collectively work on a matrix row at a time. If distribution of nonzeros across matrix

rows is skewed, this kernel suffers from load imbalance. The optimized SpMV schedule used in Figure 22 where the two loops are fused and then split in the position space provides perfect static load balancing for loads of the sparse tensor values at the cost of overhead from coordinate recovery. Figure 33 shows the performance of the warp-per-row schedule and the load-balanced position split schedule as the distribution of nonzeros per row becomes more skewed according to an exponential function. The number of nonzeros remains fixed and rows are randomly shuffled. As expected, the warp-per-thread schedule performs worse as skew increases, while the load-balanced schedule benefits from long rows and performs better with increased skew. For skewed matrices, the load-balanced kernel is thus preferable.

7.5 Scheduling for Maximal Parallelism

Loop fusion to increase the amount of parallelism, despite higher overhead, can make sense in parallelism-constrained situations. The warp-per-row GPU SpMV schedule described in Section 7.3 assigns each row to be executed by a different warp. For matrices with few rows, however, this results in too little parallelism to occupy the GPU. For such matrices, fusing before parallelizing the two loops creates more parallelism. For example, we execute the SpMV kernels generated from both schedules on a short and wide $100 \times 100k$ matrix with 10k nonzeros per row. As expected from the experiment design, the warp-per-row kernel has too little parallelism and the fused kernel runs 4.5× faster on average across 10 runs.

8 RELATED WORK

The compiler transformation framework we present in this paper automates the generation of many sparse linear and tensor algebra kernels that performance engineering researchers have previously discovered and coded by hand [Baskaran et al. 2017, 2014; Bell and Garland 2009; Intel 2012; Smith et al. 2015; Yang et al. 2018]. We designed our transformations so that they can recreate these kernels, with the notable exception of the MergeSpMV kernel by Merrill and Garland [2016], which requires extensions to our approach. Our transformations, however, can be composed to create many other optimized versions of each kernel and, most importantly, generalize to any tensor algebra expression.

The sparse iteration spaces that are transformed in this paper can be compared to the polyhedral model view of iteration spaces [Allen and Cocke 1972; Ancourt and Irigoin 1991; Feautrier 1988]. In the polyhedral model, an iteration space is defined as a set of affine inequalities where each point inside the space is included. The sparse iteration spaces, on the other hand, are hyperrectangles where only an irregular subset of the points are included. This subset is described by intersections and unions of the coordinates stored in compressed (tensor) data structures. And the array accesses are affine expressions of the positions and coordinates stored in these data structures. Finally, since we limit ourselves to compiling sparse tensor algebra, the only dependencies—reductions and data structure dependencies across nested loops—are respected by construction.

We will compare and contrast our techniques to two bodies of prior work: work on compiler optimizations for sparse loop nests and work on optimizations for dense (i.e., affine) loop nests.

Sparse Linear and Tensor Algebra Optimizations

The first work on compiler optimization of sparse loops, defined here as loops whose iteration domains are not contiguous, was done by Bik and Wijshoff [1993] and was followed by many other lines of work: Bernoulli [Kotlyar et al. 1997], SIPR [Pugh and Shpeisman 1999], the OpenMP Stream Optimizer [Lee et al. 2009], and the SPF [Strout et al. 2018] and CHiLL-I/E compilers [Venkat et al. 2015]. These start with dense or sparse imperative code, analyze it, and then apply optimizations. In

contrast, the work in this paper views tensor algebra as the programming language to be compiled, lowers it to sparse iteration spaces, optimizes them, and then generates sparse code.

In terms of optimization capabilities, the prior work falls into two categories: source code rewrites and iteration space transformations. Lee et al. [2009] describe the OpenMP Stream Optimizer which supports a limited form of the collapse transformation using source code rewrites. Specifically, their compiler can collapse an outer dense loop and an inner sparse loop that iterates over a single operand, meaning it implements a tensor multiplication with a single sparse operand (e.g., SpMV). The work in this paper can collapse, split, and reorder iteration spaces induced by any combination of multiplications and additions over any number of dense and sparse data structures.

The Bernoulli project [Kotlyar et al. 1997] developed a compiler that lifted out relational descriptions of the iteration spaces of dense loops, as a cross product of each loop's domain. This novel view of iteration spaces let them insert sparsity filters and then pushed those filters into the cross products to produce relational joins that describe sparse iteration spaces. Their optimizations were limited, however, to selecting data structures and they did not have the iteration space optimizations we describe in this paper.

The CHiLL-I/E [Venkat et al. 2015] and the Sparse Polyhedral Framework [Strout et al. 2018] are related compiler approaches that analyze and transform dense loops to sparse counterparts with the aid of inspector-executor techniques. They then lift the loops into the polyhedral model, using the idea of uninterpreted functions [Wonnacott and Pugh 1995] to model loops over a single sparse operand as a dense dimension with unknown bounds, which in the terminology of this paper would be a single loop in position space. As a result, they can apply polyhedral optimization to the loop nest. Since neither the polyhedral model nor their extension can model iteration space dimensions that are set combinations of multiple data structures, their approach is limited to tensor multiplications with a single sparse operand. The work in this paper, by contrast, models iteration space dimensions as arbitrary set expressions, and can therefore perform optimizing transformations on any combination of multiplications and additions over any number of dense and sparse data structures.

Finally, the TACO compiler [Kjolstad et al. 2017b] is a sparse tensor algebra compiler that can compile a basic tensor algebra expression with any number of additions and multiplications, where operands are stored in many different data structures [Chou et al. 2018], to fused sparse code. At its core is the iteration graph representation from which TACO generates fast sparse code. Furthermore, Kjolstad et al. [2019] described the precompute and reorder transformations within the TACO framework as well as the concrete index notation representation. Their work did not, however, include the split and collapse transformations we describe in this paper. These transformations are key to tiling strategies for data locality, parallelization, vectorization, and to fast execution on GPUs. In this paper, we add these transformations, the enabling concepts of derived index variables and provenance graphs, a GPU backend, and scheduling primitives for vectorization and parallelization.

Dense Linear and Tensor Algebra Optimizations

There is a long history of work on compiler optimization of dense loops. We discuss two lines of work that are of particular relevance for the work in this paper, namely dense tensor algebra compilers and scheduling languages.

Recent work on dense tensor algebra has focused on two application areas: quantum chemistry and machine learning. While the two areas share some similarities, different types of tensors and operations are important in each domain. The Tensor Contraction Engine [Auer et al. 2006] automatically optimizes dense tensor contractions, and is developed primarily for chemistry applications. Libtensor [Epifanovsky et al. 2013] and CTF [Solomonik et al. 2014] are cast tensor contractions as matrix multiplication by transposing tensors. In machine learning, TensorFlow [Abadi et al. 2016]

and other frameworks [Jia et al. 2014; Paszke et al. 2017] combine tensor operations to efficiently apply gradient descent for learning, and are among the most popular packages used for deep learning. TVM [Chen et al. 2018] takes this further by adopting and modifying Halide's scheduling language to make it possible for machine learning practitioners to control schedules for dense tensor computations. Tensor Comprehensions (TC) [Vasilache et al. 2018] is another framework for defining new deep learning building blocks over tensors, utilizing the polyhedral model.

Scheduling languages have become a staple of domain-specific programming models for dense computations [Baghdadi et al. 2019] and graph algorithms [Zhang et al. 2018] following the CHiLL [Chen et al. 2008] and Halide compilers [Ragan-Kelley et al. 2012]. The idea is to separate the algorithm (what to compute) from the schedule (how to compute it), making the same algorithm run efficiently on different platforms by only changing the schedule. By providing schedules as a clean API, these compilers separate the system that applies transformations (the mechanism) from the system that decides what transformations to apply (the policy). In practice, schedules are often fully or partially hand-specified by performance engineers who want the best performance, although there is much research on automatic scheduling [Adams et al. 2019; Mullapudi et al. 2016]. The work in this paper follows the scheduling language design pioneered by Halide, based on the traditional loop optimizations split, collapse, and reorder, but generalizes those transformations for the first time to sparse iteration spaces that result in general sparse loops.

9 CONCLUSION

This paper presented a comprehensive framework for transformations on sparse iteration spaces. The resulting transformation machinery and code generator can recreate tiled, vectorized, parallelized, and load-balanced CPU and GPU codes from the literature, and generalizes to far more tensor algebra expressions and optimization combinations. Furthermore, as the sparse iteration space transformation machinery works on a high-level intermediate representation that is independent of target code generators, it points towards portable sparse tensor algebra compilation. We believe this work is a first step towards putting sparse tensor algebra on the same optimization and code generation footing as dense tensor algebra and array codes.

ACKNOWLEDGMENTS

We would like to thank our anonymous reviewers, and especially our shepherd, for their valuable comments that helped us improve this manuscript. We also thank Michael Pellauer, Ajay Brahmakshatriya, Michael Garland, Rawn Henry, Suzy Mueller, Peter Ahrens, Joel Emer, and Yunming Zhang for helpful discussion, suggestions, and reviews. Finally, we thank Jessica Shi for adding the scheduling language to the TACO web tool GUI. This work was supported by DARPA under Award Number HR0011-18-3-0007; the Application Driving Architectures (ADA) Research Center, a JUMP Center co-sponsored by SRC and DARPA; the U.S. Department of Energy, Office of Advanced Scientific Computing Research under Award Number DE-SC0018121; the National Science Foundation under Grant No. CCF-1533753; and the Toyota Research Institute. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of the funding agencies.

REFERENCES

Martín Abadi, Paul Barham, Jianmin Chen, Zhifeng Chen, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Geoffrey Irving, Michael Isard, Manjunath Kudlur, Josh Levenberg, Rajat Monga, Sherry Moore, Derek G. Murray, Benoit Steiner, Paul Tucker, Vijay Vasudevan, Pete Warden, Martin Wicke, Yuan Yu, and Xiaoqiang Zheng. 2016. TensorFlow: A System for Large-scale Machine Learning. In *Proceedings of the 12th USENIX Conference on Operating Systems Design and Implementation (OSDI'16)*. USENIX Association, Berkeley, CA, USA, 265–283.

Andrew Adams, Karima Ma, Luke Anderson, Riyadh Baghdadi, Tzu-Mao Li, Michaël Gharbi, Benoit Steiner, Steven Johnson, Kayvon Fatahalian, Frédo Durand, et al. 2019. Learning to optimize halide with tree search and random programs. *ACM Transactions on Graphics (TOG)* 38, 4 (2019), 121.

Frances E. Allen and John Cocke. 1972. A Catalogue of Optimizing Transformations. In *Design and Optimization of Compilers*, R. Rustin (Ed.). Prentice-Hall, Englewood Cliffs, NJ, 1–30.

Corinne Ancourt and François Irigoin. 1991. Scanning polyhedra with DO loops. *Principles and Pratice of Parallel Programming* 26, 7 (April 1991), 39–50. https://doi.org/10.1145/109626.109631

Alexander A. Auer, Gerald Baumgartner, David E. Bernholdt, Alina Bibireata, Venkatesh Choppella, Daniel Cociorva, Xiaoyang Gao, Robert Harrison, Sriram Krishnamoorthy, Sandhya Krishnan, Chi-Chung Lam, Qingda Lu, Marcel Nooijen, Russell Pitzer, J. Ramanujam, P. Sadayappan, and Alexander Sibiryakov. 2006. Automatic code generation for many-body electronic structure methods: the tensor contraction engine. *Molecular Physics* 104, 2 (2006), 211–228. https://doi.org/10.1080/00268970500275780

R. Baghdadi, J. Ray, M. B. Romdhane, E. D. Sozzo, A. Akkas, Y. Zhang, P. Suriana, S. Kamil, and S. Amarasinghe. 2019. Tiramisu: A Polyhedral Compiler for Expressing Fast and Portable Code. In *2019 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*. 193–205. https://doi.org/10.1109/CGO.2019.8661197

Utpal Banerjee. 1990. Unimodular transformations of double loops. Available as Nicolau A., Gelernter D., Gross T., Padua D. (eds) Advances in languages and compilers for parallel computing (1991). The MIT Press, Cambridge, pp 192-219. In *Proceedings of the Workshop on Languages and Compilers for Parallel Computing (LCPC)*.

M. Baskaran, T. Henretty, B. Pradelle, M. H. Langston, D. Bruns-Smith, J. Ezick, and R. Lethin. 2017. Memory-efficient parallel tensor decompositions. In *2017 IEEE High Performance Extreme Computing Conference (HPEC)*. 1–7.

Muthu Baskaran, Benoit Meister, and Richard Lethin. 2014. Low-overhead load-balanced scheduling for sparse tensor computations. In *2014 IEEE High Performance Extreme Computing Conference (HPEC)*. 1–6.

Nathan Bell and Michael Garland. 2009. Implementing Sparse Matrix-Vector Multiplication on Throughput-Oriented Processors. In *International Conference for High Performance Computing, Networking, Storage, and Analysis*. ACM, Portland, Oregon, 18:1–18:11. https://doi.org/10.1145/1654059.1654078

Aart JC Bik and Harry AG Wijshoff. 1993. Compilation techniques for sparse matrix computations. In *Proceedings of the 7th international conference on Supercomputing*. ACM, 416–424.

Chun Chen, Jacqueline Chame, and Mary Hall. 2008. *CHiLL: A framework for composing high-level loop transformations*. Technical Report. University of Southern California. 28 pages.

Tianqi Chen, Thierry Moreau, Ziheng Jiang, Lianmin Zheng, Eddie Yan, Haichen Shen, Meghan Cowan, Leyuan Wang, Yuwei Hu, Luis Ceze, Carlos Guestrin, and Arvind Krishnamurthy. 2018. TVM: An Automated End-to-End Optimizing Compiler for Deep Learning This paper is included in the Proceedings of the. In *Symposium on Operating Systems Design and Implementation*. USENIX Association, Carlsbad, CA, 578–594. https://www.usenix.org/conference/osdi18/presentation/chen

Stephen Chou, Fredrik Kjolstad, and Saman Amarasinghe. 2018. Format Abstraction for Sparse Tensor Algebra Compilers. *Proceedings of the ACM on Programming Languages* 2, OOPSLA (nov 2018), 123:1–123:30. https://doi.org/10.1145/3276493

Stephen Chou, Fredrik Kjolstad, and Saman Amarasinghe. 2020. Automatic Generation of Efficient Sparse Tensor Format Conversion Routines. *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)* (June 2020).

Timothy A. Davis and Yifan Hu. 2011. The University of Florida Sparse Matrix Collection. *ACM Trans. Math. Softw.* 38, 1, Article 1 (Dec. 2011).

Evgeny Epifanovsky, Michael Wormit, Tomasz Kuś, Arie Landau, Dmitry Zuev, Kirill Khistyaev, Prashant Manohar, Ilya Kaliman, Andreas Dreuw, and Anna I. Krylov. 2013. New implementation of high-level correlated methods using a general block tensor library for high-performance electronic structure calculations. *Journal of computational chemistry* 34, 26 (2013), 2293–2309. https://doi.org/10.1002/jcc.23377

Paul Feautrier. 1988. Parametric integer programming. *RAIRO-Operations Research* 22, 3 (1988), 243–268. https://doi.org/10.1051/ro/1988220302431

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

Changwan Hong, Aravind Sukumaran-Rajam, Israt Nisa, Kunal Singh, and P. Sadayappan. 2019. Adaptive Sparse Tiling for Sparse Matrix Multiplication. In *Proceedings of the 24th Symposium on Principles and Practice of Parallel Programming (PPoPP '19)*. Association for Computing Machinery, New York, NY, USA, 300–314. https://doi.org/10.1145/3293883.3295712

Intel. 2012. *Intel math kernel library reference manual.* Technical Report. 630813-051US, 2012. http://software.intel.com/sites/products/documentation/hpc/mkl/mklman/mklman.pdf.

Inah Jeon, Evangelos E. Papalexakis, U Kang, and Christos Faloutsos. 2015. HaTen2: Billion-scale Tensor Decompositions. In *IEEE International Conference on Data Engineering (ICDE)*.

Yangqing Jia, Evan Shelhamer, Jeff Donahue, Sergey Karayev, Jonathan Long, Ross Girshick, Sergio Guadarrama, and Trevor Darrell. 2014. Caffe: Convolutional Architecture for Fast Feature Embedding. *arXiv preprint arXiv:1408.5093* (2014).

Fredrik Kjolstad. 2020. *Sparse Tensor Algebra Compilation.* Ph.D. Dissertation. Massachusetts Institute of Technology.

Fredrik Kjolstad, Peter Ahrens, Shoaib Kamil, and Saman Amarasinghe. 2019. Tensor Algebra Compilation with Workspaces. In *International Symposium on Code Generation and Optimization.* IEEE Press, Washington, DC, 180–192. http://dl.acm.org/citation.cfm?id=3314872.3314894

Fredrik Kjolstad, Stephen Chou, David Lugato, Shoaib Kamil, and Saman Amarasinghe. 2017a. taco: A Tool to Generate Tensor Algebra Kernels. In *Proceedings of the 32nd IEEE/ACM International Conference on Automated Software Engineering.* IEEE Press, 943–948.

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

Vladimir Kotlyar, Keshav Pingali, and Paul Stodghill. 1997. A relational approach to the compilation of sparse matrix programs. In *Euro-Par'97 Parallel Processing.* Springer, 318–327.

Leslie Lamport. 1974. The Parallel Execution of DO loops. *Commun. ACM* 17, 2 (Feb. 1974), 83–93. https://doi.org/10.1145/360827.360844

Seyong Lee, Seung-Jai Min, and Rudolf Eigenmann. 2009. OpenMP to GPGPU: A Compiler Framework for Automatic Translation and Optimization. In *Proceedings of the 14th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP 09).* Association for Computing Machinery, New York, NY, USA, 10. https://doi.org/10.1145/1504176.1504194

Duane Merrill and Michael Garland. 2016. Merge-Based Parallel Sparse Matrix-Vector Multiplication. *International Conference for High Performance Computing, Networking, Storage and Analysis, SC* November (2016). https://doi.org/10.1109/SC.2016.57

Ravi Teja Mullapudi, Andrew Adams, Dillon Sharlet, Jonathan Ragan-Kelley, and Kayvon Fatahalian. 2016. Automatically Scheduling Halide Image Processing Pipelines. *ACM Trans. Graph.* 35, 4, Article 83 (July 2016), 11 pages. https://doi.org/10.1145/2897824.2925952

Israt Nisa, Jiajia Li, Aravind Sukumaran-Rajam, Richard W. Vuduc, and P. Sadayappan. 2019. Load-Balanced Sparse MTTKRP on GPUs. In *2019 IEEE International Parallel and Distributed Processing Symposium, IPDPS 2019, Rio de Janeiro, Brazil, May 20-24, 2019.* 123–133. https://doi.org/10.1109/IPDPS.2019.00023

NVIDIA V10.1.243. 2019. cuSPARSE Software Library.

Sreepathi Pai and Keshav Pingali. 2016. A compiler for throughput optimization of graph algorithms on GPUs. In *Proceedings of the 2016 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications.* 1–19.

Adam Paszke, Sam Gross, Soumith Chintala, Gregory Chanan, Edward Yang, Zachary DeVito, Zeming Lin, Alban Desmaison, Luca Antiga, and Adam Lerer. 2017. Automatic differentiation in PyTorch. (2017).

William Pugh and Tatiana Shpeisman. 1999. SIPR: A new framework for generating efficient code for sparse matrix computations. In *Languages and Compilers for Parallel Computing.* Springer, 213–229.

Jonathan Ragan-Kelley, Andrew Adams, Sylvain Paris, Marc Levoy, Saman Amarasinghe, and Frédo Durand. 2012. Decoupling algorithms from schedules for easy optimization of image processing pipelines. *ACM Transactions on Graphics* 31, 4 (2012), 1–12. https://doi.org/10.1145/2185520.2335383

Jonathan Ragan-Kelley, Connelly Barnes, Andrew Adams, Sylvain Paris, Frédo Durand, and Saman Amarasinghe. 2013. Halide: A Language and Compiler for Optimizing Parallelism, Locality, and Recomputation in Image Processing Pipelines. In *Proceedings of the 34th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '13).* ACM, New York, NY, USA, 519–530. https://doi.org/10.1145/2491956.2462176

Ryan Senanayake. 2020. *A Unified Iteration Space Transformation Framework for Sparse and Dense Tensor Algebra.* M.Eng. Thesis. Massachusetts Institute of Technology, Cambridge, MA. http://groups.csail.mit.edu/commit/papers/2020/ryan_2020.pdf

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/

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.* ACM, 5.

Shaden Smith, Niranjay Ravindran, Nicholas Sidiropoulos, and George Karypis. 2015. SPLATT: Efficient and Parallel Sparse Tensor-Matrix Multiplication. In *IEEE International Parallel and Distributed Processing Symposium.* IEEE, 61–70. https://doi.org/10.1109/IPDPS.2015.27

Edgar Solomonik, Devin Matthews, Jeff R. Hammond, John F. Stanton, and James Demmel. 2014. A massively parallel tensor contraction framework for coupled-cluster computations. *J. Parallel and Distrib. Comput.* 74, 12 (Dec. 2014), 3176–3190.

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

Patricia Suriana, Andrew Adams, and Shoaib Kamil. 2017. Parallel Associative Reductions in Halide. In *Proceedings of the 2017 International Symposium on Code Generation and Optimization (CGO '17)*. IEEE Press, 281–291.

Nicolas Vasilache, Oleksandr Zinenko, Theodoros Theodoridis, Priya Goyal, Zachary DeVito, William S. Moses, Sven Verdoolaege, Andrew Adams, and Albert Cohen. 2018. *Tensor Comprehensions: Framework-Agnostic High-Performance Machine Learning Abstractions*. Technical Report. 12 pages. arXiv:1802.04730 http://arxiv.org/abs/1802.04730

Anand Venkat, Mary Hall, and Michelle Strout. 2015. Loop and Data Transformations for Sparse Matrix Code. In *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI 2015)*. 521–532. https://doi.org/10.1145/2737924.2738003

Michael J Wolfe. 1982. *Optimizing Supercompilers for Supercomputers*. Ph.D. Dissertation. University of Illinois at Urbana-Champaign.

David Wonnacott and William Pugh. 1995. Nonlinear array dependence analysis. In *Proc. Third Workshop on Languages, Compilers and Run-Time Systems for Scalable Computers*.

Carl Yang, Aydın Buluç, and John D. Owens. 2018. Design Principles for Sparse Matrix Multiplication on the GPU. In *Euro-Par 2018: Parallel Processing*, Marco Aldinucci, Luca Padovani, and Massimo Torquati (Eds.). Springer International Publishing, Cham, 672–687.

Yunming Zhang, Mengjiao Yang, Riyadh Baghdadi, Shoaib Kamil, Julian Shun, and Saman Amarasinghe. 2018. GraphIt: A High-Performance Graph DSL. *Proc. ACM Program. Lang.* 2, OOPSLA, Article 121 (Oct. 2018), 30 pages. https://doi.org/10.1145/3276491

A APPENDIX

A.1 Scheduled TACO SPMV CPU (Figure 21)

```
int CHUNK_SIZE = 32;

Format CSR({Dense, Sparse});
Tensor<double> A("A", {NUM_I, NUM_J}, CSR);
Tensor<double> x("x", {NUM_J}, {Dense});
Tensor<double> y("y", {NUM_I}, {Dense});

IndexVar i("i"), j("j");
y(i) = A(i, j) * x(j);

IndexVar i0("i0"), i1("i1");
IndexStmt stmt = y.getAssignment().concretize();
stmt = stmt.split(i, i0, i1, CHUNK_SIZE)
  .order({i0, i1, j})
  .parallelize(i0, ParallelUnit::CPUThread, OutputRaceStrategy::NoRaces);
```

A.2 Scheduled TACO SPMV GPU (Figure 22)

```
int NNZ_PER_THREAD = 7;
int NNZ_PER_WARP = 7 * 32;
int NNZ_PER_TB = 7 * 512;

Format CSR({Dense, Sparse});
Tensor<double> A("A", {NUM_I, NUM_J}, CSR);
Tensor<double> x("x", {NUM_J}, {Dense});
Tensor<double> y("y", {NUM_I}, {Dense});

IndexVar i("i"), j("j");
IndexExpr precomputedExpr = A(i, j) * x(j);
y(i) = precomputedExpr;

IndexVar f("f"), fpos("fpos"), fpos1("fpos1"), fpos2("fpos2");
IndexVar block("block"), warp("warp"), thread("thread");
IndexVar thread_nz("thread_nz"), thread_nz_pre("thread_nz_pre");
TensorVar precomputed("precomputed",
    Type(Float64, {Dimension(thread_nz)}), taco::dense);
IndexStmt stmt = y.getAssignment().concretize();
stmt = stmt.fuse(i, j, f)
  .pos(f, fpos, A(i, j))
  .split(fpos, block, fpos1, NNZ_PER_TB)
  .split(fpos1, warp, fpos2, NNZ_PER_WARP)
  .split(fpos2, thread, thread_nz, NNZ_PER_THREAD)
  .order({block, warp, thread, thread_nz})
  .precompute(precomputedExpr, thread_nz, thread_nz_pre, precomputed)
  .unroll(thread_nz_pre, NNZ_PER_THREAD)
  .parallelize(block, ParallelUnit::GPUBlock,
                      OutputRaceStrategy::IgnoreRaces)
  .parallelize(warp, ParallelUnit::GPUWarp,
                     OutputRaceStrategy::IgnoreRaces)
  .parallelize(thread, ParallelUnit::GPUThread,
                       OutputRaceStrategy::Atomics);
```

A.3 Scheduled TACO SPMSPV CPU (Figure 23)

```
int CHUNK_SIZE = 32;

Format CSC({Dense, Sparse}, {1, 0});
Tensor<double> A("A", {NUM_I, NUM_J}, CSC);
Tensor<double> x("x", {NUM_J}, {Sparse});
Tensor<double> y("y", {NUM_I}, {Dense});

IndexVar i("i"), j("j");
y(i) = A(i, j) * x(j);

IndexVar jpos("jpos"), jpos0("jpos0"), jpos1("jpos1");
IndexStmt stmt = y.getAssignment().concretize();
stmt = stmt.order({j,i})
  .pos(j, jpos, x(j))
  .split(jpos, jpos0, jpos1, CHUNK_SIZE)
  .parallelize(jpos0, ParallelUnit::CPUThread,
                   OutputRaceStrategy::Atomics);
```

A.4 Scheduled TACO SPMSPV GPU (Figure 24)

```
int CHUNK_SIZE = 32;
int BLOCK_SIZE = 512;

Format CSC({Dense, Sparse}, {1, 0});
Tensor<double> A("A", {NUM_I, NUM_J}, CSC);
Tensor<double> x("x", {NUM_J}, {Sparse});
Tensor<double> y("y", {NUM_I}, {Dense});

IndexVar i("i"), j("j");
y(i) = A(i, j) * x(j);

IndexVar jpos("jpos"), jpos_bound("jpos_bound");
IndexVar block("block"), thread("thread");
IndexStmt stmt = y.getAssignment().concretize();
stmt = stmt.order({j,i})
  .pos(j, jpos, x(j))
  .bound(jpos, jpos_bound, 0, BoundType::MinExact)
  .split(jpos_bound, block, thread, BLOCK_SIZE)
  .reorder({block, thread, i})
  .parallelize(block, ParallelUnit::GPUBlock,
                   OutputRaceStrategy::IgnoreRaces)
  .parallelize(thread, ParallelUnit::GPUThread,
                    OutputRaceStrategy::Atomics);
```

A.5 Scheduled TACO SPMM CPU (Figure 29)

```
int CHUNK_SIZE = 8;
int TILE_SIZE = 8;

Format CSR({Dense, Sparse});
Tensor<double> A("A", {NUM_I, NUM_J}, CSR);
Tensor<double> B("B", {NUM_J, NUM_K}, {Dense, Dense});
Tensor<double> C("C", {NUM_I, NUM_K}, {Dense, Dense});

IndexVar i("i"), j("j"), k("k");
C(i, k) = A(i, j) * B(j, k);

IndexVar i0("i0"), i1("i1");
IndexVar jpos("jpos"), jpos0("jpos0"), jpos1("jpos1");
IndexStmt stmt = C.getAssignment().concretize();
stmt = stmt.split(i, i0, i1, CHUNK_SIZE)
  .pos(j, jpos, A(i,j))
  .split(jpos, jpos0, jpos1, TILE_SIZE)
  .order({i0, i1, jpos0, k, jpos1})
  .parallelize(i0, ParallelUnit::CPUThread, OutputRaceStrategy::NoRaces)
  .parallelize(k, ParallelUnit::CPUVector, OutputRaceStrategy::IgnoreRaces);
```

A.6 Scheduled TACO SPMM GPU (Figure 30)

```
int NUM_COLS = 128;
int NNZ_PER_WARP = 16;
int NNZ_PER_TB = 16 * (512 / 32);

Format CSR({Dense, Sparse});
Tensor<double> A("A", {NUM_I, NUM_J}, CSR);
Tensor<double> B("B", {NUM_J, NUM_K}, {Dense, Dense});
Tensor<double> C("C", {NUM_I, NUM_K}, {Dense, Dense});

IndexVar i("i"), j("j"), k("k");
C(i, k) = A(i, j) * B(j, k);

IndexVar f("f"), fpos("fpos"), block("block"), fpos1("fpos1"), warp("warp");
IndexVar nnz_pre("nnz_pre"), nnz("nnz");
IndexVar dense_val_unbounded("dense_val_unbounded");
IndexVar dense_val("dense_val"), thread("thread"), thread_nz("thread_nz");
TensorVar precomputed("precomputed", Type(Float64,
                      {Dimension(nnz)}), taco::dense);
IndexStmt stmt = C.getAssignment().concretize();
stmt = stmt.order({i, j, k})
  .fuse(i, j, f)
  .pos(f, fpos, A(i, j))
  .split(fpos, block, fpos1, NNZ_PER_TB)
  .split(fpos1, warp, nnz, NNZ_PER_WARP)
  .split(k, dense_val_unbounded, thread, WARP_SIZE)
  .bound(dense_val_unbounded, dense_val,
         NUM_COLS / WARP_SIZE, BoundType::MaxExact);
  .order({block, warp, dense_val, thread, nnz})
  .parallelize(block, ParallelUnit::GPUBlock,
                      OutputRaceStrategy::IgnoreRaces)
  .parallelize(warp, ParallelUnit::GPUWarp,
                     OutputRaceStrategy::IgnoreRaces)
  .parallelize(thread, ParallelUnit::GPUThread,
                       OutputRaceStrategy::Atomics);
```

A.7 Scheduled TACO SDDMM CPU (Figure 27)

```
int CHUNK_SIZE = 8;

Format CSR({Dense, Sparse});
Tensor<double> A("A", {NUM_I, NUM_K}, CSR);
Tensor<double> B("B", {NUM_I, NUM_K}, CSR);
Tensor<double> C("C", {NUM_I, NUM_J}, {Dense, Dense});
Tensor<double> D("D", {NUM_J, NUM_K}, Format({Dense, Dense},{1,0}));

IndexVar i("i"), j("j"), k("k");
A(i,k) = B(i,k) * C(i,j) * D(j,k);

IndexStmt stmt = A.getAssignment().concretize();
IndexVar i0("i0"), i1("i1");
IndexVar kpos0("kpos0"), kpos1("kpos1");
stmt = stmt.split(i, i0, i1, CHUNK_SIZE)
  .parallelize(i0, ParallelUnit::CPUThread,
                OutputRaceStrategy::NoRaces);
```

A.8 Scheduled TACO SDDMM GPU (Figure ??)

```
int NUM_COLS = 128;
int NNZ_PER_WARP = 16;
int NNZ_PER_TB = 16 * (512 / 32);

Format CSR({Dense, Sparse});
Tensor<double> A("A", {NUM_I, NUM_K}, CSR);
Tensor<double> B("B", {NUM_I, NUM_K}, CSR);
Tensor<double> C("C", {NUM_I, NUM_J}, {Dense, Dense});
Tensor<double> D("D", {NUM_J, NUM_K}, Format({Dense, Dense},{1,0}));

IndexVar i("i"), j("j"), k("k");
A(i,k) = B(i,k) * C(i,j) * D(j,k);

IndexStmt stmt = A.getAssignment().concretize();

IndexVar f("f"), fpos("fpos"), block("block"), fpos1("fpos1"), warp("warp");
IndexVar nnz_pre("nnz_pre"), nnz("nnz");
IndexVar dense_val_unbounded("dense_val_unbounded");
IndexVar dense_val("dense_val"), thread("thread"), nnz("nnz");
stmt = stmt.reorder({i, k, j})
  .fuse(i, k, f)
  .pos(f, fpos, B(i,k))
  .split(fpos, block, fpos1, NNZ_PER_TB)
  .split(fpos1, warp, nnz, NNZ_PER_WARP)
  .split(j, dense_val_unbounded, thread, WARP_SIZE)
  .bound(dense_val_unbounded, dense_val,
         NUM_COLS / WARP_SIZE, BoundType::MaxExact)
  .reorder({block, warp, nnz, thread, dense_val})
  .unroll(dense_val, CO_FACTOR)
  .parallelize(block, ParallelUnit::GPUBlock,
                      OutputRaceStrategy::IgnoreRaces)
  .parallelize(warp, ParallelUnit::GPUWarp,
                     OutputRaceStrategy::Atomics)
  .parallelize(thread, ParallelUnit::GPUThread,
                       OutputRaceStrategy::ParallelReduction);
```

A.9 Scheduled TACO TTV CPU (Figure ??)

```
int CHUNK_SIZE = 8;

Format CSR({Dense, Sparse});
Tensor<double> A("A", {NUM_I, NUM_J}, CSR);
Tensor<double> B("B", {NUM_I, NUM_J, NUM_K}, {Dense, Sparse, Sparse});
Tensor<double> c("c", {NUM_K}, {Dense});
A(i,j) = B(i,j,k) * c(k);

IndexVar i("i"), j("j"), k("k");
A(i,j) = B(i,j,k) * c(k);

IndexStmt stmt = A.getAssignment().concretize();
IndexVar f("f"), fpos("fpos"), chunk("chunk"), fpos2("fpos2");
stmt = stmt.fuse(i, j, f)
  .pos(f, fpos, B(i,j,k))
  .split(fpos, chunk, fpos2, CHUNK_SIZE)
  .order({chunk, fpos2, k})
  .parallelize(chunk, ParallelUnit::CPUThread,
                   OutputRaceStrategy::NoRaces);
```

A.10 Scheduled TACO TTV GPU (Figure ??)

```
int NNZ_PER_WARP = 16;
int NNZ_PER_TB = 16 * (512 / 32);

Format CSR({Dense, Sparse});
Tensor<double> A("A", {NUM_I, NUM_J}, CSR);
Tensor<double> B("B", {NUM_I, NUM_J, NUM_K}, {Dense, Sparse, Sparse});
Tensor<double> c("c", {NUM_K}, {Dense});

IndexVar i("i"), j("j"), k("k");
IndexExpr precomputedExpr = B(i,j,k) * c(k);
A(i,j) = precomputedExpr;

IndexStmt stmt = A.getAssignment().concretize();
IndexVar jk("jk"), f("f"), fpos("fpos"), block("block");
IndexVar fpos1("fpos1"), warp("warp"), fpos2("fpos2");
IndexVar thread("thread"), thread_nz("thread_nz");
IndexVar thread_nz_pre("thread_nz_pre");

TensorVar precomputed("precomputed", Type(Float64,
          {Dimension(thread_nz)}), taco::dense);

stmt = stmt.fuse(j, k, jk)
  .fuse(i, jk, f)
  .pos(f, fpos, B(i,j,k))
  .split(fpos, block, fpos1, NNZ_PER_TB)
  .split(fpos1, warp, fpos2, NNZ_PER_WARP)
  .split(fpos2, thread, thread_nz, NNZ_PER_WARP/WARP_SIZE)
  .reorder({block, warp, thread, thread_nz})
  .precompute(precomputedExpr, thread_nz, thread_nz_pre, precomputed)
  .unroll(thread_nz_pre, NNZ_PER_WARP/WARP_SIZE)
  .parallelize(block, ParallelUnit::GPUBlock,
                      OutputRaceStrategy::IgnoreRaces)
  .parallelize(warp, ParallelUnit::GPUWarp,
                     OutputRaceStrategy::IgnoreRaces)
  .parallelize(thread, ParallelUnit::GPUThread,
                       OutputRaceStrategy::Atomics);
```

A.11 Scheduled TACO MTTKRP Order-3 CPU (Figure 31)

```
int CHUNK_SIZE = 32;
IndexVar i("i"), j("j"), k("k"), l("l");

Tensor<double> A("A", {NUM_I, NUM_J}, {Dense, Dense});
Tensor<double> B("B", {NUM_I, NUM_K, NUM_L}, {Dense, Sparse, Sparse});
Tensor<double> C("C", {NUM_K, NUM_J}, {Dense, Dense});
Tensor<double> D("D", {NUM_L, NUM_J}, {Dense, Dense});

IndexExpr precomputedExpr = B(i,k,l) * D(l,j);
A(i,j) = precomputedExpr * C(k,j);

IndexVar i1("i1"), i2("i2");
IndexStmt stmt = A.getAssignment().concretize();
TensorVar precomputed("precomputed", Type(Float64,
                      {Dimension(j)}), taco::dense);
stmt = stmt.order({i, k, l, j})
        .precompute(precomputedExpr, j, j, precomputed)
        .split(i, i1, i2, CHUNK_SIZE)
        .parallelize(i1, ParallelUnit::CPUThread,
                         OutputRaceStrategy::NoRaces);
```

A.12 Scheduled TACO MTTKRP Order-4 CPU (Figure 31)

```
int CHUNK_SIZE = 32;
IndexVar i("i"), j("j"), k("k"), l("l"), m("m");
IndexVar i1("i1"), i2("i2");

Tensor<double> A("A", {NUM_I, NUM_J}, {Dense, Dense});
Tensor<double> B("B", {NUM_I, NUM_K, NUM_L, NUM_M},
                      {Dense, Sparse, Sparse, Sparse});
Tensor<double> C("C", {NUM_K, NUM_J}, {Dense, Dense});
Tensor<double> D("D", {NUM_L, NUM_J}, {Dense, Dense});
Tensor<double> E("E", {NUM_M, NUM_J}, {Dense, Dense});

IndexExpr BE = B(i,k,l,m) * E(m,j);
IndexExpr BDE = BE * D(l, j);
A(i,j) = BDE * C(k,j);
IndexStmt stmt = A.getAssignment().concretize();
TensorVar BE_workspace("BE_workspace", Type(Float64,
                       {Dimension(j)}), taco::dense);
TensorVar BDE_workspace("BDE_workspace", Type(Float64,
                        {Dimension(j)}), taco::dense);
stmt = stmt.order({i, k, l, m, j})
  .precompute(BDE, j, j, BDE_workspace)
  .precompute(BE, j, j, BE_workspace)
  .split(i, i1, i2, CHUNK_SIZE)
  .parallelize(i1, ParallelUnit::CPUThread, OutputRaceStrategy::NoRaces);
```

A.13 Scheduled TACO MTTKRP Order-5 CPU (Figure 31)

```
int CHUNK_SIZE = 32;
IndexVar i("i"), j("j"), k("k"), l("l"), m("m"), n("n");
IndexVar i1("i1"), i2("i2");

Tensor<double> A("A", {NUM_I, NUM_J}, {Dense, Dense});
Tensor<double> B("B", {NUM_I, NUM_K, NUM_L, NUM_M, NUM_N},
                      {Dense, Sparse, Sparse, Sparse, Sparse});
Tensor<double> C("C", {NUM_K, NUM_J}, {Dense, Dense});
Tensor<double> D("D", {NUM_L, NUM_J}, {Dense, Dense});
Tensor<double> E("E", {NUM_M, NUM_J}, {Dense, Dense});
Tensor<double> F("F", {NUM_N, NUM_J}, {Dense, Dense});

IndexExpr BF = B(i,k,l,m,n) * F(n,j);
IndexExpr BEF = BF * E(m,j);
IndexExpr BDEF = BEF * D(l, j);
A(i,j) = BDEF * C(k,j);
IndexStmt stmt = A.getAssignment().concretize();
TensorVar BF_workspace("BF_workspace", Type(Float64,
                       {Dimension(j)}), taco::dense);
TensorVar BEF_workspace("BEF_workspace", Type(Float64,
                        {Dimension(j)}), taco::dense);
TensorVar BDEF_workspace("BDEF_workspace", Type(Float64,
                        {Dimension(j)}), taco::dense);
stmt = stmt.order({i, k, l, m, n, j})
  .precompute(BDEF, j, j, BDEF_workspace)
  .precompute(BEF, j, j, BEF_workspace)
  .precompute(BF, j, j, BF_workspace)
  .split(i, i1, i2, CHUNK_SIZE)
  .parallelize(i1, ParallelUnit::CPUThread, OutputRaceStrategy::NoRaces);
```

A.14 Scheduled TACO MTTKRP GPU (Figure 32)

```
int NUM_COLS = 32;
int NNZ_PER_WARP = 4;
int NNZ_PER_TB = 4 * (512 / WARP_SIZE);

Tensor<double> A("A", {NUM_I, NUM_J}, {Dense, Dense});
Tensor<double> B("B", {NUM_I, NUM_K, NUM_L}, {Dense, Sparse, Sparse});
Tensor<double> C("C", {NUM_K, NUM_J}, {Dense, Dense});
Tensor<double> D("D", {NUM_L, NUM_J}, {Dense, Dense});

IndexVar i("i"), j("j"), k("k");
A(i,j) = B(i,k,l) * C(k,j) * D(l,j);

IndexVar kl("kl"), f("f"), fpos("fpos"), block("block"), fpos1("fpos1");
IndexVar warp("warp"), nnz("nnz"), dense_val("dense_val");
IndexVar dense_val_unbounded("dense_val_unbounded"), thread("thread");
IndexStmt stmt = A.getAssignment().concretize();
stmt = stmt.order({i,k,l,j})
  .fuse(k, l, kl)
  .fuse(i, kl, f)
  .pos(f, fpos, B(i, k, l))
  .split(fpos, block, fpos1, NNZ_PER_TB)
  .split(fpos1, warp, nnz, NNZ_PER_WARP)
  .split(j, dense_val_unbounded, thread, WARP_SIZE)
  .bound(dense_val_unbounded, dense_val,
         NUM_COLS / WARP_SIZE, BoundType::MaxExact)
  .order({block, warp, dense_val, thread, nnz})
  .parallelize(block, ParallelUnit::GPUBlock,
                      OutputRaceStrategy::IgnoreRaces)
  .parallelize(warp, ParallelUnit::GPUWarp,
                     OutputRaceStrategy::IgnoreRaces)
  .parallelize(thread, ParallelUnit::GPUThread,
                       OutputRaceStrategy::Atomics);
```

A.15 SpMV Thread per Row on GPU (Section 7.3)

```
Format CSR({Dense, Sparse});
Tensor<double> A("A", {NUM_I, NUM_J}, CSR);
Tensor<double> x("x", {NUM_J}, {Dense});
Tensor<double> y("y", {NUM_I}, {Dense});

IndexVar i("i"), j("j");
y(i) = A(i, j) * x(j);

IndexVar block("block"), warp("warp"), thread("thread");
IndexVar thread_nz("thread_nz"), i1("i1"), jpos("jpos");
IndexVar block_row("block_row"), warp_row("warp_row");
IndexStmt stmt = y.getAssignment().concretize();
stmt = stmt.split(i, block, thread, ROWS_PER_TB)
  .parallelize(block, ParallelUnit::GPUBlock,
                      OutputRaceStrategy::NoRaces)
  .parallelize(thread, ParallelUnit::GPUThread,
                       OutputRaceStrategy::NoRaces);
```

A.16 SpMV Warp per Row on GPU (Section 7.3)

```
Format CSR({Dense, Sparse});
Tensor<double> A("A", {NUM_I, NUM_J}, CSR);
Tensor<double> x("x", {NUM_J}, {Dense});
Tensor<double> y("y", {NUM_I}, {Dense});

IndexVar i("i"), j("j");
IndexExpr precomputedExpr = A(i, j) * x(j);
y(i) = precomputedExpr;

IndexVar block("block"), warp("warp"), thread("thread");
IndexVar i1("i1"), jpos("jpos"), block_row("block_row");
IndexVar warp_row("warp_row"), thread_nz("thread_nz");
TensorVar precomputed("precomputed",
    Type(Float64, {Dimension(thread_nz)}), taco::dense);
IndexStmt stmt = y.getAssignment().concretize();
stmt = stmt.split(i, block, block_row, ROWS_PER_TB)
  .split(block_row, warp_row, warp, BLOCK_SIZE / WARP_SIZE)
  .pos(j, jpos, A(i, j))
  .split(jpos, thread_nz, thread, WARP_SIZE)
  .order({block, warp, warp_row, thread, thread_nz})
  .parallelize(block, ParallelUnit::GPUBlock,
                      OutputRaceStrategy::IgnoreRaces)
  .parallelize(warp, ParallelUnit::GPUWarp,
                     OutputRaceStrategy::IgnoreRaces)
  .parallelize(thread, ParallelUnit::GPUThread,
                       OutputRaceStrategy::Temporary);
```

A.17 SpMV on GPU with no Unrolling (Section 7.3)

```
Format CSR({Dense, Sparse});
Tensor<double> A("A", {NUM_I, NUM_J}, CSR);
Tensor<double> x("x", {NUM_J}, {Dense});
Tensor<double> y("y", {NUM_I}, {Dense});

IndexVar i("i"), j("j");
IndexExpr precomputedExpr = A(i, j) * x(j);
y(i) = precomputedExpr;

IndexVar f("f"), fpos("fpos"), fpos1("fpos1"), fpos2("fpos2");
IndexVar block("block"), warp("warp");
IndexVar thread("thread"), thread_nz("thread_nz");
IndexStmt stmt = y.getAssignment().concretize();
stmt = stmt.fuse(i, j, f)
  .pos(f, fpos, A(i, j))
  .split(fpos, block, fpos1, NNZ_PER_TB)
  .split(fpos1, warp, fpos2, NNZ_PER_WARP)
  .split(fpos2, thread, thread_nz, NNZ_PER_THREAD)
  .order({block, warp, thread, thread_nz})
  .parallelize(block, ParallelUnit::GPUBlock,
                      OutputRaceStrategy::IgnoreRaces)
  .parallelize(warp, ParallelUnit::GPUWarp,
                     OutputRaceStrategy::IgnoreRaces)
  .parallelize(thread, ParallelUnit::GPUThread,
                       OutputRaceStrategy::Atomics);
```

A.18 SpMM on CPU Tiled (Section ??)

```
Format CSR({Dense, Sparse});
Tensor<double> A("A", {NUM_I, NUM_J}, CSR);
Tensor<double> B("B", {NUM_J, NUM_K}, {Dense, Dense});
Tensor<double> C("C", {NUM_I, NUM_K}, {Dense, Dense});

IndexVar i("i"), j("j"), k("k");
C(i, k) = A(i, j) * B(j, k);

IndexVar i0("i0"), i1("i1");
IndexVar jpos("jpos"), jpos0("jpos0"), jpos1("jpos1");
IndexStmt stmt = C.getAssignment().concretize();
stmt = stmt.pos(j, jpos, A(i,j))
  .split(jpos, jpos0, jpos1, TILE_SIZE)
  .order({i, jpos0, k, jpos1});
```

A.19 SpMM on CPU No Tiling Section ??)

```
Format CSR({Dense, Sparse});
Tensor<double> A("A", {NUM_I, NUM_J}, CSR);
Tensor<double> B("B", {NUM_J, NUM_K}, {Dense, Dense});
Tensor<double> C("C", {NUM_I, NUM_K}, {Dense, Dense});

IndexVar i("i"), j("j"), k("k");
C(i, k) = A(i, j) * B(j, k);

IndexStmt stmt = C.getAssignment().concretize();
```